RWTH Aachen

Department of Computer Science
Technical Report

RULE'04
Fifth International Workshop on Rule-Based Programming

Proceedings

Slim Abdennadher and Christophe Ringeissen (eds.)

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2004- 04

RWTH Aachen · Department of Computer Science · June 2004

Preface

This volume contains the proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004), part of the Federated Conference on Rewriting, Deduction and Programming (RDP 2004), held during May 31 – June 5, 2004, in Aachen, Germany.

Rule-based programming is currently experiencing a renewed period of growth with the emergence of new concepts and systems that allow a better understanding and better usability. On the theoretical side, after the in-depth study of rewriting concepts during the eighties, the nineties saw the emergence of the general concepts of rewriting logic and of the rewriting calculus. On the practical side, new languages and systems such as ASF+SDF, BURG, CHRS, Claire, ELAN, Maude, and Stratego have shown that rules are a useful programming tool.

The practical application of rule-based programming prompts research into the algorithmic complexity and optimization of rule-based programs as well as into the expressivity, semantics and implementation of rule-based languages.

The purpose of this workshop is to bring together researchers from the various communities working on rule-based programming to foster fertilisation between theory and practice, as well as to favour the growth of this programming paradigm.

The previous editions of the RULE workshop were held at Valencia (2003) during the RDP conference "Rewriting, Deduction and Programming", and Pittsburg (2002), Firenze (2001), Montreal (2000) during the PLI Conferences "Principles, Logics, and Implementations of high-level programming languages".

There were 14 submissions of overall high quality, authored by researchers from countries including Austria, France, Germany, Italy, Japan, Spain, Thailand, UK, and USA. All submissions were thoroughly evaluated and an electronic program committee meeting was held through the Internet. The program committee selected 9 papers, which can be found in this volume.

We would like to thank the program committee members and all the referees for their care and time in evaluating and selecting the submitted papers, Juergen Giesl and the RDP organizing committee for taking care of the local organization. Post-workshop proceedings will be published in the Electronic Notes in Theoretical Computer Science (ENTCS). We would like to thank Professor Michael Mislove and Elsevier for providing us this opportunity.

Slim Abdennadher,
Christophe Ringeissen.

Workshop Organizers

Slim Abdennadher	German U. in Cairo, Egypt
Christophe Ringeissen	LORIA-INRIA, France

Program Committee

Slim Abdennadher	German U. in Cairo, Egypt
Mark van den Brand	CWI, The Netherlands
Steven Eker	SRI International, USA
Thom Fruehwirth	U. Ulm, Germany
Michael Hanus	U. Kiel, Germany
Jan Maluszynski	U. Linkoping, Sweden
Narciso Martí-Oliet	UCM, Spain
Olivier Michel	U. Evry, France
Christophe Ringeissen	LORIA-INRIA, France

External Reviewers

Horatiu Cirstea
Grit Denker
Wlodzimierz Drabent
Christine Eisenbeis
Olivier Fissore
Christophe Gaston
Florent Jacquemard
Pierre-Etienne Moreau
Ulf Nilsson
Miguel Palomino
Alberto Verdejo
Laurent Vigneron

Contents

Playing with Maude

Miguel Palomino, Narciso Martí-Oliet, and Alberto Verdejo

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
{miguelpt,narciso,alberto}@sip.ucm.es

Abstract This paper is an introduction to rule-based programming in Maude. We illustrate in particular the use of operator attributes to structure the state of a system, and the difference between equations and rules. We use mathematical games and puzzles for our examples illustrating the expressive power of Maude.
Keywords: Rule-based programming, Maude, puzzles.

1 Presentation

Though not a formal branch of Mathematics, mathematical games and puzzles of all sorts constitute an important subclass in the realm of mathematical problems, with a long tradition and extensive literature [1,4,11,12]. Most of them have in common the fact that they are easy to state and understand, which does not mean that a precise solution is always trivial to find.

Here we make use of a collection of these problems to introduce Maude, a specification language that efficiently implements rewriting logic [10], which includes equational logic as a sublogic. We are not concerned with finding neat and concise mathematical solutions, but rather we would like to find out how easy is to express those problems in the rewriting logic formalism underlying Maude, and how far we can go in their resolution by the use of just brute force and as less ingenuity as possible. In this regard, a clear conclusion is that many of these problems can be represented/specified in Maude in a much simpler way than it would be possible in other more conventional languages. Among the main reasons why the rule-based programming paradigm supported by Maude allows so natural a representation of many problems, we would like to mention:

- The syntax is user-definable to a great extent, which allows to choose the more appropriate one for each problem. In particular, operators declared by the user can have attributes like associativity and commutativity, which makes *multiset* rewriting trivial. All the specifications in this paper make essential use of this feature.
- An expressive version of *equational logic* allows a (first-order) version of functional programming to describe the static aspects of a system.
- The dynamic aspects are described by means of rules that represent the possible transitions or *changes* in a system. Those rules need only specify the part of the system that actually changes, which makes them quite simple. This corresponds to the fact that *rewriting logic* is a logic very suitable for expressing concurrent action and change [7] in which the frame problem [6] has been avoided.
- The transitive closure of the relation defined by the rules is automatically computed by the Maude system. This, combined with the flexible `search` command, lets the user explore all computations starting at a given state.

On the other hand, it is also true that some of these examples suffer from the state explosion problem which makes it difficult to solve them just by checking all possible combinations.

Most of the problems introduced here are well-known and can be found (in some form or another) in a number of sources: see [11] for a classic reference on the subject, [4] for a delightful exposition on how to tackle these problems, [1] for an on-line presentation, or even [12] for more algebraic ones. In many cases, a clear mathematical solution exists, but not always, and anyway our goal is to show the ease with which Maude lends itself to the specification of these problems, and to try to solve them without much thinking.

This paper is thus an introduction to rule-based programming in Maude by means of a collection of puzzles showing the language's expressive power. Even though current Maude users no doubt will find the examples here to be very simple, those new to it may still find them attractive and be encouraged to use Maude for more "serious" applications. Anyway, even that would be too ambitious a goal: the main reason why this was written down was, plain and simple, to have some fun. And we hope that you will have some fun while reading it, too. All the examples can be downloaded from `http://maude.sip.ucm.es/games`.

2 A Brief Overview on Rewriting Logic and Maude

The motivation for this section is not to provide a crash course on rewriting logic and Maude. On the contrary, we only intend to give the flavor of the underlying theory and provide enough information so that the specifications of the examples in the next sections can be understood. For a thorough treatment we refer the interested reader to the paper in which rewriting logic was first presented [10], to the Maude manual [3], and to [9], where many more papers on rewriting logic are referenced.

Rewriting logic was proposed by Meseguer as a unified model for concurrency in the early nineties. Since then, it has proved its value as a logic *of* change as well as a logical and semantic framework [8]. As a consequence of that success some implementations were developed; the one we use is called Maude and can be obtained at `http://maude.cs.uiuc.edu` free of charge.

The states (or configurations) of a system, its static part, are specified in rewriting logic by means of an equational theory. The transitions, the dynamic part, are specified by means of rules that rewrite some terms (representing parts of a system) into others.

To illustrate both these ideas and Maude syntax consider the following example. We have some natural numbers written on a blackboard and we are allowed, at any given time, to replace any two of them by their arithmetic mean. In this case the static part corresponds to the representation of the blackboard and the numbers themselves. To represent the numbers we have first to declare their sort (or type) and then write the well-known Peano constructors.

```
sort Nat .
op 0 : -> Nat .
op s : Nat -> Nat .
```

Since we will also need to add numbers we declare an operator

```
op _+_ : Nat Nat -> Nat .
```

Note the use of Maude's mixfix syntax, with _ indicating where the arguments are to be written. Its behavior is defined inductively by means of the following two *equations*.

```
vars N M : Nat .
eq N + 0 = N .
eq N + s(M) = s(N + M) .
```

Division `_div_` would be defined analogously. As for the blackboard, it can be represented as a (nonempty) *multiset*, or bag, of numbers.

```
sort Blackboard .
subsort Nat < Blackboard .
op __ : Blackboard Blackboard -> Blackboard [assoc comm] .
```

The `subsort` declaration tells Maude that a single number constitutes a valid representation for the blackboard. Multiset union is represented with empty syntax `__`. Note that this operator has two *attributes*, `assoc` and `comm`, so that terms of sort `Blackboard` are considered *modulo* associativity and commutativity (e.g., `s(0) 0` and `0 s(0)` become indistinguishable).

Finally, the system's dynamics is specified by the single *rule*

```
rl [replace] : N M => (N + M) div s(s(0)) .
```

The word in brackets after the keyword `rl` is the rule's name and is optional. Note that it is enough to specify the behavior of the two numbers that are going to be erased, without considering the rest of the numbers in the blackboard.

The `rewrite` command can be used to *execute* the system, by means of an interpreter which applies the rules (using a default internal strategy) and stops when no rule can be applied.

```
Maude> rewrite s(s(s(s(s(0))))) s(s(0))) s(s(0)) .
result NzNat: s(s(s(s(0))))
```

But the numbers chosen to be replaced by their mean can be selected arbitrarily, that is, in a nondeterministic way, and this affects the final result. The `search` command can be used to explore the computation tree. It receives the term to be rewritten, the relation used to obtain final states (`=>*` for zero or more rewrites), and the final state (a new variable `N:Nat` of sort `Nat` in this case). The computation tree is traversed in a *breadth-first* way.

```
Maude> search s(s(s(s(s(s(0)))))) s(s(s(0))) s(s(0)) =>* N:Nat .

Solution 1 (state 4)
N --> s(s(s(s(0))))
Solution 2 (state 5)
N --> s(s(s(0)))
No more solutions.
```

3 The Hopping Rabbits

Two teams of n rabbits each, wearing T-shirts marked with a cross and a circle respectively, are placed facing each other on a row with $2n + 1$ positions. The `x`-team occupies the first n positions and the `o`-team the last n; the middle one is left empty. The goal is to swap the positions of the teams (the players of each team are indistinguishable), with the rabbits moving according to the rules of the game:

1. Rabbits from the `x`-team can only move rightward, and rabbits from the `o`-team can only move leftward.
2. A rabbit is allowed to advance one position if that position is empty.
3. A rabbit can jump over a rival if the position behind it is free.

This puzzle is also known as the *toads and frogs puzzle* or *traffic jam*. It is possible to generalize the puzzle so that the number of elements in each team is different [11].

We represent the state of the game as a nonempty *list* of rabbits, specified by means of an *associative* append operator written with empty syntax `__`; note that associativity is built into the list constructor `__` using the attribute `assoc`. Each rabbit is represented as a constant `x` or `o`, according to its team, and the constant `free` represents the empty position.

The initial state of the game depends on the number n of rabbits in each team. This is specified by means of an operator `initial` that builds the appropriate initial state, as indicated in the equations below to define this operator. Notice how equations are used to define the initial state, while rules are used to represent the transitions corresponding to the legal moves in the game. As pointed out in the introduction, we use two logics, each for a different purpose: equational logic for the static aspects of a system, and rewriting logic for the dynamic aspects.

Since the rules need only specify the parts of the system that change, in this game we only need to consider the positions adjacent to the free position. Thus there are four possible legal moves, and each one is represented by a rule whose label identifies the corresponding move.

The complete specification is then as follows; the second line imports the predefined module `NAT` that specifies the natural numbers with the usual notation and arithmetic operations [3, Section 7.2].

```
mod RABBIT-HOP is
  protecting NAT .
  sorts Rabbit RabbitList .
  subsort Rabbit < RabbitList .

  ops x o free : -> Rabbit .
  op __ : RabbitList RabbitList -> RabbitList [assoc] .
  op initial : Nat -> RabbitList .

  var N : Nat .
  vars L R : RabbitList .
  var B : Rabbit .

  eq initial(0) = free .
  eq initial(s(N)) = x initial(N) o .

  rl [xAdvances] : x free => free x .
  rl [xJumps] : x o free => free o x .
  rl [oAdvances] : free o => o free .
  rl [oJumps] : free x o => o x free .
endm
```

Since we are interested in knowing how to reach the final position, and in general there are several possible rules that can be applied in a given state, we use the `search` command. The example below is with $n = 3$.

```
Maude> search initial(3) =>* o o o free x x x .

Solution 1 (state 71)
empty substitution

No more solutions.
```

The sequence of 15 steps leading to the final position can be obtained as follows, where we only show the beginning of the output.

```
Maude> show path 71 .
state 0, RabbitList: x x x free o o o
===[ rl x free => free x [label xAdvances] . ]===>
state 1, RabbitList: x x free x o o o
===[ rl free x o => o x free [label oJumps] . ]===>
state 4, RabbitList: x x o x free o o
===[ rl free o => o free [label oAdvances] . ]===>
state 9, RabbitList: x x o x o free o
...
```

4 The Josephus Problem

As related in [11], Flavius Josephus was a famous Jewish historian who, during the Jewish-Roman war in the first century, was trapped in a cave with a group of 40 Jewish soldiers surrounded by Romans. Legend has it that, preferring death to being captured, the Jews decided to gather in a circle and rotate a dagger around it so that every third remaining person would commit suicide. Apparently, Josephus was too keen to live and quickly found out the safe position.

The problem of finding that safe position can be modeled very easily in Maude. The circle representation becomes a (circular) *list* once the beginning position is chosen. The operator `__` is used to build nonempty lists of (nonzero) natural numbers (sort `NzNat` in Maude's predefined module `NAT`) representing the original positions of the soldiers in the circle; its associativity is specified with the attribute `assoc`. Though it is not explicitly represented, we assume that the dagger is initially at position 1.

The idea then consists in continually taking the first two elements in the list and moving them to the end of it while "killing" the third one; when only two are left, the one who initially has the dagger has to commit suicide. Note that in this way the dagger remains always implicitly located at the beginning of the list. Since we need to keep track of both the actual start and end of the list, we enclose it using the operator `{_}`. In this way, rewriting takes place only at the top of the term that represents the state.

As in the previous example, the operator `initial` and the corresponding equations are used to build the initial state. Then the rules correspond to the system transitions; we have got three rules for the cases when there are two, three, or more soldiers in the circle. Notice that there is no rule corresponding to a single soldier list, because this is the situation in which the last remaining soldier decides not to follow the rules of the game. In Maude modules, several rules can have the same label, and comments are introduced with `---`.

```
mod JOSEPHUS is
  protecting NAT .
  sorts Moriturem Circle .
  subsort NzNat < Moriturem .

  op __ : Moriturem Moriturem -> Moriturem [assoc] .
  op {_} : Moriturem  -> Circle .
  op initial : NzNat -> Moriturem .

  var M : Moriturem .
  vars I1 I2 I3 N : NzNat .
```

```
  eq initial(1) = 1 .
  eq initial(s(N)) = initial(N) s(N) .

  rl [kill] : { I1 I2 I3 M } => { M I1 I2 } .
  rl [kill] : { I1 I2 I3 } => { I1 I2 } . --- This rule is necessary
                                          --- because M cannot be empty
  rl [kill] : { I1 I2 } => { I2 } .
endm
```

Had we been in the same position as Josephus (and had we had a laptop to run Maude on it), we could have found out the safe spot by executing the command:

```
Maude> rewrite { initial(41) } .
result Circle: {31}
```

Note that at any moment until the end only one of the three rules can be applied, thus the final state is reached deterministically.

It is also easy to modify the program so that every i-th person commits suicide, where i is a parameter. The idea is the same, but because of the parameter now it is necessary to explicitly represent the dagger. For that, we use the constructor `dagger : NzNat NzNat -> Moriturem`, whose second argument stores the value of i while the first one acts as a counter: each time an element is moved from the beginning of the list to the end, the first argument is decreased by one; once it reaches 1, the element that is currently the head of the list is "killed" (removed from the list).

```
mod JOSEPHUS-GENERALIZED is
  protecting NAT .
  sorts Moriturem Circle .
  subsort NzNat < Moriturem .

  op dagger : NzNat NzNat -> Moriturem .
  op __ : Moriturem Moriturem -> Moriturem [assoc] .
  op {_} : Moriturem  -> Circle .
  op initial : NzNat NzNat -> Moriturem .

  var M : Moriturem .
  vars I I1 I2 N : NzNat .

  eq initial(1, I)  = dagger(I, I) 1 .
  eq initial(s(N), I) = initial(N, I) s(N) .

  rl [kill] : { dagger(1, I) I1 M } => { dagger(I, I) M } .
  rl [kill] : { dagger(s(N), I) I1 M } => { dagger(N, I) M I1 } .
  rl [kill] : { dagger(N, I) I1 } => { I1 } .  --- The last one throws the dagger away!
endm

Maude> rewrite { initial(41, 3) } .
result Circle: {31}
```

5 The Three Basins Puzzle

The following is a classic puzzle with a recent cameo in the 1995 Hollywood hit *Die Hard: With a Vengeance*. In the movie, McClane and Zeus have to deactivate a bomb by placing 4 gallons of water on a balance. The supply of water is unlimited, but they only have three basins with capacities of 3, 5, and 8 gallons, respectively.

The problem can be specified in Maude as follows. A basin is represented by means of the constructor `basin` with two natural numbers as arguments: the first one is the basin capacity and the second one is how much it is filled. We can think of a basin as an object with two attributes. This way of thinking leads to an *object-based* style of programming, where objects change their attributes as result of interacting with other objects; these interactions are represented as rules on *configurations* that are nonempty *multisets* of objects [3, Chapter 8]. The multiset constructor is written with empty syntax and declared with attributes `assoc` and `comm`. The constant `initial` defines the initial configuration.

At any given time we can either empty one of the basins, or fill it completely; the rules `empty` and `fill` below take care of this. When there is enough space in one of the basins to hold the current content of another, we can transfer all the water from this second one by using rule `transfer1`. Note that this is a *conditional rule* (introduced with keyword `crl`), with the condition at the end, after keyword `if`. The case when, after pouring one basin over another, there is still some water left is dealt with by the conditional rule `transfer2` (where the operator `sd` denotes the subtraction operation over natural numbers). These last two rules could be combined into a single one, but the result would not be so clear.

```
mod DIE-HARD is
  protecting NAT .
  sorts Basin BasinSet .
  subsort Basin < BasinSet .

  op basin : Nat Nat -> Basin .    --- Capacity / Content
  op __ : BasinSet BasinSet -> BasinSet [assoc comm] .
  op initial : -> BasinSet .

  vars M1 N1 M2 N2 : Nat .

  eq initial = basin(3, 0) basin(5, 0) basin(8,0) .

  rl [empty] : basin(M1, N1) => basin(M1, 0) .
  rl [fill] : basin(M1, N1) => basin(M1, M1) .
  crl [transfer1] : basin(M1, N1) basin(M2, N2) =>
                    basin(M1, 0) basin(M2, N1 + N2) if N1 + N2 <= M2 .
  crl [transfer2] : basin(M1, N1) basin(M2, N2) =>
                    basin(M1, sd(N1 + N2, M2)) basin(M2, M2) if N1 + N2 > M2 .
endm
```

We can now find out the shortest solution with the help of the `search` command, due to the breadth-first way of searching (the argument `[1]` tells Maude to look only for one solution). Notice that the *pattern* used after the arrow `=>*` represents any set of basins with one of them having 4 gallons.

```
Maude> search [1] initial =>* basin(N:Nat, 4) B:BasinSet .

Solution 1 (state 75)
B:BasinSet --> basin(3, 3) basin(8, 3)
N:Nat --> 5
```

The sequence of actions that leads to the solution can be seen with `show path 75`, where we omit part of the information about the rules used.

```
Maude> show path 75 .
state 0, BasinPack: basin(3, 0) basin(5, 0) basin(8, 0)
```

```
===[ rl ... fill ]===>
state 2, BasinPack: basin(3, 0) basin(5, 5) basin(8, 0)
===[ crl ... transfer2 ]===>
state 9, BasinPack: basin(3, 3) basin(5, 2) basin(8, 0)
===[ crl ... transfer1 ]===>
state 20, BasinPack: basin(3, 0) basin(5, 2) basin(8, 3)
===[ crl ... transfer1 ]===>
state 37, BasinPack: basin(3, 2) basin(5, 0) basin(8, 3)
===[ rl ... fill ]===>
state 55, BasinPack: basin(3, 2) basin(5, 5) basin(8, 3)
===[ crl ... transfer2 ]===>
state 75, BasinPack: basin(3, 3) basin(5, 4) basin(8, 3)
```

6 Crossing the Bridge

The four components of U2, the famous band of rock music, are in a tight situation. The concert starts in 17 minutes and in order to get to the stage they must first cross an old bridge through which only a maximum of two persons can walk over at the same time. It is already dark and, because of the bad condition of the bridge, to avoid falling into the darkness it is necessary to cross it with the help of a flashlight. Unfortunately, they only have one. Knowing that Bono, Edge, Adam, and Larry take 1, 2, 5, and 10 minutes, respectively, to cross the bridge, is there a way that they can make it to the concert on time?

The current state of the group can be represented in Maude by a *multiset* consisting of singers, the flashlight, and a watch to keep record of the time. The flashlight and the singers have a `Place` associated to them, indicating whether their current position is to the left or to the right of the bridge; each singer, in addition, also carries the time it takes him to cross the bridge. As in the previous example, this specification follows an object-based style of programming. We have an auxiliary operation `changePos` that is defined by means of two equations.

The traversing of the bridge is modeled by two rewrite rules: the first one for the case in which a single person crosses it, and the second one for when there are two. Note that for somebody to be allowed to cross, their position relative to the bridge must be the same as for the flashlight, which is represented by having the same variable `P` twice on the lefthand side of the rules. Also, since `__` is commutative, the condition in the second rule amounts to no loss of generality.

```
mod U2 is
  protecting NAT .
  sorts Singer Object Group Place .
  subsorts Singer Object < Group .

  ops left right : -> Place .
  op changePos : Place -> Place .
  op flashlight : Place -> Object .
  op watch : Nat -> Object .
  op singer : Nat Place -> Singer .
  op __ : Group Group -> Group [assoc comm] .
  op initial : -> Group .

  var P : Place .
  vars M N N1 N2 : Nat .
```

```
  eq initial = watch(0) flashlight(left)
               singer(1, left) singer(2, left) singer(5, left) singer(10, left) .

  eq changePos(left) = right .
  eq changePos(right) = left .

  rl [one-crosses] : watch(M) flashlight(P) singer(N, P) =>
                     watch(M + N) flashlight(changePos(P)) singer(N, changePos(P)) .
  crl [two-cross] : watch(M) flashlight(P) singer(N1, P) singer(N2, P) =>
                    watch(M + N1) flashlight(changePos(P)) singer(N1, changePos(P))
                    singer(N2, changePos(P))
                  if N1 > N2 .
endm
```

A solution can now be found quickly by looking for a state in which all singers (and the flashlight) are to the `right` of the bridge. Notice how the `search` command is invoked with a `such that` clause that allows to introduce a condition that solutions have to fulfill.

```
Maude> search [1] initial =>* flashlight(right) watch(N:Nat) singer(1, right)
                              singer(2, right) singer(5, right) singer(10, right)
       such that N:Nat <= 17 .

Solution 1 (state 402)
N --> 17
```

The solution takes exactly 17 minutes (a happy ending after all!) and the complete trace can be shown as follows:

```
Maude> show path 402 .
state 0, Group: flashlight(left) watch(0) singer(1, left) singer(2, left)
                singer(5, left) singer(10, left)
===[ crl ... two-cross ]===>
state 5, Group: flashlight(right) watch(2) singer(1, right) singer(2, right)
                singer(5, left) singer(10, left)
===[ rl ... one-crosses ]===>
state 15, Group: flashlight(left) watch(3) singer(1, left) singer(2, right)
                 singer(5, left) singer(10, left)
===[ crl ... two-cross ]===>
state 71, Group: flashlight(right) watch(13) singer(1, left) singer(2, right)
                 singer(5, right) singer(10, right)
===[ rl ... one-crosses ]===>
state 158, Group: flashlight(left) watch(15) singer(1, left) singer(2, left)
                  singer(5, right) singer(10, right)
===[ crl ... two-cross ]===>
state 402, Group: flashlight(right) watch(17) singer(1, right) singer(2, right)
                  singer(5, right) singer(10, right)
```

After sorting out the information, it becomes clear that Bono and Edge have to be the first to cross. Then Bono returns with the flashlight, which gives to Adam and Larry. Finally, Edge takes the flashlight back to Bono and they cross the bridge together for the last time.

Note that, in order for the `search` command to stop, we need to tell Maude to look only for one solution. Otherwise, it will continue exploring all possible combinations, increasingly taking a larger amount of time, and it will never end.

7 The Looping Chips

In the next game, taken from [1], four chips of different colors have been placed in consecutive places on a 12×1 board whose ends have been glued together. Each chip can be moved 5 places from its current location, either clockwise or counterclockwise, assuming the final position is empty. The goal is to arrange the chips in reverse order, over the original four squares.

The state is again represented by a *multiset* of `Place`s, with each `Place` determined by its position in the board and the color of the chip on it or `e` if empty[1]. As in previous examples, places can be understood as objects and the state of the game at each moment is given by a configuration of objects. The constants `initial` and `final` represent the initial and final configurations.

There are two possible legal moves in the game, but taking advantage of the circularity of the board, it is possible to represent both together in one rule, as shown below; notice how the condition of the rule considers the two possible directions of the move.

```
mod CHIPS is
  protecting NAT .
  sorts Place Board Chip .
  subsort Place < Board .

  ops r b g y e : -> Chip .
  op place : Nat Chip -> Place .
  op __ : Board Board -> Board [assoc comm] .
  ops initial final : -> Board .

  eq initial = place(0,r) place(1,b) place(2,g) place(3,y)
               place(4,e) place(5,e) place(6,e) place(7,e)
               place(8,e) place(9,e) place(10,e) place(11,e) .

  eq final = place(0,y) place(1,g) place(2,b) place(3,r)
             place(4,e) place(5,e) place(6,e) place(7,e)
             place(8,e) place(9,e) place(10,e) place(11,e) .

  vars I J : Nat .
  var C : Chip .

  crl [move] : place(I,C) place(J,e) => place(I,e) place(J,C)
            if ((I + 5) rem 12 == J) or ((J + 5) rem 12 == I).
endm
```

Then, we can use the command

```
Maude> search initial =>* B:Board such that B:Board == final .

No solution.
```

to prove that it is not possible, by using the allowed moves, to reverse the original order of the chips. Notice that the constant `final` is not a pattern (because it can be reduced), and therefore cannot be used after the arrow in the `search` command; the clause at the end allows us to say the same.

[1] In this example, we could also use a list representation for the state; this would simplify the representation of places, but instead the corresponding rules would be more complex.

8 The Khun Phan Puzzle

The Khun Phan puzzle is one of those typical puzzles consisting of a rectangular board over which some pieces can be slid. The goal is to move the pieces so as to reach a certain configuration in which sometimes a picture becomes clear or other times a piece understood as some character is freed from his guards. Figure 1 shows the initial configuration that we will consider. The board is a 4×5 rectangle, there is one 2×2 piece, five rectangular pieces of size 2×1, and four smaller squares with dimension 1×1; there are only two empty spaces that must be used to slide the pieces. The goal we consider is to move the pieces so as to put the big square in the position where the small ones are initially. An additional twist would be to reach a completely symmetric position with respect to the original.

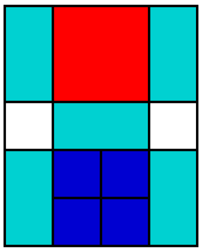

Figure 1. Khun Phan puzzle

The state of the board is also represented as a *multiset* of pieces with the operator `__`. There is a different constructor for each piece, `bigsq`, `hrect`, `vrect`, and `smallsq`, and another one, `empty`, to indicate an empty space (that is considered to be just a special kind of piece). These constructors take two natural numbers as arguments that correspond to the coordinates of the upper left corner of the piece; the origin $(1, 1)$ is located at the upper left corner of the board.

The representation of the moves as rewrite rules is then immediate: each involves a piece and at least one empty space. For each kind of piece there are four rules, corresponding to the four possible directions. For example, moving the big square one position to the right is captured by the rule `Sqr` below. Again we think of pieces as objects and rules as interactions among them. The complete specification is as follows.

```
mod KHUN-PHAN is
  protecting NAT .
  sorts Piece Board .
  subsort Piece < Board .

  op __ : Board Board -> Board [assoc comm] .
  ops empty bigsq smallsq hrect vrect : Nat Nat -> Piece .
  op init : -> Board .

  vars X Y : Nat .

  eq initial = vrect(1,1)       bigsq(2,1)          vrect(4,1)
               empty(1,3)       hrect(2,3)          empty(4,3)
               vrect(1,4) smallsq(2,4) smallsq(3,4) vrect(4,4)
                          smallsq(2,5) smallsq(3,5) .
```

```
  rl [sqr] : smallsq(X,Y) empty(s(X),Y) => empty(X,Y) smallsq(s(X),Y) .
  rl [sql] : smallsq(s(X),Y) empty(X,Y) => empty(s(X),Y) smallsq(X,Y) .
  rl [squ] : smallsq(X,s(Y)) empty(X,Y) => empty(X,s(Y)) smallsq(X,Y) .
  rl [sqd] : smallsq(X,Y) empty(X,s(Y)) => empty(X,Y) smallsq(X,s(Y)) .

  rl [Sqr] : bigsq(X,Y) empty(s(s(X)),Y) empty(s(s(X)),s(Y)) =>
             empty(X,Y) empty(X,s(Y)) bigsq(s(X),Y) .
  rl [Sql] : bigsq(s(X),Y) empty(X,Y) empty(X,s(Y)) =>
             empty(s(s(X)),Y) empty(s(s(X)),s(Y)) bigsq(X,Y) .
  rl [Squ] : bigsq(X,s(Y)) empty(X,Y) empty(s(X),Y) =>
             empty(X,s(s(Y))) empty(s(X),s(s(Y))) bigsq(X,Y) .
  rl [Sqd] : bigsq(X,Y) empty(X,s(s(Y))) empty(s(X),s(s(Y))) =>
             empty(X,Y) empty(s(X),Y) bigsq(X,s(Y)) .

  rl [hrectr] : hrect(X,Y) empty(s(s(X)),Y) => empty(X,Y) hrect(s(X),Y) .
  rl [hrectl] : hrect(s(X),Y) empty(X,Y) => empty(s(s(X)),Y) hrect(X,Y) .
  rl [hrectu] : hrect(X,s(Y)) empty(X,Y) empty(s(X),Y) =>
                empty(X,s(Y)) empty(s(X),s(Y)) hrect(X,Y) .
  rl [hrectd] : hrect(X,Y) empty(X,s(Y)) empty(s(X),s(Y)) =>
                empty(X,Y) empty(s(X),Y) hrect(X,s(Y)) .

  rl [vrectr] : vrect(X,Y) empty(s(X),Y) empty(s(X),s(Y)) =>
                empty(X,Y) empty(X,s(Y)) vrect(s(X),Y) .
  rl [vrectl] : vrect(s(X),Y) empty(X,Y) empty(X,s(Y)) =>
                empty(s(X),Y) empty(s(X),s(Y)) vrect(X,Y) .
  rl [vrectu] : vrect(X,s(Y)) empty(X,Y) => empty(X,s(s(Y))) vrect(X,Y) .
  rl [vrectd] : vrect(X,Y) empty(X,s(s(Y))) => empty(X,Y) vrect(X,s(Y)) .
endm
```

Then we can use the command

```
Maude> search initial =>* B:Board bigsq(2,4) .
```

to get all possible 964 solutions to the game. The final state used, `B:Board bigsq(2,4)`, represents any final situation such that the upper left corner of the big square is at coordinates $(2,4)$. No wonder it takes some time to find a solution: close examination of the first one, corresponding to the shortest path leading to the final configuration due to the breadth-first search, reveals that it consists of 112 moves!

Similarly, the command

```
Maude> search initial =>* vrect(1,1) smallsq(2,1) smallsq(3,1) vrect(4,1)
                                     smallsq(2,2) smallsq(3,2)
                          empty(1,3)        hrect(2,3)        empty(4,3)
                          vrect(1,4)        bigsq(2,4)        vrect(4,4) .

No solution.
```

shows that it is not possible to reach a position symmetric to the initial one.

9 Crossing the River

A shepherd needs to transport to the other side of a river a wolf, a goat, and a cabbage. He has only a boat with room for the shepherd himself and another item. The problem is that in the absence of the shepherd the wolf would eat the goat, and the goat would eat the cabbage.

We represent with constants `left` and `right` the two sides of the river. The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located; the constant `initial` denotes the initial situation in which we assume that all the objects are located in the left riverbank. The rules represent the ways of crossing the river that are allowed by the capacity of the boat; an auxiliary operation `change` is used to modify the corresponding attributes.

The interesting decision we have made in our specification is to use equations to represent the facts that the wolf eats the goat when they are alone, or that the goat eats the cabbage. Note that the statement of the problem is underspecified; it is not clear what exactly should happen were the wolf, the goat, and the cabbage left alone. In the specification below we have decided that the goat is not fast enough and gets eaten by the wolf before it can take a bite of the cabbage. Note that we use conditional equations, introduced with keyword `ceq`.

```
mod RIVER-CROSSING is
  sorts Side Group .

  ops left right : -> Side .
  op change : Side -> Side .
  ops s w g c : Side -> Group .
  op __ : Group Group -> Group [assoc comm] .
  op initial : -> Group .

  vars S S' : Side .

  eq change(left) = right .
  eq change(right) = left .

  ceq w(S) g(S) s(S') = w(S) s(S') if S =/= S' .
  ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S') if S =/= S' .

  eq initial = s(left) w(left) g(left) c(left) .

  rl [shepherd-alone] : s(S) => s(change(S)) .
  rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

By using the command `search` we can confirm that there is only one way the shepherd can safely take his belongings to the other side.

```
Maude> search initial =>* w(right) s(right) g(right) c(right) .
```

One might think about using rules instead of equations to represent the "eating transitions," but this would not be correct because it would allow paths in which the shepherd leaves for example the goat and the cabbage alone and later comes back to find that the cabbage is still there.

10 Dominoes on the Chessboard

We are given an 8×8 board and 31 dominoes, each of which can be used to cover exactly two squares of the board. Is it possible to arrange the dominoes on the board so as to leave uncovered the upper left and the lower right corners?

The answer is no, and a neat solution is given in [4, Chapter 1] among other places. It is enough to imagine the board painted like a chessboard and realize that each domino necessarily covers both a black and a white square: since the corners to be left uncovered are of the same color, such a covering is not possible. This solution, however, requires some ingenuity and, given our present lazy approach, that is not a desirable characteristic. Therefore, we are going to model the problem in Maude and try to solve it by sheer force.

Again, the state of the board is represented as a *multiset* of squares. Each square has three arguments: the first two are its coordinates (column/row) and the last one indicates whether it is already covered or still empty. Since the position of the squares in the board is fixed, the attribute `comm` for `__` could be thought to be unnecessary. This, however, allows a more homogeneous and simple presentation of the rules taking care of positioning the dominoes *both* horizontally and vertically, by focusing only on those two squares involved in placing the domino. Having the board represented as a list by removing the attribute `comm` would force us to represent all the squares in between them in one of the rules.

```
mod CHESS-COVER is
  protecting NAT .
  sorts Status Pos Board State .
  subsort Pos < Board .

  ops e c : -> Status .
  op sq : Nat Nat Status -> Pos .
  op __ : Board Board -> Board [assoc comm] .
  ops initial final : -> Board .

  vars I J I1 J1 : Nat .
  var B : Board .

  eq initial = sq(1,1,e) sq(2,1,e) sq(3,1,e) sq(4,1,e) sq(5,1,e) ...
             ... sq(4,8,e) sq(5,8,e) sq(6,8,e) sq(7,8,e) sq(8,8,e) .

  eq final = sq(1,1,e) sq(2,1,c) sq(3,1,c) sq(4,1,c) sq(5,1,c) ...
           ... sq(4,8,c) sq(5,8,c) sq(6,8,c) sq(7,8,c) sq(8,8,e) .

  rl [hor] : sq(I,J,e) sq(s(I),J,e) => sq(I,J,c) sq(s(I),J,c) .
  rl [ver] : sq(I,J,e) sq(I,s(J),e) => sq(I,J,c) sq(I,s(J),c) .
endm
```

Now, the command

```
Maude> search initial =>* B:Board such that B:Board == final .
```

should return the answer. This time, however, a state explosion problem occurs and in our computer the program runs out of memory before producing any result. To solve it, we are forced to use some ingenuity after all. Note that instead of placing the dominoes in an arbitrary order we could do it starting either from the top of the board towards the bottom, or from the left towards the right, or even in a diagonal manner beginning at the upper left corner. The first two approaches still do not return an answer, but the third does. To implement it, we need an auxiliary operator `cDiag` that checks whether all positions in the board that come before a given square according to the diagonal order have been already covered. Also, as we did in Section 4, we need to have full control of all the elements in the board and for that we enclose it inside the constructor `{_}`.

```
ceq cDiag(I,J,sq(I1,J1,e) B) = false if (I1 + J1 < I + J) /\ (I1 + J1 >= 3) .
eq cDiag(I,J,B) = true [owise] .

crl [hor] : { B sq(I,J,e) sq(s(I),J,e) } => { B sq(I,J,c) sq(s(I),J,c) }
            if cDiag(I,J,B) .
crl [ver] : { B sq(I,J,e) sq(I,s(J),e) } => { B sq(I,J,c) sq(I,s(J),c) }
            if cDiag(I,J,B) .
```

The "otherwise" attribute, `owise`, is just a convenient way of specifying the behavior of `cDiag` in all remaining cases without having to write equations for them. The result is still an equational theory since the `owise` attribute is just a shorthand for a conditional equation [3, Section 4.5.4].

Finally, the result of the command

```
Maude> search { initial } =>* { B:Board } such that B:Board == final .

No solution.
```

proves that such a covering is not possible.

11 By Way of Conclusion

We have specified several other games and puzzles, but we think that by now the pattern by which these problems are modeled and solved in Maude should be clear. There are however several advanced features available in Maude that can be useful in some examples, and that we have not considered here with the idea of keeping an introductory level.

The first one is the possibility of using membership axioms [3, Chapter 4] to refine the representation of the state. For example, the multiset constructor allows repetition of elements, but this should be forbidden in some situations; as another example, in the Khun Phan puzzle a piece cannot be stacked on top of another. In the puzzles above we have not made use of this, but memberships are the right tool to make sure all the elements are different.

Another important feature is the availability of a model checker for linear temporal logic [3, Chapter 9]. For example, we have modeled the river crossing puzzle without the "eating equations" by using the model checker instead of the `search` command to represent a safe path by means of a temporal formula.

To gain some perspective we have also carried out a small comparison with other rule-based programming languages, namely, ELAN [2], CHR [5], and ASF+SDF [13]. We acknowledge beforehand that none of us is an expert in any of them, so our conclusions should be taken with a grain of salt. ELAN, which is also based on rewriting logic, is the most similar to Maude, offering a clear distinction between the statics and the dynamics of a system, by means of two different kinds of rules. Like Maude, ELAN also supports rewriting modulo associativity and commutativity, though not modulo associativity only, so the examples involving lists are more cumbersome to describe. The examples involving multiset rewriting can be easily specified; however, since in most cases the rules are nonterminating and the predefined search strategy in ELAN is depth-first, it cannot be used to find a solution (a breadth-first strategy could be specified but by no means in a straightforward manner). Regarding efficiency, for the only problem we were able to translate almost verbatim from Maude, the Josephus problem in Section 4, the performance of the Maude interpreter was much better than even the compiled ELAN version (in some of our experiments, Maude finished within seconds while ELAN took several hours). On the other hand, after trying to specify some of the games in CHR and ASF+SDF, as far as we know there is no distinction between equations and rules and no support for equational attributes, both of which have been essential in our examples. All this suggests to us that the representation of transition systems in these two formalisms is not as natural and straightforward as in Maude. In addition, breadth-first search in CHR seems to present the same problems as in ELAN.

As we mentioned in the introduction, the main goal of this paper was to have some fun while introducing rule-based programming in Maude; we hope that our games have whetted the appetite of the reader for more Maude applications.

References

1. A. Bogomolny. Interactive mathematics miscellany and puzzles. `http://www.cut-the-knot.org/games.shtml`.
2. P. Borovanský, C. Kirchner, H. Kirchner and P.-E. Moreau. ELAN from the rewriting logic point of view *Theoretical Computer Science*, 285(2):155–185, 2002.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.1). `http://maude.cs.uiuc.edu/manual/`, 2004.
4. D. Fomin, S. Genkin, and I. Itenberg. *Mathematical Circles: The Russian Experience*, volume 7 of *Mathematical World*. American Mathematical Association, 1996.
5. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
6. P. J. Hayes. What the frame problem is and isn't. In Z. W. Pylyshyn, editor, *The Robot's Dilemma: The Frame Problem in Artificial Intelligence*, pages 123–137. Ablex Publishing Corp., 1987.
7. N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, pages 1–53. Kluwer Academic Press, 1999.
8. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.
9. N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. W. W. Rouse Ball and H. S. M. Coxeter. *Mathematical Recreations and Essays*. Dover, 1987.
12. D. O. Shklarsky, N. N. Chentzov, and I. M. Yaglom. *The USSR Olympiad Problem Book*. Dover, 1993.
13. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing, 5, World Scientific, 1996.

On-demand evaluation for Maude*

Francisco Durán[1], Santiago Escobar[2] and Salvador Lucas[2]

[1] LCC, Universidad de Málaga, Campus de Teatinos, Málaga, Spain. duran@lcc.uma.es
[2] DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain. {sescobar,slucas}@dsic.upv.es

Abstract Strategy annotations provide a simple mechanism for introducing some laziness in the evaluation of expressions. As an eager programming language, Maude can take advantage of them and, in fact, they are part of the language. Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the ordering in which the arguments are (eventually) evaluated in function calls. A positive index enables the evaluation of an argument whereas 'zero' means that the function call has to be attempted. The use of negative indices has been proposed to express *evaluation on-demand*, where the *demand* is an attempt to match an argument term with the left-hand side of a rewrite rule. In this paper we show how to furnish Maude with the ability of dealing with on-demand strategy annotations.

1 Introduction

Handling infinite objects is a typical feature of lazy (functional) languages. Although reductions in Maude [4,5] are basically *innermost* (or eager), Maude is able to exhibit a similar behavior by using *strategy annotations* [17]. Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the ordering in which the arguments are (eventually) evaluated in function calls: when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy $(i_1 \cdots i_n)$ for f are evaluated (following the specified ordering). If 0 is found, a reduction step on the whole term $f(t_1, \ldots, t_k)$ is attempted. In fact, Maude gives a strategy annotation $(1\ 2\ \cdots\ k\ 0)$ to each symbol f without an explicit strategy annotation.

Example 1. Consider the following modules `LAZY-NAT` and `LIST-NAT` defining sorts `Nat` and `LNat`, and symbols `0` and `s` for defining natural numbers, and symbols `nil` (the empty list) and `_._` for the construction of lists.

```
fmod LAZY-NAT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat [strat (0)] .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm

fmod LIST-NAT is
  pr LAZY-NAT .
  sorts LNat .
  subsort Nat < LNat .
  op _._ : Nat LNat -> LNat [strat (1 0)] .
  op nil : -> LNat .
  op nats : -> LNat .
  op incr : LNat -> LNat .
  op length : LNat -> Nat .
```

* Work partially supported by CICYT TIC2001-2705-C03-01 and TIC2001-2705-C03-02, MCyT Acción Integrada HU 2003-0003 and Agencia Valenciana de Ciencia y Tecnología GR03/025.

```
  vars X Y : Nat .   vars XS YS : LNat .
  eq incr(X . XS) = s(X) . incr(XS) .
  eq nats = 0 . incr(nats) .
  eq length(nil) = 0 .
  eq length(X . XS) = s(length(XS)) .
endfm
```

Strategy annotations can often improve the termination behavior of programs (by pruning all infinite rewrite sequences starting from any expression). In the example above, the strategies `(0)` and `(1 0)` for symbols `s` and `_._`, respectively, guarantee that the resulting program is terminating[1] (note that both strategies are necessary for such a proof of termination). Strategy annotations can also improve the efficiency of computations (e.g., by reducing the number of attempted matchings or avoiding useless or duplicated reductions) [10].

Nevertheless, the absence of some indices in the local strategies can also jeopardize the ability of such strategies to compute normal forms. For instance, the evaluation of the expression `s(0) + s(0)` w.r.t. Example 1 using Maude[2] yields the following:

```
Maude> (red s(0) + s(0) .)
result Nat: s(0 + s(0))
```

Due to the annotation (0) for the symbol `s`, the contraction of the redex `0 + s(0)` is not possible and the evaluation stops here.

The handicaps, regarding correctness and completeness of computations, of using (only) positive annotations are discussed in, e.g., [1,2,15,18,19], and a number of solutions have been proposed:

1. Performing a *layered normalization*: when the evaluation stops due to the replacement restrictions introduced by the strategy annotations, it is resumed over concrete inner parts of the resulting expression until the normal form is reached (if any) [16];
2. transform the program to obtain a different one which is able to obtain sufficiently interesting outputs (e.g., constructor terms) [2]; and
3. use strategy annotations with *negative* indices which allows for some extra evaluation *on-demand*, where the *demand* is an attempt to match an argument term with the left-hand side of a rewrite rule [18,19,1].

In [7], we have introduced two new commands (`norm` and `eval`) to make techniques 1 and 2 available for the execution of Maude programs. In this paper we show how we have brought on-demand strategies into Maude. Before entering into details, we show how negative indices can improve Maude strategy annotations.

Example 2. (Continuing Example 1) The following `NATS-TO-BIN` module implements the binary encoding of natural numbers as lists of 0 and 1 (starting from the least significative bit).

```
fmod NATS-TO-BIN is
  ex LAZY-NAT .
  pr LIST-NAT .
  op 1 : -> Nat .
```

[1] The termination of the specification can be formally proved by using the tool MU-TERM, see `http://www.dsic.upv.es/~slucas/csr/termination/muterm`.

[2] The Maude 2.1 interpreter [5] is available at `http://maude.cs.uiuc.edu`.

```
  op natToBin : Nat -> LNat .
  op natToBin2 : Nat Nat -> LNat .
  vars M N X : Nat .   vars XS YS : LNat .
  eq natToBin2(0, 0) = 0 .
  eq natToBin2(0, M) = 0 . natToBin(M) .
  eq natToBin2(s(0), 0) = 1 .
  eq natToBin2(s(0), M) = 1 . natToBin(M) .
  eq natToBin2(s(s(N)), M) = natToBin2(N, s(M)) .
  eq natToBin(N) = natToBin2(N, 0) .
endfm
```

The evaluation of the expression `natToBin(s(0) + s(0))` should yield the binary representation of 2. However, we get:

```
Maude> (red natToBin(s(0) + s(0)) .)
result LNat: natToBin2(s(0 + s(0)), 0)
```

The problem is that the current strategy annotations disallow the evaluation of subexpression `0 + s(0)` in `natToBin2(s(0 + s(0)), 0)`, thus disabling the application of the last equation for `natToBin2`. The use of the command `norm` introduced in [7] does not solve this problem, since it just normalizes non-reduced subexpressions:

```
Maude> (norm natToBin(s(0) + s(0)) .)
result LNat: natToBin2(s(s(0)), 0)
```

As we show below, on-demand strategy annotations can solve this problem. In fact, the use of the strategy `(-1 0)` for symbol `s`, declaring its first argument as evaluable only on-demand, permits to recover the desired behavior while keeping termination of the program (see Examples 3 and 4 below).

In this paper, we furnish Maude with the ability of dealing with on-demand strategy annotations. The reflective capabilities of Maude are the key for building such language extensions, which turn out to be very simple to use thanks to the infrastructure provided by Full Maude. Full Maude is an extension of Maude written in Maude itself, that endows Maude with notation for object-oriented modules and with a powerful and extensible module algebra [4]. Its design, and the level of abstraction at which it is given, make of it an excellent metalevel tool to test and experiment with features and capabilities not present in (Core) Maude [8,9,4]. We make use of the extensibility and flexibility of Full Maude to permit the use of both `red` (the usual evaluation command of Maude) and `norm` (introduced in [7]) with Maude programs using on-demand strategy annotations.

2 On-demand evaluation strategy

As explained in the introduction, the absence of some indices in the local strategies of Maude programs can jeopardize the ability of such strategies to compute normal forms. In [18,19,1], *negative* indices are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [19]. For instance, the evaluation of the subterm `0 + s(0)` of the term `natToBin2(s(0 + s(0)), 0)` in Example 2 is *demanded* by the last equation for symbol `natToBin2`, i.e., by its left-hand side `natToBin2(s(s(N)), M)`: the argument of the outermost occurrence of the symbol `s` in `natToBin2(s(0 + s(0)), 0)` is rooted by a defined function symbol, `_+_`, whereas the corresponding operator in the left-hand side is `s`. Thus, before being able to apply the rule, we have to further evaluate `0 + s(0)`.

As for our running example, we may conclude that the evaluation with (only) positive annotations either enters in an infinite derivation —e.g., for the term `length(nats)`, with the strategy `(1 0)` for symbol `s`— or does not provide the intended normal form —e.g., with the strategy `(0)` for symbol `s`, see Example 2—. The strategy `(-1 0)`, however, gives an appropriate local strategy for symbol `s`, since it makes its argument to be evaluated only "on demand". Then, the evaluation of the expression `natToBin(s(0) + s(0))` under the strategy `(-1 0)` for `s` is able to reduce the symbol `natToBin2`, and to remove it from the top position, thus obtaining a head-normal form (see Example 3 below). This also permits to use the resulting expression as the starting point of a layered evaluation (with `norm`) leading to the normal form (see Example 4 below). Note that this is achieved without entering in a non-terminating evaluation. We refer the reader to [11] for a recent and detailed discussion about the use of on-demand strategy annotations in programming.

In this paper, we follow the computational model defined in [1] for dealing with negative annotations. A local strategy for a k-ary symbol $f \in \mathcal{F}$ is a sequence $\varphi(f)$ of integers in $\{-k, \ldots, -1, 0, 1, \ldots, k\}$, which are given inside parentheses. A mapping φ that associates a local strategy $\varphi(f)$ to every $f \in \mathcal{F}$ is called an E-strategy map [18]. In order to evaluate an expression e, each symbol in e is conveniently annotated according to the E-strategy map. The evaluation of the annotated expression takes a term and the strategy associated to its top symbol, and then proceeds by considering the annotations of such a strategy sequentially [1]: if a positive argument index is provided, then the evaluation jumps to the subterm at such argument position; if a negative argument index is provided, then the index is consumed but nothing is done; if a zero is found, then we try to find a rule to be applied on such a term. If no rule can be applied, then we proceed to perform their (demanded) evaluation, that is, we try to reduce one of the subterms in positions with (consumed or present) negative indices. All consumed indices (positive and negative) are kept associated to each symbol in the term using an extra strategy list, so that demanded positions can be searched. See [1] for a formal description of the procedure and for details about why the memory list is necessary compared to other frameworks for negative annotations as OBJ3 [14] and CafeOBJ [18,19].

In this paper, we do not consider AC symbols or rules with non-linear left-hand side. Strategy annotations are explicitly prohibited for AC symbols (see [12,13]) and the completeness of evaluation with strategy annotations is only guaranteed for linear left-hand sides (see [16,1]).

3 Reflection and the `META-LEVEL` module

Maude's design and implementation systematically exploits the reflective capabilities of rewriting logic [4], providing key features of the universal theory in its built-in `META-LEVEL` module. In particular, `META-LEVEL` has sorts `Term` and `Module`, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\overline{t}$ of sort `Term` and a term $\overline{\mathcal{R}}$ of sort `Module`.

The basic cases in the representation of terms are obtained by subsorts `Constant` and `Variable` of the sort `Qid` of quoted identifiers. Constants are quoted identifiers that contain the name of the constant and its type separated by a dot, e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a colon, e.g., `'N:Nat`. Then, a term is constructed in the usual way, by applying an operator symbol to a list of terms.

```
subsorts Constant Variable < Qid Term .
```

```
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc] .
op _[_] : Qid TermList -> Term [ctor] .
```

For example, the term `natToBin2(s(s(0)), 0)` of sort `LNat` in the module `NATS-TO-BIN` is metarepresented as

```
'natToBin2['s['s['0.Nat]], '0.Nat]
```

The `META-LEVEL` module also includes declarations for metarepresenting modules. For example, a functional module can be represented as a term of sort `Module` using the following operator.

```
op fmod_is_sorts_.____endfm : Qid ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet -> FModule [ctor] .
```

Similar declarations allow us to represent the different types of declarations we can find in a module.

The module `META-LEVEL` also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, `metaApply`, `metaRewrite`, `metaReduce`, etc., and also generic parsing and pretty printing functions `metaParse` and `metaPrettyPrint` [6,4]. For example, the function `metaReduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module:

```
op metaReduce : Module Term -> [ResultPair] .
op {_,_} : Term Type -> ResultPair [ctor] .
```

`metaReduce` returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind.

All these functionalities are very useful for metaprogramming, and in particular when building formal tools. Moreover, Full Maude provides a powerful setting in which additional facilities are available, making the addition of new commands or the redefinition of previous ones, as in this paper, simpler. The specification of Full Maude and its execution environment can then be used as the infrastructure on which building new features.

4 Extending Full Maude to handle on-demand strategy annotations

We provide the reduction of terms taking into account on-demand annotations as a redefinition of the usual evaluation command `red` of Maude (which considers only positive annotations).

Example 3. Consider the specification resulting from replacing in Example 2 the declaration of the operator `s` by this other one:

```
op s : Nat -> Nat [strat (-1 0)] .
```

The on-demand evaluation of `natToBin(s(0) + s(0))` obtains a head-normal form:

```
Maude> (red natToBin(s(0) + s(0)) .)
result LNat : 0 . natToBin(s(0))
```

As for other commands in Full Maude, we may define the actions to take when the new commands are used by defining its corresponding meta-function. For instance, a `red` command is executed by appropriately calling the metalevel `metaReduce` function. In order to furnish Maude with on-demand evaluation we provide a new metalevel operation metaReduceOnDemand which extends the reflective and metalevel capabilities of Maude, as explained in Section 3. The operation metaReduceOnDemand takes arguments of sorts Module, OpDeclSet and Term, and returns a term of sort ResultPair. Its arguments represent, respectively, the module on which the reduction takes place, the operation declarations in such a module, and the term to be reduced. The result returned is as the one given by metaReduce (see Section 3). Note that (Core) Maude cannot handle negative annotations, and therefore, the function takes a valid module, i.e. a module without negative annotations, and the set of operation declarations with any kind of annotation. The redefined command red must then select between metaReduce and metaReduceOnDemand depending on whether negative annotations are present or not.

Basically, metaReduceOnDemand calls the auxiliary function procStrat which is the function that really processes the strategy list associated to the top symbol of the term.

```
var M : Module .   var OPDS : OpDeclSet .  var T T' : Term .

op metaReduceOnDemand : Module OpDeclSet Term -> [ResultPair] .
op procStrat : Module OpDeclSet AnnTerm -> AnnTerm .

ceq metaReduceOnDemand(M, OPDS, T)
  = {T', leastSort(M, T')}
  if T' := erase(procStrat(M, OPDS, annotate(M, OPDS, T))) .
```

In order to include annotations into Maude's representation of terms, we transform the Maude's metalevel sort Term into a sort AnnTerm (of annotated terms), where symbols are equipped with a memory list and a strategy list (as explained in Section 2). Furthermore, we provide two functions: annotate and erase to move between the sorts Term and AnnTerm.

```
sorts AnnVariable AnnTerm AnnTermList .
subsorts AnnVariable < AnnTerm < AnnTermList .
op _{} : Variable -> AnnVariable .
op _{_} : Constant IntListNil -> AnnTerm .
op _{_|_}[_] : Qid IntListNil IntListNil AnnTermList -> AnnTerm .
op _,_ : AnnTermList AnnTermList -> AnnTermList [assoc] .

op annotate : Module OpDeclSet TermList -> AnnTermList .
op erase : AnnTermList -> TermList .
```

The function procStrat follows the description given in Section 2 when processing the strategy list associated to the top symbol of the term to evaluate. When a positive index is found, the evaluation of such argument is forced, and the positive index is moved from the strategy list (right component) to the memory list (left component) of the top symbol. For example, the equation for an annotated term rooted by a symbol with arity greater than 0 is as follows.

```
var N N' : Int .   var NL NL' : IntListNil .
var F : Qid .      var ATL : AnnTermList .

ceq procStrat(M, OPDS, F{NL | N NL'}[ATL])
  = procStrat(M, OPDS,
      F{NL @@ N | NL'}[procStratSel(M, OPDS, ATL, 1, N)])
  if N > 0 .
```

When a negative index is found, no evaluation in that argument is started, and the negative index is moved from the strategy list (right component) to the memory list (left component).

```
ceq procStrat(M, OPDS, F{NL | N NL'}[ATL])
  = procStrat(M, OPDS, F{NL @@ N | NL'}[ATL] )
  if N < 0 .
```

When an index 0 is found, the function procStrat attempts to match the term against the left-hand sides of the rules using the metalevel function metaApply.[3] If there is a match, then the rule is applied. If no match is obtained, then we determine if any demanded position exists using the function procStratOD, which performs a matching algorithm to detect which positions under negative annotations are actually demanded by some rule (see [1] for details). If a demanded position exists, then the evaluation of such a position is started, and then we will retry the matching against the left-hand sides of the rules after the evaluation is completed. If no demanded position exists, the current index 0 is removed from the strategy list and the rest of the strategy list is considered.

```
var MA : ResultTriple? .

ceq procStrat(M, OPDS, F{NL | 0 NL'}[ATL])
  = if MA == failure
    then procStratOD(M, OPDS, F{NL | 0 NL'}[ATL])
    else procStrat(M, OPDS, annotate(M, OPDS, getTerm(MA)))
    fi
  if MA :=
     metaApply(moveEqsToRls(M), F[erase(ATL)], 'on-demand, none, 0).
```

When the function procStratOD is executed, i.e. when a demanded position is being searched, the computational model of [1] specifies that the search order defined by the position order in the strategy must be followed, i.e. if (-1 -2 0) is the strategy for symbol _._, then any demanded subterm under the first argument would be selected first, despite any demanded subterm under the second argument (see [1] for details).

Once implemented the function metaReduceOnDemand, we need to redefine parts of Full Maude so that the command red can be able to execute metaReduce or metaReduceOnDemand. There is no need to define a new command and extend Full Maude to accept that command, as it was done for norm and eval commands in [7]. We just need to modify the way the red command is processed.

In the current version of Maude, input/output is accomplished by the predefined LOOP-MODE module, which provides a generic read-eval-print loop. In the case of Full Maude, the persistent state of the loop is given by a single object of class Database which maintains the database of the system. This object has an attribute db, to keep the actual database in which all the modules being entered are stored (a set of records), an attribute default, to keep the identifier of the current module by default, and attributes input and output to simplify the communication of the read-eval-print loop given by the LOOP-MODE module with the database. Using the notation for classes in object-oriented modules we can declare such a class as follows:

```
  class DatabaseClass | db : Database,    default : ModName,
                        input : TermList, output : QidList .
```

The state of the read-eval-print loop is then given by an object of class DatabaseClass. In the case of Full Maude, the handling of the read-eval-print loop is defined in the modules DATABASE-HANDLING and FULL-MAUDE.

[3] The function metaApply applies only rules, and therefore equations must be turned into rules before metaApply is applied.

The module FULL-MAUDE includes the rules to initialize the loop (rule init), and to specify the communication between the loop—the input/output of the system—and the database (rules in and out). Depending on the kind of input that the database receives, its state will be changed, or some output will be generated. To parse some input using the built-in function metaParse, Full Maude needs the metarepresentation of the signature in which the input is going to be parsed. In Full Maude, such a grammar is provided by the FULL-MAUDE-SIGN module, in which we can find the appropriate declarations so that any valid input, namely modules, theories, views, and commands, can be parsed. Since we do not want to change the grammar FULL-MAUDE-SIGN, which is used for parsing the inputs, we do not need to change the FULL-MAUDE module.

The module DATABASE-HANDLING defines the behavior of the database upon new entries. The behavior associated to commands is managed by rules describing transitions which call the function procCommand. For example, the rule defining what to do when the red command is received is as follows.

```
rl [red] :
   < O : X@Database | db : DB, input : ('red_.[T]),
      output : nil, default : MN, Atts >
   => < O : X@Database | db : DB, input : nilTermList,
         output : procCommand('red_.[T], MN, DB),
         default : MN, Atts > .
```

When a red command is entered, the parsing of the input returns a term of the form red_.[T], where T is a variable of sort T representing a bubble. The result of the parsing is placed in the input attribute of the database object. The function procCommand specifies what to do when the term red_.[T] is received, with MN and DB variables with values the name of the current default module and the state of the database, respectively. In the original case of the command red, procCommand calls the function procRed with the appropriate arguments, namely the name of the default module, the flatten module itself, the bubble representing the argument of the command, the variables in the default module, and the database. Note that depending on whether the default module is a built-in or not, and whether it is compiled or not, procCommand will do different things, so that the arguments for procRed are obtained. In the redefinition for command red, procCommand calls a new function procReduceOnDemand which redefines procRed.

```
eq procCommand('red_.['bubble[T]], MN, DB)
  = if MN inModNameSet builtIns
    then procReduceOnDemand(MN, DUMMY(MN), 'bubble[T], none, DB)
    else if compiledUnit(MN, DB)
         then procReduceOnDemand(MN, getFlatUnit(MN, DB),
                'bubble[T], getVbles(MN, DB), DB)
         else procReduceOnDemand(MN,
                getFlatUnit(MN, evalModExp(MN, DB)),
                'bubble[T], getVbles(MN, evalModExp(MN, DB)),
                evalModExp(MN, DB))
         fi
    fi .
```

The function procReduceOnDemand is in charge of evaluating the bubble given as argument of the red command, calling the function metaReduce or metaReduceOnDemand, and then preparing the results (a list of quoted identifiers that will be passed to the output channel of the read-eval-print loop to be shown to the user). The function procReduceOnDemand detects whether negative annotations are present in the module or not[4] (using the function `noNegAnns`), then calling `metaReduceOnDemand` or `metaReduce`. As said above, since Core Maude does not accept strategies with negative annotations, the function `procReduceOnDemand` must call the function `metaReduceOnDemand` with the module without such negative annotations (`remNegAnns` is in charge of removing them) and the operator declarations with them. Finally, the equations defining `procReduceOnDemand` are as follows.

```
op procReduceOnDemand : ModExp Module Term OpDeclSet Database
   -> QidList .
ceq procReduceOnDemand(MN, M, T, VDS, DB)
    *** No negative annotation -> Use metalevel metaReduce
  = if RP? :: ResultPair
    then ('\b 'reduce 'in
         ...
    else ('\r 'Error: '\o 'Incorrect 'command. '\n)
    fi
  if noNegAnns(getOps(M))
         ...
     /\ TM := solveBubblesRed(T, remNegAnns(M), B, VDS, DB)
     /\ RP? := metaReduce(getModule(TM), getTerm(TM)) .

ceq procReduceOnDemand(MN, M, T, VDS, DB)
    *** Negative annotations -> Use metalevel metaReduceOnDemand
  = if RP? :: ResultPair
    then ('\b 'reduce 'on-demand 'in
         ...
    else ('\r 'Error: '\o 'Incorrect 'command. '\n)
    fi
  if not noNegAnns(getOps(M))
         ...
     /\ TM := solveBubblesRed(T, remNegAnns(M), B, VDS, DB)
     /\ RP? :=
        metaReduceOnDemand(getModule(TM), getOps(M), getTerm(TM)) .
```

4.1 Extending Full Maude with on-demand strategy annotations to layered normalization

As explained along the paper, our goal is to provide appropriate normal forms to programs with strategy annotations. However, the redefinition of command `red` is not able to provide the normal form `0 . 1` for the program in Example 2, since the annotation 2 is missing in the strategy list for symbol `_._` (see the output of the `red` command in Example 3). However, as it was explained in Section 1, this concrete problem is solved using either a layered normalization, or a transformation. In this section, we redefine the command `norm` of [7] to perform a layered normalization of the output given by the on-demand evaluation previously presented.

Example 4. Consider the modules of Example 3. The redefinition of command `norm` now is able to provide the intended value associated to the expression `natToBin(s(0) + s(0))`.

```
Maude> (norm natToBin(s(0) + s(0)) .)
result LNat : 0 . 1
```

The redefinition of command `norm` is almost identical to the implementation of the command `norm` given in [7]. We do not give the details here, but basically, the idea is that we keep

[4] The "classical" Maude evaluation is not affected when only positive annotations are provided.

the metalevel function metaNorm and define a new metalevel function metaNormOnDemand which calls metaReduceOnDemand instead of metaReduce to reduce the initial term.

```
eq metaNormODRed(M, OPDS, T)
  = procStratOD(M, getTerm(metaReduceOnDemand(M, OPDS, T)), OPDS) .
```

We refer the reader to [7] for details about the implementation of the norm command. Note that it is also necessary to perform similar changes to those explained in Section 4:

- we redefine procCommand to call a new function procNormOnDemand, which redefines procNorm, when the term norm_.[T] is received;
- the function procNormOnDemand calls metaNorm or metaNormOnDemand depending on whether negative annotations are present or not (using again the function noNegAnns).

5 Conclusions

We have used Full Maude to furnish Maude with the ability to perform on-demand evaluations, a more sophisticated form of lazy behavior for languages such as Maude. We make use of the extensibility and flexibility of Full Maude to permit the use of both red (the usual evaluation command of Maude) and norm (introduced in [7]) with Maude programs using on-demand strategy annotations. The full specification is available at

http://www.dsic.upv.es/users/elp/toolsMaude

These features have been integrated into Full Maude, making them available inside its programming environment. The high level at which the specification/implementation of Full Maude is given makes this approach particularly attractive when compared to conventional implementations (see e.g. [3]). The flexibility and extensibility that Full Maude affords has made the extension quite simple and in a very short time.

It is worth noting however that our prototype of on-demand evaluation is not comparable in efficiency to other implementations of evaluation with negative annotations such as in CafeOBJ[5] or OnDemandOBJ[6]. The goal of this piece of work is not to provide a competitive implementation, but to provide on-demand evaluation for a language such as Maude. Note that OnDemandOBJ does not include all the capabilities of Maude, and that the computational model of CafeOBJ for dealing with negative annotations has some drawbacks (see [1]). In fact, it is not fair looking at it as an implementation of the on-demand strategies, not even as a prototype. It should be seen as an executable specification of it, closer to its mathematical definition (given in [1]) than to its implementation. Although a more efficient executable specification/implementation of the on-demand evaluation following a similar approach could be given, we are convinced that a direct implementation of the on-demand evaluation into Maude is desirable.

References

1. M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving On-Demand Strategy Annotations. In M. Baaz and A. Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'02*, LNAI 2514:1-18, Springer, 2002.

[5] Available at http://www.ldl.jaist.ac.jp/cafeobj

[6] Available at http://www.dsic.upv.es/users/elp/ondemandOBJ

2. M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In F. Gadducci and U. Montanari, editors, *Proc. of the 4th International Workshop on Rewriting Logic and its Applications, WRLA'02*, ENTCS 71. Elsevier, 2004.
3. M. Alpuente, S. Escobar, and S. Lucas. OnDemandOBJ: A Laboratory for Strategy Annotations. In J.-L. Giavitto and P.-E. Moreau, editors, *Proc. of the 4th International Workshop on Rule-Based Programming, RULE'03*, ENTCS 86(2). Elsevier, 2003.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science* 285(2):187-243, 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, Proc. of *14th International Conference on Rewriting Techniques and Applications, RTA'03*, LNCS 2706:76-87, Springer, 2003.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 manual. Available in `http://maude.cs.uiuc.edu`, 2003.
7. F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In N. Martí-Oliet, editor, *Proc. of the 5th International Workshop on Rewriting Logic and its Applications, WRLA'04*, ENTCS to appear 2004.
8. F. Durán and J. Meseguer. An extensible module algebra for Maude. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15. Elsevier, 1998.
9. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, June 1999.
10. S. Eker. Term Rewriting with Operator Evaluation Strategies. *Electronic Notes in Theoretical Computer Science*, volume 15, 20 pages, 1998. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15. Elsevier, 1998.
11. S. Escobar. *Strategies and Analysis Techniques for Functional Program Optimization*. PhD Thesis, Universidad Politécnica de Valencia, October 2003.
12. M.C.F. Ferreira and A.L. Ribeiro. Context-Sensitive AC-Rewriting. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications, RTA'99*, LNCS 1631:173-181. Springer, 1999.
13. J. Giesl and A. Middeldorp. Transformation Techniques for Context-Sensitive Rewrite Systems. Journal of Functional Programming, to appear, 2004.
14. J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.
15. S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82-93, ACM Press, 2001.
16. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.
17. S. Lucas. Semantics of programs with strategy annotations. Technical Report DSIC-II/08/03, DSIC, Universidad Politécnica de Valencia, 2003.
18. M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *Proc. of 3rd International Workshop on Rewriting Logic and its Applications, WRLA'00*, ENTCS 36. Elsevier, 2001.
19. K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 International Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, 2000.

A Rewriting-based Framework for Web Sites Verification*

M. Alpuente[1], D. Ballis[2], and M. Falaschi[2]

[1] DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain.
alpuente@dsic.upv.es.

[2] Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy.
{demis,falaschi}@dimi.uniud.it.

Abstract In this paper, we develop a framework for the automated verification of Web sites which can be used to specify integrity conditions for a given Web site, and then automatically check whether these conditions are fulfilled. First, we provide a rewriting-based, formal specification language which allows us to define syntactic as well as semantic properties of the Web site. Then, we formalize a verification technique which obtains the requirements not fulfilled by the Web site, and helps to repair the errors by finding out incomplete information and/or missing pages. Our methodology is based on a novel rewriting-based technique, called *partial rewriting*, in which the traditional pattern matching mechanism is replaced by tree *simulation*, a suitable technique for recognizing patterns inside semistructured documents. The framework has been implemented in the prototype Web verification system VERDI which is publicly available.

1 Introduction

The increasing complexity of Web sites has turned their design and construction into a challenging problem. Systematic, formal approaches can bring many benefits to quality Web site construction, giving support for automated Web site verification. This paper presents an approach to Web site specification and verification based on rewriting-like machinery. We use rewriting-based technology both to specify the integrity conditions and to formalize a verification technique which obtains the requirements not fulfilled by the Web site, and then is able to repair errors by finding out missing pages and/or incomplete information, such as the data or the links available in a particular page.

Although the management of Web sites has received significant attention in recent years [6,11,12], few works address the semantic verification of Web sites. In [12], a declarative verification algorithm is proposed which checks a particular class of integrity constraints concerning the Web site's structure, but not the contents of a given instance of the site. [11] proposes a methodology which consists of using inference rules and axioms to define some semantic constraints concerning the Web site contents. Then, a verification technique is proposed which is based on compiling the specification into Prolog code. Our idea in this paper is that term rewriting techniques can support in a natural way not only intuitive, high level Web site specification, but also efficient Web site verification and repairing techniques. As far as we know, rewriting-based techniques have not been explored in this context to date. We only know of two related approaches which focus on transformation rather than verification issues: a rewriting-based implementation is provided in [15] for (a fragment of) XSLT, the rule-based language designed by W3C for the transformation of XML documents, whereas rewrite rules are used in [3] to perform HTML transformations with the aim of improving Web applications by cleaning up syntax, reorganizing frames, or updating to new standards.

Our contribution. We first provide a rewriting-based, formal specification language which allows us to define conditions on both the structure and the contents of Web sites

* This work has been partially supported by MCYT under grants TIC2001-2705-C03-01 and HU2003-0003.

in a simple and concise way. For instance, it allows us to enforce that some information is available at a given Web page, some links between pages do exist or even the existence of the Web pages themselves. In our formalism, web pages (HTML/XML documents) are modeled as Herbrand terms, and, consequently, Web sites are finite sets of terms. Then, we formalize a verification technique in which a Web site is checked w.r.t. a given Web specification in order to detect incomplete and/or missing Web pages. Moreover, by analyzing the requirements not fulfilled by the Web site, we are also able to find out the missing information which is needed to repair the Web site. Since reasoning on the Web calls for formal methods specifically fitting the Web context, we develop a novel, rewriting-based technique called *partial rewriting*, in which the traditional pattern matching mechanism is replaced with tree *simulation* [14] in order to provide a suitable mechanism for recognizing patterns inside semistructured documents. The notion of simulation has been already used before for dealing with semistructured data in a number of query and transformation languages [6,8,13,9]. The reason is twofold: on the one hand, it provides a powerful method to extract information from semistructured data; on the other hand, efficient algorithms exist for computing simulations [14]. To assess the feasibility and efficiency of our approach, we have implemented the prototype system VERDI (VErification and Rewriting for Debugging Internet sites), which is based on the verification methodology that we propose and is publicly available online.

Plan of the paper. Section 2 summarizes some preliminary definitions and notations. In Section 3, we formulate a simple method for translating HTML/XML documents into Herbrand terms. Section 4 is devoted to formalize the specification language, whereas Section 5 formalizes the *partial rewriting* mechanism, which is based on page simulation. In Section 6, we introduce our verification technique, which is formalized as a fixpoint computation. First, the set of requirements to be fulfilled by the Web site `W` is computed as the fixpoint of a suitable operator associated with the Web site specification `I`. Then, by using simulations we select those requirements which are not satisfied by `W` and the corresponding incomplete/missing Web pages which are the source for the errors. The requirements which are not satisfied also allow us to ascertain the missing information which is needed to repair the Web site. Some notes regarding the implementation of the system VERDI are given in Section 7. Section 8 concludes. More details and missing proofs can be found in [2].

2 Preliminaries

We call *alphabet* a finite set of symbols. Given the alphabet A, A^* denotes the set of all finite sequences of elements over A. Syntactic equality between objects is represented by $\equiv$.

By $\mathcal{V}$ we denote a countably infinite set of variables and Σ denotes a set of function symbols, or *signature*. We consider varyadic signatures (i.e signatures in which function symbols have an un-bounded arity, that is, they may be followed by an arbitrary number of arguments) as in [10]. $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ denote the *non-ground term algebra* and the *term algebra* built on $\Sigma \cup \mathcal{V}$ and Σ, respectively. $\tau(\Sigma)$ is usually called the Herbrand universe over Σ. A term t is *linear*, if no variable appears more than once in t.

Terms are viewed as labelled trees in the following way: a term in $\tau(\Sigma)$ is a tree $(V, E, r, label)$, where V is a set of vertices, E is a set of edges (i.e. pairs of vertices), $r \in V$ is the *root* vertex and *label* is a *labeling* function such that $label(v) \in (\Sigma \cup \mathcal{V})$, for each $v \in V$. Let us see a small example.

Example 1. Consider the term $t \equiv (f(g(a), X))$ in $\tau(\{f, g, a\}, \{X\})$. Term t can be represented by the structure $(V, E, r, label)$, where $V = \{v_0, v_1, v_2, v_3\}$, $E = \{(v_0, v_1), (v_0, v_2), (v_1, v_3)\}$, $r \equiv v_0$, and function *label* is defined as follows: $label(v_0) = f$, $label(v_1) = g$, $label(v_2) = X$, $label(v_3) = a$.

Given two vertices $v, v' \in V$ of a term $t \equiv (V, E, r, label)$, by $v \geq v'$ we mean that v is a *descendant* of v' in t. By $t|_v$ we mean the subterm rooted at vertex v of t. We denote the *depth* of a vertex v in a term t, that is the number of edges between r and v in t, as $depth(t, v)$. A *substitution* $\sigma \equiv \{X_1/t_1, X_2/t_2, \ldots\}$ is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\tau(\Sigma, \mathcal{V})$. By $Var(t)$ we denote the set of variables occurring in term t.

In the following, we consider marked terms. Given Σ and $\mathcal{V}$, we denote the *marked* version of Σ ($\mathcal{V}$, respectively) as $\underline{\Sigma}$ ($\underline{\mathcal{V}}$, respectively). A syntactic object $\underline{o} \in \underline{\Sigma} \cup \underline{\mathcal{V}}$ is called the *marked version* of $o \in \Sigma \cup \mathcal{V}$. Given a term $t \equiv (V, E, r, label) \in \tau(\Sigma, \mathcal{V})$, a *marking* for t is a (boolean) function $\mu \colon V \to \{yes, no\}$. The *empty* marking ε for t is a marking for t such that $\varepsilon(v) = no$, for each $v \in V$. We define the *marked part* of a term t as

$$mark(t, \mu) \equiv (\{v \in V \mid \mu(v) = yes\}, \{(v_1, v_2) \in E \mid \mu(v_1) = \mu(v_2) = yes\}, r, label).$$

A *valid* marking μ for a term $t \equiv (V, E, r, label)$ is the empty marking for t or a marking for t such that the two following conditions hold:

1. $\mu(r) = yes$;
2. $mark(t, \mu)$ is a term in $\tau(\Sigma, \mathcal{V})$.

Given a term $t \equiv (V, E, r, label)$ and a valid marking μ for t, by slightly abusing notation we recursively define a *marked* term $\mu(t)$ as follows:

$$\mu(t) = \begin{cases} \underline{X} & t \equiv (\{v\}, \emptyset, v, label) \wedge label(v) = X \in \mathcal{V} \\ & \wedge\, \mu(v) = yes \\ X & t \equiv (\{v\}, \emptyset, v, label) \wedge label(v) = X \in \mathcal{V} \\ & \wedge\, \mu(v) = no \\ \underline{f}(\mu(t_1), \ldots, \mu(t_n)) & t \equiv (V, E, r, label) \equiv f(t_1, \ldots, t_n) \wedge\, \mu(r) = yes \\ f(\mu(t_1), \ldots, \mu(t_n)) & t \equiv (V, E, r, label) \equiv f(t_1, \ldots, t_n) \wedge\, \mu(r) = no \end{cases}$$

When no confusion can arise, we simply denote the marked term $\varepsilon(t)$ by t.

Example 2. Consider again term $t \equiv (f(g(a), X))$ of Example 1. Let μ_1 be a marking for t defined as $\mu_1(v_0) = \mu_1(v_2) = \mu_1(v_3) = yes$, $\mu_1(v_1) = no$. Additionally, let μ_2 be a marking for t such that $\mu_2(v_0) = \mu_2(v_1) = yes$, $\mu_2(v_2) = \mu_2(v_3) = no$. Note that μ_1 is not a valid marking for t as the marked part of t is not a term in $\tau(\{f, g, a\}, \{X\})$, whereas μ_2 is valid for t and $\mu_2(t) = \underline{f}(\underline{g}(a), X)$ is a marked term.

3 Denotation of Web Sites

In this paper, a *Web page* is either an XML[18] or a HTML[17] document, and a *Web site* is a finite collection of Web pages. In the sequel, we provide a formalization of these concepts by means of semistructured expressions, which can be seen as an abstract syntax which generalizes the two markup languages XML and HTML. Then, we show how semistructured expressions can be translated into ordinary terms of a given term algebra in such a way that Web sites are represented as finite sets of (ground) terms.

Semistructured Expressions. XML/HTML documents consist of nested structured data, which can be defined inductively. Abstracting from XML and HTML, we give a formal definition of semistructured expressions which are suitable for representing structured documents written in one of these two languages.

Let us consider two alphabets T and $\mathcal{T}ag$. We denote the set T^* by $\mathcal{T}ext$. An object $\mathtt{t} \in \mathcal{T}ag$ is called *tag* element, while an element $\mathtt{w} \in \mathcal{T}ext$ is called *text* element. A *semistructured expression* $\mathtt{e}$ over $\mathcal{T}ext$ and $\mathcal{T}ag$ sets can be specified by the following syntax[1]

$$\begin{array}{rcl} \mathtt{e} & := & <\mathtt{t}>\ \mathtt{elist}\ </\mathtt{t}> \ \mid\ \mathtt{w} \qquad \forall\, \mathtt{w} \in \mathcal{T}ext, \mathtt{t} \in \mathcal{T}ag \\ \mathtt{elist} & := & \mathtt{e}\,\mathtt{elist} \ \mid\ \epsilon \end{array}$$

We denote the set of all the semistructured expressions over $\mathcal{T}ext$ and $\mathcal{T}ag$ by $\mathcal{S}(\mathcal{T}ext, \mathcal{T}ag)$. Note that $\mathcal{T}ext \subseteq \mathcal{S}(\mathcal{T}ext, \mathcal{T}ag)$.

Example 3. The following object is a semistructured expression.

```
<members>
    <member>
       <name> mario </name>
       <surname> rossi </surname>
       <status> professor </status>
    </member>
    <member>
       <name> franca </name>
       <surname> bianchi </surname>
       <status> technician </status>
    </member>
    <member>
       <name> giulio </name>
       <surname> verdi </surname>
       <status> student </status>
    </member>
</members>
```

Roughly speaking, a semistructured expression is either a raw or a structured piece of text, where the structure is provided by tags. Consequently, tags allow us to mark up some textual content, which may contain an arbitrary amount of further well-bracketed markup. Informally, the more tags we add, the more the text is structured, and in some sense its "formal organization" will also increase. Note that we have not explicitly dealt with XML/HTML attributes, as they can be seen as common tagged elements and thus modeled as semistructured expressions. On the other hand, without loss of generality, other XML/HTML features such as namespaces, DTDs and/or schemas, that are not relevant to this work are not conveyed by our notion of semistructured expression.

In the literature, slightly different formalisms have been introduced for modeling XML and HTML documents, e.g. in [1] semistructured expressions are directed graphs which can deal with crossing references. Nevertheless, we prefer the hierarchical representation which does not cause any serious restriction in many practical contexts while it greatly simplifies our methodology.

[1] Note that symbol ϵ in the syntax given for semistructured expressions denotes the empty string and must not be confused with the empty marking ε.

Term representation. Semistructured expressions are provided with a tree-like structure, therefore they can be conveniently translated into terms by applying the following straightforward transformation.

Definition 1. Let $\mathtt{e}$ be a semistructured expression over $\mathcal{T}ext$ and $\mathcal{T}ag$. Then, $\mathtt{e}$ is represented by a term of the Herbrand universe $\tau(\mathcal{T}ext \cup \mathcal{T}ag)$ by the translation $s\_to\_t\colon \mathcal{S}(\mathcal{T}ext, \mathcal{T}ag) \to \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ defined as follows:

$$s\_to\_t(\mathtt{e}) = \begin{cases} \mathtt{w} & \text{if } \mathtt{e} \equiv \mathtt{w} \in \mathcal{T}ext \\ \mathtt{t}(s\_to\_t(\mathtt{e}_1), \ldots, s\_to\_t(\mathtt{e}_n)) & \text{if } \mathtt{e} \equiv \texttt{<t>}\ \mathtt{e}_1 \ldots \mathtt{e}_n \texttt{</t>} \end{cases}$$

Example 4. Consider again semistructured expression of Example 3. Then, the term $\mathtt{p}$ computed by function s_to_t for that semistructured expression is

```
members(
   member(name(mario),surname(rossi),status(professor)),
   member(name(franca),surname(bianchi),status(technician)),
   member(name(giulio),surname(verdi),status(Student))
)
```

To summarize, a Web page, which is coded as an HTML/XML document, can be represented as a semistructured expression, which is then easily translated into a corresponding term of a suitable term algebra. Therefore, in the remaining of this work, a Web page is modeled by a term in $\tau(\mathcal{T}ext \cup \mathcal{T}ag)$. Besides, a *marked* Web page is defined as $\mu(\mathtt{p})$, where $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ and μ is a valid marking for $\mathtt{p}$. A *Web site* is a finite collection of marked Web pages $\{\varepsilon(\mathtt{p}_1) \ldots \varepsilon(\mathtt{p}_n)\}$. In the following, we will also consider terms of the non-ground term algebra $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$, which may contain variables. An element $\mathtt{s} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ is called *Web page template*. $\mu(\mathtt{s})$ is a *marked* Web page template, when $\mathtt{s} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ and μ is a valid marking for $\mathtt{s}$. In our methodology, (marked) Web page templates are used for specifying properties on Web sites as described in the following section.

4 Web specification language

In the following, we present a term rewriting specification language, which is helpful to express properties about the content and the structure of a given Web site. Roughly speaking, a specification is a finite set of rules, where the terms in the left-hand side and in the right-hand side of each rule represent (eventually marked) Web page templates. The operational mechanism, formalized in Section 5, is based on a novel rewriting-based mechanism, which is able to extract partial structure from a term, and then rewrite it.

Formally, Web site specifications are as follows.

Definition 2. A *rule* is a pair of terms $\mathtt{l} \rightharpoonup \mu(\mathtt{r})$ such that $\mathtt{l}, \mathtt{r} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$, $\mathtt{l}$ is linear, $Var(\mathtt{r}) \subseteq Var(\mathtt{l})$ and μ is a valid marking for $\mathtt{r}$. A *Web site specification* $\mathtt{I}$ is a finite set of rules $\{\mathtt{l}_1 \rightharpoonup \mu_1(\mathtt{r}_1), \ldots, \mathtt{l}_n \rightharpoonup \mu_n(\mathtt{r}_n)\}$.

Given a Web specification $\mathtt{I}$, we denote the set of all left-hand sides (right-hand sides disregarding markings) of rules in $\mathtt{I}$ by $\mathtt{Lhs}_\mathtt{I}$ ($\mathtt{Rhs}_\mathtt{I}$, respectively). In symbols, $\mathtt{Lhs}_\mathtt{I} = \{\mathtt{l} \mid \mathtt{l} \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I}\}$ and $\mathtt{Rhs}_\mathtt{I} = \{\mathtt{r} \mid \mathtt{l} \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I}\}$.

The following example illustrates the definition of a Web specification. Marks are introduced by the user to help locating errors. We do not take care of marks for the time being but postpone the formal handling of marking information and the description of the verification framework to Section 6.

Example 5. Consider the following Web specification, which models some required properties of a research group Web site containing information about group members affiliation, scientific publications and personal data.

$$\begin{aligned}
\texttt{member(name(X), surname(Y))} &\rightharpoonup \underline{\texttt{hpage}}\texttt{(name(X), }\underline{\texttt{surname}}\texttt{(}\underline{\texttt{Y}}\texttt{), status)}\\
\texttt{hpage(status(professor))} &\rightharpoonup \underline{\texttt{hpage}}\texttt{(}\underline{\texttt{status}}\texttt{(}\underline{\texttt{professor}}\texttt{), teaching)}\\
\texttt{pubs(pub(name(X), surname(Y)))} &\rightharpoonup \underline{\texttt{member}}\texttt{(name(X), surname(Y))}
\end{aligned}$$

First rule formalizes the following property: if there is a Web page containing a member list, then for each member, a home page exists containing (at least) the name, the surname and the status of this member. Second rule states that whenever a home page of a professor is recognized, then that page must also include some teaching information. Finally, the third rule specifies that whenever there exists a Web page containing information about scientific publications, each author of a publication should be a member of the research group.

Informally, rules of a Web specification formalize conditions to be fulfilled by a given Web site. Intuitively, the interpretation of a rule $\mathtt{l} \rightharpoonup \mu(\mathtt{r})$ w.r.t. a Web site $\mathtt{W}$ is as follows: if (an instance of) $\mathtt{l}$ is recognized in $\mathtt{W}$, also (an instance of) $\mathtt{r}$ must be recognized in a subset of $\mathtt{W}$, which is determined by computing the sets of all Web pages which embed (an instance of) the marked part of $\mathtt{r}$. This mechanism is formalized by partial rewriting.

5 Partial Rewriting

In order to mechanize the intended semantics of Web specification rules, we first devise a mechanism which is able to recognize the structure and the labeling of a given Web page template inside a particular page of the Web site. This is provided by page simulation.

5.1 Page Simulations

The notion of *page simulation* for Web pages allows us to analyze and extract the partial structure of the Web site which is subject to verification.

Roughly speaking, a Web page $\mathtt{p}_1$ is simulated by a Web page $\mathtt{p}_2$, if the tree-structure of $\mathtt{p}_1$ is "embedded" into the tree-structure of $\mathtt{p}_2$. In other words, a simulation of a Web page (i.e. a labelled tree) $\mathtt{p}_1$ in a Web page $\mathtt{p}_2$ can be seen as a relation among the nodes of $\mathtt{p}_1$ and the nodes of $\mathtt{p}_2$ which preserves the edges and the labelings. Before formalizing the idea, we illustrate it by means of a rather intuitive example.

Example 6. Consider the following Web pages (called $\mathtt{p}_1$ and $\mathtt{p}_2$, respectively):

```
hpage(name,surname,status(professor),teaching)

hpage(name(mario),surname(rossi),status(professor),
      teaching(course(logic1),course(logic2)),
      hobbies(hobby(reading),hobby(gardening)))
```

Looking at Figure 1, we observe that the structure of $\mathtt{p}_1$ can be recognized inside the structure of $\mathtt{p}_2$ by considering the relation among nodes of $\mathtt{p}_1$ and nodes of $\mathtt{p}_2$ which is described by the dashed arrows in the figure. This relation essentially provides the so-called *simulation of* $\mathtt{p}_1$ *in* $\mathtt{p}_2$. Note that vice-versa does not hold: no relations can be found among nodes of $\mathtt{p}_2$ and nodes of $\mathtt{p}_1$, which "embed" the structure of $\mathtt{p}_2$ into $\mathtt{p}_1$. In other words, there does not exist a simulation of $\mathtt{p}_2$ in $\mathtt{p}_1$.

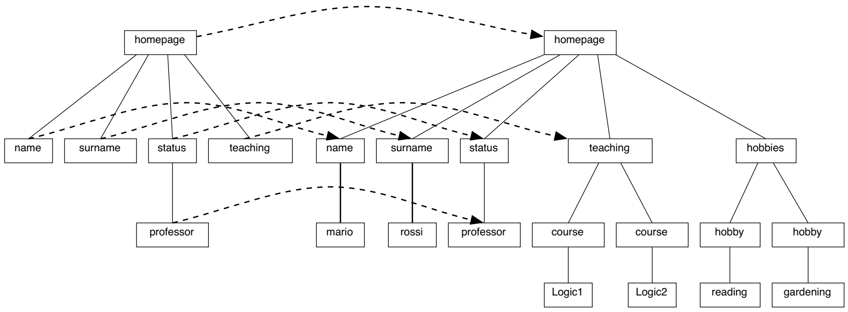

Figure 1. Page simulation between $\mathtt{p}_1$ and $\mathtt{p}_2$.

Simulations have been used in a number of works dealing with querying and transformation of semistructured data. For instance, [1,13] propose some techniques based on simulation for analyzing semistructured data w.r.t. a given schema. The language Xcerpt [7,6] is a (logic) query language for XML and semistructured documents which implements a sort of unification by exploiting the notion of graph simulation. Other approaches involving simulation, or closely related notions, have been employed to measure similarity among semistructured documents [4]. To keep our framework simple, we do not consider a semantic change/load for labels; this would require to introduce ontologies, which are outside the scope of the paper.

Basically, the reason why simulations are successfully employed in the implementation of these kinds of manipulation and querying methods is twofold. Firstly, it is a simple and powerful technique to extract and recognize the partial structure of a document; secondly, there are several efficient algorithms to compute (graph and tree) simulations (see [14]).

In the following, we provide our notion of simulation which is a slight adaptation of the one given in [6] to consider Web page templates: we generalize the usual label relation to cope with the case when variables are used as labels, in the following definition.

Definition 3. Let $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ be two Web page templates in $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. The *label* relation $\sim \subseteq V_1 \times V_2$ is defined as follows:

$$v_1 \sim v_2 \quad \text{iff} \quad label_1(v_1) = label_2(v_2) \text{ or } label_1(v_1) \in \mathcal{V}.$$

Definition 4. Let $\mathtt{s}_1 \equiv (r_1, V_1, E_1, label_1)$, $\mathtt{s}_2 \equiv (r_2, V_2, E_2, label_2)$ be two Web page templates in $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ and $\sim \subseteq V_1 \times V_2$ be the corresponding label relation. A *page simulation* of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t $\sim$ is a relation $\mathbf{S} \subseteq V_1 \times V_2$ such that, for each $v_1 \in V_1, v_2 \in V_2$

1. $r_1 \, \mathbf{S} \, r_2$;
2. $v_1 \, \mathbf{S} \, v_2 \implies v_1 \sim v_2$;
3. $v_1 \, \mathbf{S} \, v_2 \wedge (v_1, v_1') \in E_1 \implies \exists\, v_2' \in V_2,\ v_1' \, \mathbf{S} \, v_2' \wedge (v_2, v_2') \in E_2$.

We define the *projection* of a simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t $\sim$ as $\pi(\mathbf{S}) = \{v_2 \mid (v_1, v_2) \in \mathbf{S}\}$.

Roughly speaking, Definition 4 ensures two degrees of similarity between Web page templates, not only w.r.t. the labelings but also w.r.t. the structures of the templates. On the one hand, Condition (2) of Definition 4 formalizes the similarity w.r.t labelings, that is, any pair of nodes (v, v') in a page simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ have the same label, otherwise node v must be labelled by a variable, which somehow means that the label of v can be seen as a generalization of any concrete label of v'. Finally, Condition (1) and Condition (3) provide a relation between the tree structure of $\mathtt{s}_1$ and the tree structure of $\mathtt{s}_2$.

Note that simulations are just relations among nodes of two given Web page templates. For our purposes, we are interested in simulations which are injective mappings from nodes of a given Web page template to nodes of another Web page template. As it will be apparent later, those simulations allow us to project the structure of a Web page template into another one, thus performing a sort of "partial" pattern matching between templates, which will be exploited to formulate our verification technique.

In the following, we define a subclass of simulations called minimal simulations.

Definition 5. Let $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ be two Web page templates in $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. A page simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$ is *minimal* if there are no page simulations $\mathbf{S}'$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$ such that $\mathbf{S}' \subseteq \mathbf{S}$.

Let us see an example which illustrates the notion of minimal simulation.

Example 7. Let us consider the following Web page templates $\mathtt{s}_1$ and $\mathtt{s}_2$: `hobbies(hobby(X))`, `hobbies(hobby(reading),hobby(gardening))`. In Figure 2(a), the dashed arrows represent a non-minimal simulation of $\mathtt{s}_1$ in $\mathtt{s}_2$, while in Figures 2(b) and 2(c) two minimal simulations of $\mathtt{s}_1$ in $\mathtt{s}_2$ are depicted. Note that the last two simulations are mappings.

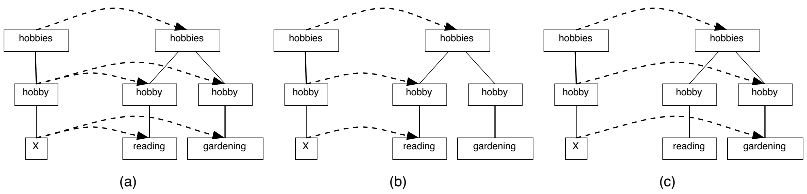

Figure 2. non-minimal and minimal simulations

Lemma 1. *Let $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ be two Web page templates in $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. A minimal page simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$ is a mapping $\mathbf{S} : V_1 \to V_2$.*

Minimal simulations do not guarantee that the tree structure of a given Web page template can be recognized inside another template. For this purpose, we need to furtherly restrict our class of simulations. Let us see an example.

Example 8. Consider Web page templates $\mathtt{s}_1 \equiv \mathtt{f(X,Y)}$ and $\mathtt{s}_2 \equiv \mathtt{f(a)}$. Note that there exists a minimal page simulation of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$ (see Figure 3), but the tree structure of $\mathtt{s}_1$ cannot be recognized as part of $\mathtt{s}_2$, e.g. the vertex with label `f` in $\mathtt{s}_1$ has two outgoing edges, while the corresponding vertex in $\mathtt{s}_2$ has only one.

To solve the problem presented in Example 8, we simply restrict ourselves to consider minimal *injective* page simulations, which provide a one-to-one correspondence among edges of the two considered Web page templates.

It is not difficult to demonstrate that minimal injective simulations are particular instances of Kruskal's *embeddings* [5] w.r.t. the relation $\sim$. In other words, a minimal injective

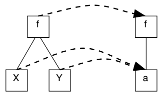

Figure 3. minimal non-injective simulation

page simulation of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$ exists iff $\mathtt{s}_1$ is embedded into $\mathtt{s}_2$ w.r.t. $\sim$, i.e., we are able to find out the structure and the labeling of $\mathtt{s}_1$ inside $\mathtt{s}_2$. Note that the minimal simulation of $\mathtt{s}_1$ in $\mathtt{s}_2$ depicted in Figure 3 is not injective and thus no embedding of $\mathtt{s}_1$ into $\mathtt{s}_2$ exists. Instead, Figures 2(b) and 2(c) illustrate two minimal injective simulations, that is, two embeddings between Web page templates.

5.2 Rewriting Web page templates

Definition 6. Let $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2) \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. We say that $\mathtt{s}_2$ *partially matches* $\mathtt{s}_1$ via substitution σ iff

1. there exists a minimal injective simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$;
2. for each $(v, v') \in \mathbf{S}$ such that $label(v) = X \in \mathcal{V}$, $\sigma(X) = (\mathtt{s}_{2|v'})$.

In Definition 6, we consider only minimal injective simulations between Web page templates $\mathtt{s}_1$ and $\mathtt{s}_2$, since this trivially ensures the existence of a substitution σ such that there exists a simulation of $\mathtt{s}_1\sigma$ in $\mathtt{s}_2$ w.r.t. $\sim$; in other words, $\mathtt{s}_1\sigma$ is embedded into $\mathtt{s}_2$.

Example 9. Consider again Example 7. We have that $\mathtt{s}_2$ partially matches $\mathtt{s}_1$ via $\{X/reading\}$ (see Figure 2(b)) and $\mathtt{s}_2$ partially matches $\mathtt{s}_1$ via $\{X/gardening\}$ (see Figure 2(c)). Note that performing partial matching by the non-minimal simulation of Figure 2(a) would produce $\sigma \equiv \{X/reading, X/gardening\}$, which is not a substitution.

Now we are ready to define a partial rewrite relation between marked Web page templates.

Definition 7. Let $\mathtt{s} \equiv (V, E, r, label)$, $\mathtt{t} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. Let μ_1 and μ_2 be two valid markings for $\mathtt{s}$ and $\mathtt{t}$, respectively. Then, $\mu_1(\mathtt{s})$ *partially rewrites* to $\mu_2(\mathtt{t})$ via rule $r \equiv \mathtt{l} \rightharpoonup \mu(\mathtt{r})$ and substitution σ (in symbols, $\mu_1(\mathtt{s}) \rightharpoonup_r^\sigma \mu_2(\mathtt{t})$) iff there exists $v \in V$ such that

1. $\mathtt{s}_{|v}$ *partially matches* $\mathtt{l}$ via σ;
2. $\mathtt{t} = \mathtt{r}\sigma$.
3. Let $\mathtt{r} \equiv (V_\mathtt{r}, E_\mathtt{r}, r, label_\mathtt{r})$ and $\mathtt{r}\sigma \equiv (V_{\mathtt{r}\sigma}, E_{\mathtt{r}\sigma}, r, label_{\mathtt{r}\sigma})$. For each $v \in V_{\mathtt{r}\sigma}$,

$$\mu_2(v) = \begin{cases} \mu(v) & \text{if } v \in (V_\mathtt{r} \cap V_{\mathtt{r}\sigma}) \\ \mu(v') & \text{if } v \in (V_{\mathtt{r}\sigma} \setminus V_\mathtt{r}) \wedge (\exists\, v' \in V_\mathtt{r}, v \geq v', label_\mathtt{r}(v') \in Var(\mathtt{r})) \end{cases}$$

When rule r and substitution σ are understood, we simply write $\mu_1(\mathtt{s}) \rightharpoonup \mu_2(\mathtt{t})$.

It is worth noting that we provide a notion of partial rewriting in which the context of the selected reducible expression $\mathtt{s}_{|v}$ of the Web page template which is rewritten is disregarded after the rewrite step (see point (2) of Definition 7). Roughly speaking, given a Web specification rule $\mathtt{l} \rightharpoonup \mu(\mathtt{r})$, partial rewriting allows us to extract a subpart of a given Web page (template) $\mathtt{s}$, which partially matches $\mathtt{l}$, and to replace $\mathtt{s}$ by an instance of $\mathtt{r}$; namely, $\mathtt{r}\sigma$ (see points (1) and (2) of Definition 7). Point (3) of Definition 7 establishes that rewritten templates inherit markings from the right-hand sides of the applied rules. More precisely,

- each vertex of $\mathtt{r}\sigma$, which is not affected by substitution σ, maintains the same marking of $\mathtt{r}$;
- each vertex, which belongs to a subterm of $\mathtt{r}\sigma$ replacing a variable $\underline{X}$ of $\mathtt{r}$, is marked *yes*;
- each vertex, which belongs to a subterm of $\mathtt{r}\sigma$ replacing a variable X of $\mathtt{r}$, is marked *no*.

Example 10. Consider the Web page $\mathtt{p}$ of Example 4 and the first rule $\mathtt{r}_1$ of the Web specification of Example 5. Then, Web page template $\varepsilon(\mathtt{p})$ partially rewrites to the following three Web pages by applying $\mathtt{r}_1$.

$$\varepsilon(\mathtt{p}) \rightharpoonup_{\mathtt{r}_1} \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{mario}), \underline{\mathtt{surname}}(\underline{\mathtt{rossi}}), \mathtt{status})$$
$$\varepsilon(\mathtt{p}) \rightharpoonup_{\mathtt{r}_1} \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{franca}), \underline{\mathtt{surname}}(\underline{\mathtt{bianchi}}), \mathtt{status})$$
$$\varepsilon(\mathtt{p}) \rightharpoonup_{\mathtt{r}_1} \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{giulio}), \underline{\mathtt{surname}}(\underline{\mathtt{verdi}}), \mathtt{status})$$

Roughly speaking, markings in the right-hand sides of the rules allow us to find sets of Web pages, which might be incomplete or missing. Then, real buggy pages are detected inside these sets. We formalize the idea in the following section.

6 The verification framework

In the following, we show how simulation and partial rewriting can be applied to verify a given Web site $\mathtt{W}$ w.r.t. a Web specification $\mathtt{I}$. Essentially, the main idea is to compute the set of all possible marked Web pages that can be derived from $\mathtt{W}$ via $\mathtt{I}$ by means of partial rewriting. These marked Web pages can be thought of as requirements to be fulfilled by $\mathtt{W}$. Then, we check whether the computed requirements are satisfied by $\mathtt{W}$ by using simulation and marking information. In summary, the method works in two steps: (1) compute the set of requirements $\mathtt{Req}_{\mathtt{I},\mathtt{W}}$ for $\mathtt{W}$ w.r.t. $\mathtt{I}$, (2) check $\mathtt{Req}_{\mathtt{I},\mathtt{W}}$ in $\mathtt{W}$.

6.1 Computing the set of requirements

Let us introduce the following operator.

Definition 8. Let $\mathtt{T}$ be a set of marked Web page templates and $\mathtt{I}$ be a Web specification. Then,

$$\mathtt{R}_\mathtt{I}(\mathtt{T}) = \mathtt{T} \cup \{\mu_2(\mathtt{s}_2) \mid \exists\, \mu_1(\mathtt{s}_1) \in \mathtt{T}, r \equiv \mathtt{l} \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I} \text{ s.t. } \mu_1(\mathtt{s}_1) \rightharpoonup_r \mu_2(\mathtt{s}_2)\}$$

Roughly speaking, the operator in Definition 8 computes all marked templates which result from partial rewriting the Web page templates of $\mathtt{T}$ by using the Web specification $\mathtt{I}$, and returns the union of the resulting set and $\mathtt{T}$. By repeatedly applying this operator, it is possible to compute all marked Web pages that can be derived from an initial Web site after an arbitrary number of partially rewriting steps. For this purpose, we formalize the *ordinal powers* of the operator $\mathtt{R}_\mathtt{I}$ w.r.t. a Web site $\mathtt{W}$ as follows: $\mathtt{R}_\mathtt{I} \uparrow^{\mathtt{W}} 0 = \mathtt{W}$, $\mathtt{R}_\mathtt{I} \uparrow^{\mathtt{W}} n = \mathtt{R}_\mathtt{I}(\mathtt{R}_\mathtt{I} \uparrow^{\mathtt{W}} (n-1))$, $n > 0$.

It is immediate to demonstrate that the operator $\mathtt{R}_\mathtt{I}$ is continuous on the lattice consisting of the powerset of the term algebra of the marked Web page templates ordered by set inclusion. This ensures that a least fixpoint of $\mathtt{R}_\mathtt{I}$ exists and can be reached after ω applications of $\mathtt{R}_\mathtt{I}$, that is, $\mathtt{R}_\mathtt{I} \uparrow^{\mathtt{W}} \omega$ where ω is the first infinite ordinal. Moreover, the least fixpoint of $\mathtt{R}_\mathtt{I}$ contains all the marked Web pages derivable from Web pages in $\mathtt{W}$ via $\mathtt{I}$.

Now, recalling the interpretation of the rules of the Web site specification given in Section 4, Web pages derived by the application of a Web specification must be recognized as (part of) some Web page in the Web site. Therefore, those Web pages in the least fixpoint of $\mathtt{R_I}$ which are not in $\mathtt{W}$ can be intended as requirements to be fulfilled by $\mathtt{W}$. Thus, we define the *set of requirements* for $\mathtt{W}$ w.r.t. $\mathtt{I}$ as $\mathtt{Req}_{\mathtt{I},\mathtt{W}} = \mathit{lfp}(\mathtt{R_I}) \setminus \mathtt{W}$, where $\mathit{lfp}(\mathtt{R_I})$ is the least fixpoint of the operator $\mathtt{R_I}$.

Clearly, the fixpoint of $\mathtt{R_I}$ (and hence $\mathtt{Req}_{\mathtt{I},\mathtt{W}}$) for an arbitrary Web specification might be infinite. Consider for instance the following example.

Example 11. Let $\mathtt{W} \equiv \{\mathtt{h(g(0),f(0))}\}$ be a Web site and $\mathtt{I} \equiv \{\mathtt{h(g(X))} \rightharpoonup \mathtt{h(g(g(X)))}\}$ be a Web specification. Then, $\mathtt{Req}_{\mathtt{I},\mathtt{W}} = \{\mathtt{h(g(g(0)))}, \mathtt{h(g(g(g(0))))}, \mathtt{h(g(g(g(g(0)))))}, \ldots\}$ is an infinite set of requirements which is infinite.

Fortunately, the computation of the set of requirements is finite for some interesting classes of Web specifications. Trivially, non-recursive specifications allow to reach $\mathit{lfp}(\mathtt{R_I})$ after a finite number of applications of $\mathtt{R_I}$, i.e., $\mathit{lfp}(\mathtt{R_I}) = \mathtt{R_I} \uparrow^{\mathtt{W}} k$, $k \in \mathbb{N}$. However, non-recursive definitions are not expressive enough for verification purposes, since some relevant conditions about Web sites cannot be formalized without resorting to recursion; e.g., some properties stated in Example 5 cannot be formulated by using a non-recursive specification.

In the following, we define a class of recursive Web specifications for which the set of requirements is finite. Basically, the idea is to consider those specifications for which the computation of the least fixpoint only generates Web pages whose size is bounded.

The following definition formalizes the considered class of Web site specifications.

Definition 9. A Web specification $\mathtt{I}$ is *bounded* iff, for each $\mathtt{l} \equiv (V_1, E_1, r_1, \mathit{label}_1) \in \mathtt{Lhs_I}$, $\mathtt{r} \equiv (V_2, E_2, r_2, \mathit{label}_2) \in \mathtt{Rhs_I}$ and each minimal injective simulation $\mathbf{S}$ of $\mathtt{l}$ in $\mathtt{r}_{|v}$ w.r.t. $\sim$, $v \in V_2$, the following property holds

$$\text{if } v_2 \in \pi(\mathbf{S}) \text{ and } \mathit{label}_2(v_2) \in \mathit{Var}(\mathtt{r}_{|v}), \text{ then for all } v_1 \in V_1 \text{ s.t. } \mathit{label}_1(v_1) \in \mathit{Var}(\mathtt{l}),$$
$$\mathit{depth}(\mathtt{r}_{|v}, v_2) = \mathit{depth}(\mathtt{l}, v_1).$$

Roughly speaking, Definition 9 states that, whenever a left-hand side $\mathtt{l}$ of a rule is simulated by (a subterm of) the right-hand side $\mathtt{r}$ of a (possibly different) rule, then no variables in the substructure of $\mathtt{r}$ which is recognized by simulation must be located at positions which are deeper than all the positions of the variables in $\mathtt{l}$.

Example 12. Consider again the specification $\mathtt{I}$ in Example 11. The left-hand side of the rule $\mathtt{h(g(X))} \rightharpoonup \mathtt{h(g(g(X)))}$ is simulated by its own right-hand side. Moreover, variable $\mathtt{X}$ in the right-hand side is located at depth 3, while the unique variable in the left-hand side is at depth 2. Thus, $\mathtt{I}$ is not bounded.

Now, take into account specification

$$\mathtt{I}' \equiv \{\mathtt{m(n(X))} \rightharpoonup \mathtt{h(n(X),s(s(X)))}, \mathtt{h(n(X))} \rightharpoonup \mathtt{m(n(X),t)}\}.$$

Then, $\mathtt{m(n(X))}$ is simulated by $\mathtt{m(n(X),t)}$ and $\mathtt{h(n(X))}$ is simulated by $\mathtt{h(n(X),s(s(X)))}$. In both cases, variables occurring in the substructures of the right-hand sides which are recognized by simulation and variables of the respective left-hand sides are located at the same depth. Therefore, the Web specification $\mathtt{I}'$ is bounded.

For bounded Web specifications, the least fixpoint of the operator $\mathtt{R_I}$ is finite as stated by the next proposition. This provides an effective method for computing the set of requirements $\mathtt{Req}_{\mathtt{I},\mathtt{W}}$.

Proposition 1. *Let* $\mathtt{I}$ *be a bounded Web specification and* $\mathtt{W}$ *be a Web site. Then, there exists* $k \in \mathbb{N}$ *such that* $lfp(\mathtt{R_I}) = \mathtt{R_I} \uparrow^{\mathtt{W}} k$.

Example 13. Consider the bounded Web specification I of Example 5 and the following Web site W:

```
W = { members(member(name(mario),surname(rossi),status(professor)),
             member(name(franca),surname(bianchi),status(technician)),
             member(name(giulio),surname(verdi),status(student))),
      hpage(name(mario),surname(rossi),phone(3333),status(professor),
            hobbies(hobby(reading),hobby(gardening))),
      hpage(name(franca),surname(bianchi),status(technician),phone(5555)),
      hpage(name(anna),surname(gialli),status(professor),phone(4444),
            teaching(course(algebra))),
      pubs(pub(name(mario),surname(rossi),title(blahblah1),year(2003)),
           pub(name(anna),surname(gialli),title(blahblah2),year(2002))) }
```

Then, the set of computed requirements $\mathtt{Req_{I,W}}$ is

$$\begin{array}{l}
\{\ \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{mario}), \underline{\mathtt{surname}}(\underline{\mathtt{rossi}}), \mathtt{status}),\\
\ \ \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{franca}), \underline{\mathtt{surname}}(\underline{\mathtt{bianchi}}), \mathtt{status}),\\
\ \ \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{giulio}), \underline{\mathtt{surname}}(\underline{\mathtt{verdi}}), \mathtt{status}),\\
\ \ \underline{\mathtt{hpage}}(\underline{\mathtt{status}}(\underline{\mathtt{professor}}), \mathtt{teaching}),\\
\ \ \underline{\mathtt{member}}(\mathtt{name}(\mathtt{mario}), \mathtt{surname}(\mathtt{rossi})),\\
\ \ \underline{\mathtt{member}}(\mathtt{name}(\mathtt{anna}), \mathtt{surname}(\mathtt{gialli})),\\
\ \ \underline{\mathtt{hpage}}(\mathtt{name}(\mathtt{anna}), \underline{\mathtt{surname}}(\underline{\mathtt{gialli}}), \mathtt{status})\ \}
\end{array}$$

6.2 Checking requirements in Web sites

As we have seen in Section 5.1, simulation allows us to identify the structure of a given Web page (eventually, a template) into another. By taking advantage of this fact, we can develop a methodology, which is able to discover incompleteness errors in a given Web site w.r.t. a Web specification. Basically, the idea is to verify the consistency of the Web site w.r.t. the set of requirements. To accomplish this task, we first use simulation for checking whether requirements are embedded into some Web page of the considered Web site and then exploit marking information in order to diagnose incompleteness errors in the Web site.

More precisely, our analysis allows us to discover two kinds of incompleteness errors: (1) Web pages which are missing in a Web site w.r.t. a given Web specification, (2) Web pages which are incomplete w.r.t a given Web specification.

Let us first consider the former class of errors.

Definition 10. Let W be a Web site, I be a bounded Web specification and $\mathtt{Req_{I,W}}$ be the set of requirements for W w.r.t. I. Let $\mu(\mathtt{e}) \in \mathtt{Req_{I,W}}$. The *likely missed information set* w.r.t. $\mu(\mathtt{e})$ is defined as

$$LMIS_{\mu(\mathtt{e})} = \{\mathtt{p} \equiv (V, E, r, label) \in \mathtt{W} \mid \text{there is a minimal injective simulation of } mark(\mathtt{e}, \mu) \text{ in } \mathtt{p}_{|v} \text{ w.r.t.} \sim, \text{ with } v \in V\}.$$

Roughly speaking, this definition allows us to compute a subset of the Web site containing all the web pages which are simulated by the marked part of a given requirement. These web pages could be potentially incomplete w.r.t. the web specification, since they might not satisfy the considered requirement. Let us see an example.

Example 14. Let us consider the rule r

$$\texttt{hpage(status(professor))} \rightharpoonup \underline{\texttt{hpage}}(\underline{\texttt{status}}(\underline{\texttt{professor}},\texttt{teaching}))$$

and the website W of Example 13. Rule r allows us to check whether web pages of professors contain some teaching information. Clearly, requirements computed by this rule should be only checked in such web pages. For this purpose, we use the marking information in the rhs of r in order to focus on the professor web pages. Let us consider the requirement

$$\mu_1(\texttt{e}_1) \equiv \underline{\texttt{hpage}}(\underline{\texttt{status}}(\underline{\texttt{professor}},\texttt{teaching}))$$

which can be derived from W by means of r. By applying Definition 10, we get

$$\begin{aligned} LMIS_{\mu_1(\texttt{e}_1)} = \{(1)\ & \texttt{hpage(name(mario), surname(rossi),} \\ & \quad \texttt{phone(3333), status(professor),} \\ & \quad \texttt{hobbies(hobby(reading), hobby(gardening))),} \\ (2)\ & \texttt{hpage(name(anna), surname(gialli),} \\ & \quad \texttt{status(professor), phone(4444),} \\ & \quad \texttt{teaching(course(algebra)))}\ \}. \end{aligned}$$

which contains only professor web pages to be checked for incompleteness errors.

From Definition 10, we can easily derive that, whenever the likely missed information set is empty for a given requirement $\mu(\texttt{e})$, $\mu(\texttt{e})$ is not recognized in any Web page of the Web site. In other words, that requirement identifies a missing element in the Web site.

Definition 11. Let W be a Web site, I be a bounded Web specification and $\texttt{Req}_{\texttt{I},\texttt{W}}$ be the set of requirements for W w.r.t. I. Let $\mu(\texttt{e}) \in \texttt{Req}_{\texttt{I},\texttt{W}}$ and $\texttt{p} \in \texttt{W}$. Then, $\mu(\texttt{e})$ is *missing* in W w.r.t I iff $LMIS_{\mu(\texttt{e})} = \emptyset$.

Let us see an example for clarifying our definitions.

Example 15. Consider again the set of requirements $\texttt{Req}_{\texttt{I},\texttt{W}}$ computed in Example 13. Then, $\mu(\texttt{e}) \equiv (\underline{\texttt{hpage}}(\texttt{name(giulio)}, \underline{\texttt{surname}}(\underline{\texttt{verdi}}), \texttt{status}))$ is missing in W w.r.t. I, since $LMIS_{\mu(\texttt{e})} = \emptyset$. Indeed, the requirement $\mu(\texttt{e})$ identifies a "group member" home page which does not appear in the Web site W.

Let us consider now incompleteness errors which refer to incomplete pages, that is, Web pages in which some piece of information is lacking (e.g. missing items).

Definition 12. Let W be a Web site, I be a bounded Web specification and $\texttt{Req}_{\texttt{I},\texttt{W}}$ be the set of requirements for W w.r.t. I. Let $\mu(\texttt{e}) \in \texttt{Req}_{\texttt{I},\texttt{W}}$ and $\texttt{p} \in \texttt{W}$. Then, $\texttt{p} \equiv (V, E, r, label)$ is incomplete w.r.t. $\mu(\texttt{e})$ iff

- $\texttt{p} \in LMIS_{\mu(\texttt{e})}$;
- there is a minimal injective simulation of $mark(\texttt{e}, \mu)$ in $\texttt{p}_{|v}$ w.r.t. $\sim$, with $v \in V$, s. t. there is no minimal injective simulation of e in $\texttt{p}_{|v}$ w.r.t. $\sim$.

In this case, we will call $\mu(\texttt{e})$ *incompleteness symptom* for p.

Example 16. Recall the set of requirements $\texttt{Req}_{\texttt{I},\texttt{W}}$ computed in Example 13. Then, consider the requirement $\mu_1(\texttt{e}_1) \equiv (\underline{\texttt{hpage}}(\underline{\texttt{status}}(\underline{\texttt{professor}}), \texttt{teaching}))$, we have that

$$\begin{aligned} LMIS_{\mu_1(\texttt{e}_1)} = \{(1)\ & \texttt{hpage(name(mario), surname(rossi),} \\ & \quad \texttt{phone(3333), status(professor),} \\ & \quad \texttt{hobbies(hobby(reading), hobby(gardening))),} \\ (2)\ & \texttt{hpage(name(anna), surname(gialli),} \\ & \quad \texttt{status(professor), phone(4444),} \\ & \quad \texttt{teaching(course(algebra)))}\ \}. \end{aligned}$$

Now, by applying Definition 12, we detect that Web page (1) is incomplete w.r.t. $\mu_1(\mathtt{e}_1)$, which is therefore an incompleteness symptom for (1). In fact, Web page (1) lacks teaching information.

Consider now the requirement $\mu_2(\mathtt{e}_2) \equiv (\underline{\mathtt{member}}(\mathtt{name}(\mathtt{anna}), \mathtt{surname}(\mathtt{gialli})))$. The associate likely missed information set is

$$LMIS_{\mu_2(\mathtt{e}_2)} = \{\ \mathtt{members}(\ \begin{array}{l}\mathtt{member}(\mathtt{name}(\mathtt{mario}), \mathtt{surname}(\mathtt{rossi}), \mathtt{status}(\mathtt{professor})),\\ \mathtt{member}(\mathtt{name}(\mathtt{franca}), \mathtt{surname}(\mathtt{bianchi}), \mathtt{status}(\mathtt{technician})),\\ \mathtt{member}(\mathtt{name}(\mathtt{giulio}), \mathtt{surname}(\mathtt{verdi}), \mathtt{status}(\mathtt{student})))\ \}.\end{array}$$

Note that the Web page in $LMIS_{\mu_2(\mathtt{e}_2)}$ is incomplete w.r.t. the requirement $\mu_2(\mathtt{e}_2)$, which models the fact that `anna gialli` must be a member of the group. Finally, the remaining requirements do not give rise to further errors.

It is worth pointing out that our verification framework is able to detect both the erroneous Web pages and the cause of the detected errors (i.e., the so-called incompleteness symptoms). This allows us not only to locate bugs and inconsistencies w.r.t. a given specification, but also to easily repair them by comparing incomplete pages to incompleteness symptoms, since the latter provides the missing information which is needed to complete the erroneous Web pages.

7 Implementation

The basic methodology presented so far has been implemented in the preliminary prototype system VERDI (VErification and Rewriting for Debugging Internet sites), which is written in DrScheme v205 [16] and is publicly available together with a set of tests at `http://www.dimi.uniud.it/~demis/#software`.

The implementation consists of about 80 function definitions (approximately 1000 lines of source code). VERDI includes a parser for semistructured expressions and Web specifications, and several modules implementing the user interface, the partial rewriting mechanism and the verification technique. The system allows the user to load a Web site consisting of a finite set of semistructured expressions together with a Web specification. Additionally, he/she can inspect the loaded data and finally check the Web pages w.r.t the Web site specification. The user interface is guided by textual menus, which are (hopefully) self-explaining.

We tested the system on several Web site examples which can be found at the URL address mentioned above. In each considered test case, we were able to detect the errors (i.e. missing and incomplete Web pages) efficiently. For instance, it took less than one second the verification of the Web site of Example 13 w.r.t the Web specification of the Example 5, producing error messages when necessary.

8 Conclusions

Conceiving and maintaining Web sites is a difficult task. In this paper, we provide a rewriting-based, formal specification language which can be used to impose properties both on the structure (syntactic properties) and on the contents (semantic properties) of Web sites. The computation mechanism underlying this language is based on a novel rewriting-like technique, called *partial rewriting*, in which the traditional pattern matching mechanism is replaced with tree *simulation* [14]. In our methodology, Web sites are automatically checked w.r.t. a given Web specification in order to detect incomplete and missing Web pages. Moreover, by analyzing the requirements not fulfilled by the Web site, we are also able to find out the missing information needed to repair the Web site. Our methodology exploits some marking information on terms which represent the Web pages to better locate the errors, which is provided by the user in advance. We have also discussed some implementation details of the preliminary system VERDI, a prototype implemention of the verification framework that we propose.

Finally, let us conclude by mentioning some directions for future work. We are currently extending our framework in order to provide a method for synthesizing the marking information semi-automatically. We also plan to extend the specification language in order to support the detection of regular expressions. This is useful to guarantee that proprietary or "forbidden" data are not displayed on the external version of the site (e.g. a number of credit card). On the practical side, we plan to develop a fully user-friendly system which can help Web administrators to design, check and maintain their Web sites.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.
2. M. Alpuente, D. Ballis, and M. Falaschi. A Rewriting-based Framework for Web Sites Verification. Technical Report, DSIC-II/04, UPV, 2004. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.
3. I. D. Baxter, F. Ricca, and P. Tonella. Web Application Transformations based on Rewrite Rules. *Information and Software Technology*, 44(13), 2002.
4. E. Bertino, M. Mesiti, and G. Guerrin. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. *Information Systems*, 29(1):23–46, 2004.
5. M. Bezem. *TeReSe, Term Rewriting Systems*, chapter Mathematical background (Appendix A). Cambridge University Press, 2003.
6. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. of ICLP'02*, volume 2401 of *LNCS*. Springer-Verlag, 2002.
7. F. Bry and S. Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. Technical report, 2002. Available at: `http://www.xcerpt.org`.
8. P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of ACM SIGMOD ICMD'96*, 1996.
9. A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and abstract semantics of a graphical query language. *TCS*, 275:521–560, 2002.
10. N. Dershowitz and D. Plaisted. Rewriting. *Handbook of Automated Reasoning*, 1:535–610, 2001.
11. T. Despeyroux and B. Trousse. Semantic Verification of Web Sites Using Natural Semantics. In *Proc. of RIAO'00*, 2000.
12. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying Integrity Constraints on Web Site. In *Proc. of IJCAI'99*, volume 2, pages 614–619. Morgan Kaufmann, 1999.
13. M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. of ICDE'98*, pages 14–23, 1998.
14. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *proc. of IEEE FOCS'95*, pages 453–462, 1995.
15. C. Kirchner, Z. Qian, P. K. Singh, and J. Stuber. Xemantics: a Rewriting Calculus-Based Semantics of XSLT. Technical Report A01-R-386, LORIA, 2001.
16. PLT. DrScheme web site. Available at: `http://www.drscheme.org`.
17. World Wide Web Consortium (W3C). HyperText Markup Language (HTML) 4.01, 1997. Available at: `http://www.w3.org`.
18. World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: `http://www.w3.org`.

A Compiler for Mapping a Rule-Based Event-Triggered Program to a Hardware Engine

Carsten Albrecht[1] and Andreas C. Döring[2]

[1] Institute of Computer Engineering, University of Lübeck, Germany
albrecht@iti.uni-luebeck.de
[2] IBM Zurich Research Laboratory, Rüschlikon, Switzerland
ado@zurich.ibm.com

Abstract In this paper we describe the RERAL compiler. RERAL is a Rule-based Event-driven Routing Algorithm Language. It is intended for the configuration of a router for regular networks, as found in parallel computers or computer clusters. The language combines predicate-logic-derived functional expressions with Petri-net-based asynchronity. The high performance requirements (a routing decision should take no more than few nanoseconds) imply sophisticated optimization methods in the compiler, in particular, flattening the program hierarchy, unrolling loops and mapping high-level program fragments to available application-specific hardware units.

We also point out a new application area of the concept, namely the management of a memory interface in a system-on-chip for increased bandwidth utilization.

1 Introduction

Many components of computer or embedded systems use a combination of fixed hardware, processing units, and configurable hardware. For the latter a wide variety of configuration methods are known, but most of them require a detailed understanding of the architecture, tool issues and hardware-related aspects such as timing and resource constraints. When the system has to be programmed by experts, the complexity of this task is acceptable. However, a higher abstraction layer is desirable to allow the efficient use of programmable hardware structures by a non-expert and to make the expert more productive.

In this paper we introduce a rule-based language and its tool environment that were created to describe routing algorithms for the configuration of interconnection networks in parallel computers or computer clusters. A second application, which turned out to be of interest more recently, is the medium-term control of memory-interface usage in Systems-On-Chip (SoC). Both applications have in common that

- several aspects of the application domain are combined,
- the inputs are formed by an infinite series of unrelated external events (such as the arrival of a message or a cache miss in an on-chip CPU), and
- there is some freedom in the reaction of the programmable component.

All these properties are reflected in the language RERAL (Rule-based Event-driven Routing-Algorithm Language). As the name suggests, it combines the aspects of event-triggered parallel evaluation with the descriptive power of rule-based expressions. Through the use of a compilation tool, a program can be transformed in such a way that it can be applied to a VLSI-implementable rule-evaluation engine which performs a complex algorithmic operation in few clock cycles. In particular, a hierarchically described routing decision is flattened such that it can be stored in a small on-chip memory, and individual evaluation requires only one access to this memory thanks to sophisticated address generation.

In this paper we present first the background for the two application areas (Section 3), with emphasis on routing algorithms. In particular we motivate the modular characteristic of the language and the necessary aspects of parallelism and functional complexity. The target architecture is only sketched so as to leave more room for the introduction of the language (Section 4) and the discussion of the compilation process (Section 5). Specifically, we demonstrate how a unification-based method can be used to extract specific functionalities for hardware building blocks in the rule-evaluation engine. If this functionality is spread over several rules in the user program, it has to be detected by the compiler to maintain hardware independence.

2 Related Work

The related disciplines are wide-spread; rule-based systems are typically used in the software world. However, to describe a specialized hardware system, only low-level descriptions with similarity to rule bases are in use. We have been told that the tool 'specializer' as described in [15] is able to perform many of the tasks the compiler presented here does. We did not verify this, like to point out that our compiler performs numerous hardware-specific operations that probably are not covered by a general-purpose tool. An example is the generation of the addressing logic for higher-dimensional arrays without multiplication [9].

Generating the scanner and parser from a given grammar automatically is a well-established technique, and we used Eli [13] for this purpose. For the implementation of the compiler we used the interpreter Hugs for the functional programming language Haskell [16]. The pattern matching features of the language, the rich set of functions for dealing with complex data structures including the automated memory handling provide a framework in which the required transformations in the compiler can be described on a high abstraction level.

The proprietary description language of state machines used in the software suite Log/iC has some similarity with very basic rules. Neonetworks announced a chip called Stream-Processor for networking applications that was claimed to exploit parallelism on a "supercomputer scale" and had a building block called "rule processor". However, information on this technology has been confidential, and the company has since gone out of business.

With respect to routing in parallel computer networks, Summerville et al. present an architecture for a bit-pattern-associative router in [18]. They describe their routing algorithms in a pseudo-language that is very similar to the basic pattern used in RERAL. The target hardware uses a pattern-matching circuit array which is similar to a ternary CAM (Content addressable Memory). As there are neither dedicated arithmetic circuits nor other specialized components in the proposed routing engine, only simple routing algorithms can be carried out without a huge increase in the association circuitry.

In the domain of Internet Protocol (IP)-based networks, rules for routing are popular because of the hierarchical organization of IP addressing. Consequently, there are many architectures that process rules, that combine range checks and prefix matching in IP addresses, together with range checks on the port number. An example is [19]. Because a parallel check is performed, the hardware effort scales with the number of permitted rules. Memory-oriented variants are also in use, e.g. [17]. IP-based routing requires the option to change the rule set dynamically, thus the mapping of the rule set defined on a high abstraction level to the representation in the hardware has to be computable very efficiently.

All these methods impose strong limitations on the structure of the rules, in particular they restrict the type of operations that can be applied to the variant inputs. Furthermore, only one rule set is considered for the reaction to one type of event. The reactions of the rules are very simple, such as a drop/non-drop decision in a fire wall. These limitations inhibit the implementation of advanced algorithms like the ones that will be introduced in the next section.

3 Area of Application

As pointed out in Section 1, we are considering two areas of applications: the routing in regular networks and the management of memory-access bandwidth in a SoC, such as a network processor.

Regular networks use a topology with a high regularity, e.g. a mesh or a hypercube. They are an important part of parallel computers, in particular PC clusters. The regularity allows the use of advanced routing algorithms that

- allow scaling the network size, with nearly constant hardware cost in the routers,
- adapt the path for a data item through a network dynamically, depending on link or router load (adaptivity), and
- route around failed routers or links dynamically (fault tolerance).

A good coverage of these routing algorithms and the basic architecture for routers for this class of networks is given in [10]. For our purpose it is sufficient to know that the network consists of links and routers (Figure 1) and that the data injected at the routers is transported in messages to other routers via the links until these messages reach a destination where they leave the network. The routers are identified by an address, and the messages have a header containing the destination router's address. There is a protocol, called link-level flow control, that ensures lossless transmission of data from one router to the next.

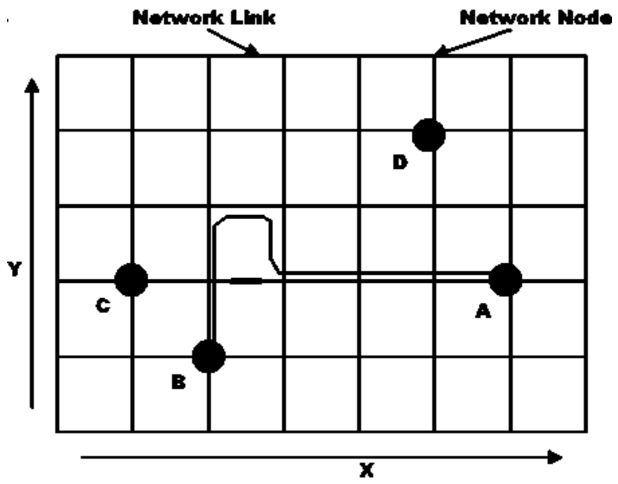

Figure 1. A mesh network, the topology for NARA.

Routing algorithms react to several distinct types of external events: the arrival of a new message, the notification by the link-level flow control of a changed load situation on an attached link or neighboring router, the notification of the failure of a network component, or the completion of the transmission of a message, which releases a resource for the next message.

Routing algorithms typically cover several more or less independent aspects. Two aspects, deadlock and livelock avoidance, ensure that a message is transported to its destination in finite time. While a deadlock results in an infinitely long waiting time of messages within the routers because their further routing depends cyclically on each other, a livelock situation occurs when one or several messages move continuously through the network without ever reaching their target. For this reason, livelock avoidance specifically includes the knowledge of the network topology, in particular the capability of the routers to determine a path to the destination for a given router address. Moreover, routing algorithms contain a local scheduling that decides which message is preferred in a resource conflict such as link usage. Two other aspects of importance are related to the avoidance of overloaded and broken routers or links, if this is possible. They are a combination of the collection and distribution process of relevant information and the application when routing an individual message.

The routing algorithm NARA (New Adaptive Routing Algorithm, [5]) is used as an example. It is intended for two-dimensional meshes as the one shown in Figure 1 where the address of a router is a coordinate pair (x, y) in an integer-addressed rectangle. To avoid livelocks, paths of minimal length are preferred, i.e. a message is routed such that it gets closer to the destination if this is possible.

```
ON in_message(indir,vc,dx,dy) -- a message has arrived
-- if all channels are in use
 IF FORALL j IN deadlock_free(indir,vc,dx,dy): out_chan(j,vc)<>free
 THEN out_set(indir,vc)<-deadlock_free(indir,vc,dx,dy);
-- if any of the minimal paths is free
 IF EXISTS i IN minimal(dx,dy): out_chan(i,vc)=free AND
    (FORALL j IN minimal(dx,dy): mean_queue(vc,i)<=mean_queue(vc,j))
 THEN !send(vc,vc,indir,i),
    out_queue(vc,i)<-message_length(vc,indir);
-- if some channels are free but not the minimal ones
 IF EXISTS i IN deadlock_free(indir,vc,dx,dy)\minimal(dx,dy): out_chan(i,vc)=free AND
    (FORALL j IN deadlock_free(indir,vc,dx,dy)\minimal(dx,dy):
       (mean_queue(vc,i)>=mean_queue(vc,j)))
 THEN !send(vc,vc,indir,i),
    out_queue(vc,i)<-message_length(vc,indir);
END in_message
```

Figure 2. RERAL implementation of the main rule base of NARA, `deadlock_free` and `minimal` are subbases.

Deadlock avoidance is based on the so-called *turn model* [14]. NARA distinguishes messages according to the difference in the Y-direction such that all messages whose destination has a smaller Y-coordinate than the source (for example a message sent from A to B) are handled separately from those with increasing Y-coordinates, e.g. C to D. For those messages with increasing Y-coordinates, the restriction is imposed that the message may not change its direction after it has used a link with increasing Y-coordinate. For the other messages, the same restriction applies symmetrically. A path of a message for this case is shown in the figure, the only downward part leads directly to the destination node. Furthermore, NARA applies an age-based local scheduling strategy that counts the events if a message loses arbitration to another message when competing for a free link. Finally, NARA contains a method for adaptivity that consists of summing up the buffer fill levels in a router and transporting this information to neighboring routers. If a message arrives, it is checked whether any of the allowed links is available with respect to deadlock avoidance.

Clearly, this algorithm is complex and its description is preferably done in a high-level language. This language should allow the use of basic data types (integers, Booleans, and arrays of them), with appropriate operations (addition, comparisons, logical operations, etc.), and, most importantly, also allows individual aspects such as deadlock-avoidance algorithm to be described separately from adaptivity or the topological properties of the network. Aspects such as variables, and the notion of events ("whenever a message arrives, do the following") are also crucial. Finally, the high performance requirements must be reflected in the language allowing a high parallelism. Most of the events can be handled concurrently, but at certain points an atomic behavior needs to be guaranteed, for instance when a shared resource is assigned, e.g. a link to a message.

Furthermore, the hardware for executing the routing algorithm has to provide the resources for storing a structured state (variables, arrays, etc.) and for performing the algorithm on arriving events accordingly. The algorithmic step involves the selection of the appropriate rules, including evaluating arithmetic expressions on the parameters (for instance the destination node address). The result is the modification of state variables, the generation of commands to the data-transport part of the router, and in some cases the generation of new events for further actions. To achieve the performance goals, we have developed an architecture for a routing engine that combines configurable, problem-specific components for arithmetic expressions in premises and conclusions of the rules. An example is the circuit for fault tolerance based on finite ordered sets [8]. The logical skeleton of the rules is mapped to a look-up table. To keep the table compact, we have developed a set of compression methods that exploit regularity and symmetry in the algorithmic structure. Some of these methods can profit from ambiguities in the algorithm; in NARA for example such an undefined situation exists for messages that stay on the same Y-coordinate level (e.g. A to C): It does not matter into which class of messages they are assigned by an implementation.

A second area for applying the hardware-based evaluation of rule bases is the management of a memory interface in a SoC [12], such as a network processor. In a structure such as that of [11], several components (network interfaces, processor cores, coprocessors, extension interfaces) share a common memory interface. Because of the high pin number required for interfacing memories, bandwidth on this interface is typically a valuable resource. Therefore, optimizing its use can help to build a cheaper system or to achieve better performance at the same cost. However, the components have very different access characteristics; compare, for instance, the sporadic memory-access pattern with a fixed line size from a data cache and the fixed pattern of a tree search engine for IP header classification. Some of the accesses have a temporal elasticity, for instance the flushing of dirty data cache lines can sometimes be done in advance, i.e. before the cache line is reused and flushing is enforced. Another example is a network interface that typically contains a buffer. Depending on the current and future pressure on the memory interface, the point in time for data transport between this on-chip memory and the off-chip memory through the interface in question can be varied. The management algorithm for this task has to take the performance goals of the application and the utilization of the various components (processors, coprocessors) into account. The reaction in a given over- or underload situation has to be translated into the reactive capabilities of the individual requester components without degrading the situation for upcoming cycles. Because of the similarity of this management problem with the routing algorithms, we believe that the concept of a rule-based language, combined with the optimizing compiler and an application-tailored rule-evaluation engine, can also be applied. Of course, intensive studies including system simulations will have to be carried out to prove this.

4 RERAL

The language itself and its usage to define routing algorithms are given in [6]. The central building block of the language is a rule that consists of a condition (premise) and a list of commands (conclusion). A set of rules forms a rule base or a subbase. In general, subbases are functions returning a value to the caller. Exploiting side effects, they become a powerful way of describing subroutines and shaping the code clearly. In contrast to traditional programming languages such as C/C++ and MODULA-2, all rules of a rule base are executed in parallel. The conditions are evaluated with respect to the global state at a rule-base call, and the commands belonging to the conclusions executed alter the global state also in parallel.

Here, we briefly state the main syntactical components of RERAL, using a routing-algorithm implementation as example.

Constant Definition

CONSTANT *LinkIndex :=* 0..5*;*

Constants are declared by finite sets of numbers, symbols or constant sets. Here, the constant *LinkIndex* consists of six numbers $\{0, 1, 2, 3, 4, 5\}$. These sets are used like types and constitute a high abstraction of hardware details.

Variable Definition

VARIABLE *Linkload(LinkIndex) IN* 0..63*;*

The variable *LinkLoad* is an array of numbers where each number is in the range of 0..63. Its size and indices are given by the constant set in parentheses. All variables have a finite (usually small) domain that is given by a set literal or a constant.

Rule-Base Declaration

DEFER SUBBASE *TorusMinimalXDim(Xdest, Ydest)*

This kind of declaration predefines a subbase that has two parameters. Subbases are one form of rule bases. They are the main structuring element and either define a function or work as subactions having side effects. The declaration is not necessary, but improves readability.

Subbase Definition

SUBBASE *opp(vc) ...* **END** *opp*

This declaration embeds the subbase. The dots replace a set of independent rules whose notion contains a high degree of parallelism. Every relevant case has to be covered by at least one rule (completeness).

Triggered Rule-Base Definition

ON *In_message(Dir , Chan) ...* **END** *In_message*

This rule base has to be executed exactly once for each occurring event bound to it. As rule bases describe only functions, time and sequence are expressed separately by the notion of events.

Rule

IF *vc=south* **THEN** *opp* $\leftarrow$ *north;*

This construct is the core idea of the rulebased language. It can be viewed as a guarded command. One rule represents one case of the algorithm. A rule base is some kind of case distinction where every rule covers at least one case. It is expressed by a predicate logic expression. In this case the action is the presentation of the function result. In addition to Boolean operators and arithmetic expressions subbases can be used.

Quantifier Expressions in Premise

EXISTS p **IN** $\{0, 1, 2, 3\}$*: OutChannelUsage(dir , p) = FREE*
FORALL p **IN** $\{0, 1, 2, 3\}$*: OutChannelUsage(dir , p) = FREE*

Both expressions are a sort of loop used in conventional languages but they avoid the sequentiality of conventional loops. The *EXISTS* expression is a short form for as many rules as the number of elements of the finite set. Here, the variable p may be reused in the conclusion. In contrast to the *EXISTS* expression, the *FORALL* expression is not a short form for multiple rules but for a sequence of *AND* expressions in the same premise. This variable cannot be reused in the conclusion but it is possible to establish the same loop in the conclusion as well, see below. Both expression introduce some kind of local name space.

Event Generation (Conclusion)

!*InternalSwitch(fromDir, fromChan, SOUTH, DetNetChan);*

Asynchronous control of the hardware is accomplished by generating an event. Events can also be used to cascade several rule interpretations to generate a final result.

Variable Assignment (Conclusion)

OutLinkUsage(ToDir) $\longleftarrow$ *OutLinkUsage(ToDir) -1*

Rule execution is atomic, i.e. if a variable is checked in the premise and changed in the conclusion, parallel execution of two rule bases has to be performed on the same system state.

FORALL-Quantifier in Conclusion

FORALL j **IN** *all_directions: !send_info(info,j,total_load)*

The conclusion can contain several commands that are executed concurrently. The same applies for "loops" which can be nested.

Overall the language eliminates as many sequential dependencies as possible. Note that the application of a subbase in a premise does not imply that the hierarchically lower subbase has to be executed before the main one. In contrast, the hierarchical structure is only a form of expression for one larger subbase that is evaluated in one piece. The subbase hierarchy allows abstraction of hardware details. The reading access of a variable cannot be distinguished syntactically from the application of a subbase. This allows the introduction of caching techniques (eliminating subbase calls) and the replacement of status arrays by methods calculating the original expected value from other sources. Hence, a higher interface can be defined whose implementation on the actual hardware can again be done using rule bases.

5 Compilation

The compilation process of RERAL programs simplifies all rule bases. There are several transformations, which can be done in an arbitrary sequence. The goal is a rule base for each event, which contains a minimal set of rules. The rules' premises and conclusions should be flat expressions (conjunctions of simple comparisons) and lists of simple commands. Only for those functions where a direct hardware implementation is available (e.g. supremum in a finite ordered set), should the corresponding identifier be found. One transformation aims at retrieving a minimal number of rule bases by replacing function calls by their subbase. Another transformation unrolls loops. They exist in premises using forall or exist quantifiers and in conclusions using a standard forall statement to loop over all elements of a finite set. Thirdly, all premises are searched for run-time-independent elements that are evaluated and the premises are shaped based on the results. Frequently the premise is reduced to a Boolean constant so that rules with false premises are dropped. All premise terms are collected in a table and replaced by a label to avoid multiple run-time computations.

The detailed processing can be approximated by the following description:

- Replacement of Constants
  The constant definitions are checked for interdependencies to detect illegal ring definitions, and ordered by them to minimize the number of replacements. At the end of this transformation, all constant values substitute their symbols. This process is especially important for subsequent premise evaluations.
- Solving Quantifiers
  All FORALL (Symbol: $\forall$) quantifiers used in premises are solved by the following equivalence:

  $\mathcal{M}$ is a finite set.
  $p : \mathcal{M} \to \mathcal{B}$ is a predicate.

  $$\boxed{\forall i \in \mathcal{M} : p(i) \quad \Longleftrightarrow \quad \bigwedge_{i \in \mathcal{M}} p(i)}$$

  The replacement of the EXISTS (Symbol: $\exists$) quantifier is more difficult. The conclusion can use the variable used by $\exists$ so that each element of $\mathcal{M}$ requires its own rule.

  $\mathcal{M}$ is a finite set.
  $p : \mathcal{M} \to \mathcal{B}$ is a predicate.
  c is a parameterized conclusion.

IF $\exists i \in \mathcal{M} : p(i)$ THEN $c(i)$	$\Longleftrightarrow$	$\forall i \in \mathcal{M}$: IF $p(i)$ THEN $c(i)$;

  These replacements often allow the reduction of rule premises at compile time. If $p(i)$ includes further quantifiers, they are solved beginning with the innermost one.
- Flattening Hierarchies
  Here, the modular structure of RERAL programs is decomposed by inserting subbases inline. The result is a distinctly grown rule base. Assuming rule base A has l rules and includes k calls of subbase B, A can grow to a size of $l^k - 1$ rules; if the subbase calls are spread over k rules, the rule base only grows to a size of $l * k - k$ rules.
  The order of inserting subbases has a high impact on the efficiency. Suppose that rule base A calls the subbases B and C and B also calls C, this can be processed in the following ways:
  - one inserts B into A, gets $\hat{A}$ as an intermediate result and inserts C into $\hat{A}$, or
  - one inserts C into B and A, gets $\hat{B}$ and $\hat{A}$, and inserts $\hat{B}$ instead of B into $\hat{A}$.

  Unfortunately, it is not possible to find an efficient way by syntactical analysis. Semantical aspects and dependencies decide this issue.
- Reductions
  Because of the increasing number of rules per rule base due to preceding steps, reduction functions are welcome. Later on, each distinct premise term requires a hardware

resource, such as a comparator, and each rule requires a table entry. Both are limited in the routing engine. Because flattening hierarchies and solving quantifiers can produce multiple copies of a single rule in the same rule base, multiple occurrences are removed. In particular, unfolded quantifiers allow reductions by evaluating comparisons of constants. Consequently, and considering that the Boolean operator $\wedge$ (*and*) appears more frequently than $\vee$ (*or*), many rules can be skipped because of fully evaluated false premises. Furthermore, arithmetic expressions are normalized by sorting variables and arranging them on one side in an inequality. These reduction steps are repeated whenever new rules are produced to avoid an explosion of the rule base.

6 Optimization

The expansion of rule bases and subbases produces an exponentially growing number of rules. Reducing and controlling all these rules is rather difficult for the compiler and in certain cases impossible because of missing run-time information considering e.g. dynamic sets. Taking into account that each rule requires chip space, the absolute number of rules is limited. Algebraic optimizations such as the evaluation of algebraic terms, e.g. comparisons, or the removal of multiple copies have yet been mentioned. Another idea is to find structures such as inline-inserted subbases or very small functions that are replaced by function calls whose hardware implementation is more space-efficient than that of the rule base. Candidates are functions working on huge sets because the required space depends on the size of these sets. When traditional methods are used to implement them, some only consume a fixed amount of space. In this case, the fixed-sized function typically outperforms the rule-based one. To define these structures, search the rule bases, and replace the occurrences by function calls, a substitution pattern is specified and each rule is transformed into a first-order logic representation and searched by a unification algorithm.

6.1 Unification

In general, unification tries to identify two symbolic expressions by replacing sub-expressions by other expressions. Assuming, for example, that f is a function symbol, a, and b are constants, and x and y are variables, the unification problem of the terms $f(a,x)$ and $f(y,b)$ is solved by the substitutions x/b and y/a, where e.g. x/b means that the right element b substitutes the left one x. The result of this example is $\{x/b, y/a\} \circ f(a,x) = \{x/b, y/a\} \circ f(y,b) = f(a,b)$. Here, applied to the language of first-order logic, the unification is a syntactic one. Baader and Snyder [3] give introduction to syntactic unification and also present Robinson's unification algorithm. It decides whether a set of terms is unifiable and, in the case of a positive decision, returns the most general unifier, i.e. a set of substitutions that constrains the functions less than all other possible unifiers do. This is often applied in automatic theorem provers.

6.2 Pattern Matching

To shape the performance of rule bases, a library could provide the programmer with performance-optimized subbases. For each library function, a substitution pattern that describes the high-level structure and function and its high-level substitute must be defined. To track these occurrences in the high-level program representation, a pattern and each rule of the rule base searched are checked by Robinson's unification algorithm. Figure 3 demonstrates an exemplary pattern. Its specification language is a mixture of first-order terms and RERAL syntax. The left-hand side defines the rule pattern searched in the rule and the right-hand side is the rule pattern that substitutes the matched rules. Identifiers beginning with a small letter represent functions with at least one argument, an initial capital letter denotes variables, and keywords are written in capital letters. The premise of the exemplary rule pattern in Figure 3 contains a minimum function whose result is used in the conclusion by all commands: "If there exists an element in the set that is smaller than or equal to all others then process it." The pattern premises resemble first-order terms; only comparison operators are predefined functions because of their high frequency. All other functions can be determined by general function symbols without any semantics known by the system. The representation of a conclusion allows two symbols: a variable and a function with arguments defined in the premise. Also a combination of the two is possible. The variable can match any sequence of commands, even an empty set. Functions must match at least one command that depends on the argument specified. In Figure 3, the function `p()` is an additional constraint for all elements that must be satisfied; `v()` is a kind of weight function. Each occurrence of a rule containing this pattern is replaced by the rule to the left by removing the innermost loop of the premise and introducing the library-function call `selectminimal`.

```
IF EXISTS A IN Set:
   [(p(A))
   AND (FORALL B IN Set:
     [(p(B))                          ==>      IF EXISTS A IN Set: [p(A)]
     AND ((v(A)) <= (v(B)))])]                 THEN c(selectminimal(Set,p(),v()));
THEN c(A);
```

Figure 3. Exemplary pattern specification.

An assumption to apply first-order unification is that the specifications of both inputs are first-order terms. Each rule and, by analogy, the pattern are transformed into a kind of prefix notation; even the rule itself is a binary function with two arguments: premise and conclusion. The transformation result of the example of Figure 3 is shown in Figure 4. As the algorithm does not match higher-order terms, functions without a determined semantics are replaced by a variable so that the transformed pattern is more general. The original meaning of the pattern is restored by a list of constraints that contains a constraint for each generalization. Another example of generalization and constraints is the multiple frequencies of the same function, in which each occurrence of a function is replaced by its own variable. All substitutes of the variables must be the same function (constraint). The constraints needed to restore the semantic of a pattern are itemized below.

- `F_PARAM_EQUAL Int [Function]`
  All functions of the list must have the same number of arguments. This constraint is used to ensure equal length of sequences assigned to variables.
- `F_ALL_ELEM [Function] Function`
  All elements of the list must be an argument of the function.
- `F_EQUAL Function Function`
  In addition to their arguments both functions must be equal.

```
(F "RULE" [
   F "PREMISE" [
     F "EXISTSQ" [ V "A", V "Set",
       F "AND" [ V "p", F "FORALLQ" [ V "B", V "Set",
                           F "AND" [ V "p0", F "LESSEQ" [V "v",V "v0"]]]]]],
F "CONCLUSION" [V "c"]],


F "RULE" [
  F "PREMISE" [
    F "EXISTSQ" [V "A",V "Set",V "p"]],
  F "CONCLUSION" [
    F "FUNCTION" [V "c", F "FUNCTION" [V "selectminimal", V "Set",
                            F "FUNCTION" [ V "p0" ], F "FUNCTION" [V "v"]]]]]
)
```

Figure 4. Transformed exemplary pattern.

- C_ALL_ELEM [Function] [Function] / C_ANY_ELEM [Function] [Function]
  All variables of the first list must be bound by all elements/at least one element of the second list.

Figure 5 shows the list of constraints for the pattern of Figure 3 and its transformation, shown in Figure 4. The transpositions are derived from the target rule and are necessary to build the target rule. They must be applied to the pattern before the most general unifier is employed.

```
SET [ F_PARAM_EQUAL 1 [V "c"],
      F_ALL_ELEM [V "A"] (V "p"),
      F_ALL_ELEM [V "B"] (V "p0"),
      F_EQUAL (V "p") (V "p0"),
      F_ALL_ELEM [V "A"] (V "v"),
      F_ALL_ELEM [V "B"] (V "v0"),
      F_EQUAL (V "v") (V "v0"),
      C_ALL_ELEM [V "A"] [V "c"]],
```

```
[TP (V "A",
     F "FUNCTION" [ V "selectminimal",
                    V "Set",
                    F "FUNCTION" [V "p0"],
                    F "FUNCTION" [V "v"]])]
```

Figure 5. Constraints (left) and transpositions (right) of the exemplary pattern.

By processing the second rule of NARA, see Figure 2, using the pattern of Figure 3, the algorithm succeeds in computing the most general unifier, see Figure 6. This is attached to the transpositions gained by transformation and then all substitutions are executed. The final result, a rule including a library-function call because of a matched pattern, is shown in Figure 7.

7 Results

The compilation process and the generated implementation of the rule-based hardware specification are usually greedy for resources. Hence, especially memory size on processing-system side, and available logic gates and timing constraints on target-system side limit number and complexity of processable rules. The usage of functions that conserve utilization of resources provides room to implement more rules. Therefore, the number of rule bases that can be performed by the routing engine increases.

```
[ TP (V "A",V "i"),
  TP (V "Set", F "FUNCTION" [ V "minimal", F "SYMB" [ V "dx" ], F "SYMB" [ V "dy" ] ]),
  TP (V "p", F "EQUAL" [ F "FUNCTION" [ V "out_chan",
                                       F "SYMB" [ V "i" ], F "SYMB" [ V "vc" ] ],
                        F "SYMB" [ V "free" ] ]),
  TP (V "B", V "j"),
  TP (V "p0", F "EQUAL" [ F "FUNCTION" [ V "out_chan",
                                       F "SYMB" [ V "j" ], F "SYMB" [ V "vc" ] ],
                        F "SYMB" [ V "free" ] ]),
  TP (V "v", F "FUNCTION" [ V "mean_queue", F "SYMB" [ V "vc" ], F "SYMB" [ V "i" ] ]),
  TP (V "v0", F "FUNCTION" [ V "mean_queue", F "SYMB" [ V "vc" ], F "SYMB" [ V "j" ] ])
]
```

Figure 6. Most general unifier of the exemplary pattern applied to the second rule of NARA.

```
IF EXISTS i IN minimal(dx, dy): out_chan(i, vc) == free
THEN !send(vc, vc, indir,
          selectminimal(minimal(dx, dy), equal(), mean_queue())),
          out_queue(vc, selectminimal(minimal(dx, dy),equal(), mean_queue()))
                 ← message_length(vc, indir);
```

Figure 7. Result of substitution for the exemplary pattern applied to the second rule of NARA.

The application of the pattern of Figure 3, which selects the minimum of a set deploying a constraint and weight function, to NARA, see Section 3, delivers some numbers that support the benefit of our approach. Our goal was the decrease in the number of rules and the shortening of the premise to shrink the tables of the routing engine. Each comparison of a premise requires an arithmetic circuit and contributes to the address length for the table access. The expansion of the second rule of NARA delivers 117 rules with 7 comparisons per premise. Making use of the unification-based optimization, only 13 rules with 3 comparisons per premise remain. Here, a reduction to a tenth of the normal number of rules and halve the number of comparisons per premise is achieved. Unfortunately, the unification-based optimization has a high mismatch ratio because of the commutativity of several functions. Because exchanged operands of a binary, commutative operation for example can lead to mismatches, the operands were sorted by length to decrease the number of mismatches.

The compiler presented breaks a hierarchically described algorithm down into few tables, one table per event-triggered rule base. The table of the example rule base of NARA requires about 1 kByte.

Moreover, the prototype of the rule-evaluation engine, implemented on XILINX 4000 FPGAs, achieves a clock frequency of 50 MHz. Assuming that a standard-cell ASIC implementation in current technology would run at about 500 MHz a custom implementation could achieve clock frequencies of state-of-the-art microprocessors. The system reaction on a single event has a very low latency (60ns) because the prototype only consumes three clock cycles per decision. Applied to routing, this value would even satisfy the requirements of a network for state-of-the-art blade servers, or high-end clusters.

8 Conclusion

Heterogeneous configurable hardware units are comparatively new, and therefore compiler construction in this area poses new challenges. The experimental compiler presented here combines many different techniques and proves that a high-level abstract language can be used to achieve a very high performance. Combined with the fast reaction to external events and the compiler-enabled high abstraction level, an execution model for problems with extremely high real-time requirements and limited hardware resources is available.

It is remarkable that this transformation system breaks a hierarchically described routing algorithm like NARA down into few tables of a small size. In addition, the reduction in size and number of rules relieves the processing system and provides new headroom for larger and more complex rule bases. Besides, if the same methods are applied to a new area, the crucial functions for the application domain have to be identified first by analyzing the target set of algorithms. This allows the direct mapping of complex subproblems to configurable hardware units. The expected types of redundancy can also be identified, which allows the selection of appropriate address-generation methods.

Since the unification-based optimization technique used is not as successful as it should be, further improvements are desirable. As semantical unification has a high complexity and computational effort other sorting functions such as any sort of weight function evaluating and combining different pattern qualities should be tested.

References

1. Carsten Albrecht. *Benutzungsmustererkennung zur Einbindung von Hardwareeinheiten in den RERAL-Compiler.* Student's Thesis, Technical Report IAB-75, Institute of Computer Engineering, University of Lübeck, Germany, 2001.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1988.
3. Franz Baader and Wayne Snyder. *Unification Theory.* Chapter 8 of Handbook of Automated Reasoning, edited by Alan Robinson and Andrei Voronkov, Elsevier Science Publishers B.V., 1999.
4. Richard Bird. *Introduction to Functional Programming using Haskell.* 2nd Edition, Prentice Hall, 1998.
5. Chris M. Cunningham and Dimiter Avresky. *Fault-Tolerant Adaptive Routing for Two-Dimensional Meshes.* In Proceedings of the First International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 1995.
6. Andreas C. Döring, Gunther Lustig, Carsten Albrecht, and Wolfgang Obelöer. *Building a Compiler for an Application-Specific Language.* 1st Scottish Functional Programming Workshop, August 29th - September 1st, 1999, Stirling, UK.
7. Andreas C. Döring, Wolfgang Obelöer, Gunther Lustig, and Erik Maehle. *A Flexible Approach for a Fault-Tolerant Router.* In Proceedings of the Workshop on Fault-Tolerant Parallel and Distributed Systems, Lecture Notes on Computer Science Vol. 1388, pp. 693-713, Springer, Berlin/Heidelberg, 1998.
8. Andreas C. Döring and Gunther Lustig. *Implementation of Finite Lattices in VLSI for Fault-State Encoding in High-Speed Networks.* In Parallel and Distributed Processing, Lecture Notes in Computer Science Vol. 1800, Springer, 2000.
9. Andreas C. Döring, Gunther Lustig. *Generating Addresses for Multi-dimensional Array Access in FPGA On-chip Memory.* Field-Programmable Logic and Applications FPL 2000, Lecture Notes in Computer Science No. 1896, 626-635, Springer-Verlag, 2000.
10. Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks – an Engineering Approach.* Morgan-Kaufmann Publishers, 2002.
11. Maria Gabrani, Gero Dittmann, Andreas Döring, Andreas Herkersdorf, Patricia Sagmeister, and Jan van Lunteren. *Design Methodology for a Modular Service-Driven Network Processor Architecture.* Computer Networks 41(5), 623 – 640, 2003.
12. Maria Gabrani, Andreas Döring, Patricia Sagmeister, Peter Buchmann and Andreas Herkersdorf *Optimizing Bandwidth Usage in a Multi-Core Chip.* Patent Application No. CH9-2003-0024. European Patent Office.
13. Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. *Eli: A Complete Flexible Compiler Construction System.* Communications of the ACM, 35(2), 121-131, 1992.
14. Christopher J. Glass and Lionel M. Ni. *The Turn Model for Adaptive Routing.* Journal of the ACM 41(5), 874 – 902, 1994.

15. Simon Helsen. *Region-based Program Specialization.* PhD Thesis, Albert-Ludwigs-Universität Freiburg, Germany, 2002.
16. Simon Peyton Jones, John Hughes et al. *Report on the Programming Language Haskell 98.* Technical Report, available at www.haskell.org, 1999.
17. Jan van Lunteren and Ton Engbersen. *Multi-Field Packet Classification Using Ternary CAM.* IEE Electronic Letters 38(1), pp. 21-23, 2002.
18. Douglas H. Summerville, José G. Delgado-Frias and Stamatis Vassiliadis. *A Flexible Bit-Pattern Associative Router for Interconnection Networks.* IEEE Transactions on Parallel and Distributed Systems, 7(5), 477–485, 1996.
19. Vikram Chopra, Ajay Desai, Ranghunath Iyer, et al. *Method and Apparatus for High-speed Network Rule Processing.* US Patent, US6510509, 2003.

Context-Free Tree languages for Descendants

Pierre Réty and Julie Vuotto

Lifo - Université d'Orléans,
B.P. 6759, 45067 Orléans cedex 2, France
{rety,vuotto}@lifo.univ-orleans.fr

Abstract The preservation of regular tree languages through rewriting, has already been studied. In this paper, we study the preservation of context-free tree languages through rewriting, for constructor-based term rewrite systems. We give positive and negative results. Positive results are effective since we give algorithms to build context-free grammars.

1 Introduction

The descendants of a set of terms E by a rewrite system R are the terms obtained by rewriting the elements of E with the rules of R. Computing descendants may be used for checking rewriting properties, like reachability, joinability,... It may also help to check safety properties for cryptographic[1] protocols [5]: the set of descendants (denoted by $R^*(E)$) expresses all the possible messages running in the net, and let $\mathcal{L}_I$ be the set of illegal messages. Intuitively, R simulates the protocol steps and the intruder actions; and illegal messages simulate intrusions. The protocol is safe (in a certain sense) iff $R^*(E) \cap \mathcal{L}_I = \emptyset$. Protocol verification has also been investigated using other techniques: [1,8,7,11].

In general, the set of descendants $R^*(E)$ is infinite. An easy way to express and handle such infinite sets is to use finite tree automata, i.e. regular tree languages [3,10,18,17,12]. However, it is undecidable whether a given rewrite system preserves regularity (also called recognizability) or not [6], and all those papers define subclasses. On the other hand, the possibility of computing a regular superset of the (possibly non-regular) set of descendants, assuming weaker restrictions, has been investigated in [4,5]. Computing only descendants obtained by rewriting respecting some strategies has also been studied [14,13].

In [12], certain restrictions are assumed in order to make $R^*(E)$ regular. R is assumed to be constructor-based, as well as:

1. Right-hand-sides of rewrite rules are linear.
2. No nested defined-functions in rhs's.
3. Innermost[2] function calls in rhs's are shallow subterms[3].
4. E is a particular regular language: E is the set of constructor-instances of a fixed linear term t (or more generally t is instantiated by arbitrary regular languages of constructor-terms).

Moreover, some counter-examples show that if anyone among the above four restrictions is not satisfied, then $R^*(E)$ is not regular. Consequently, the following question arises: does $R^*(E)$ belong to the previous class in Chomsky's hierarchy, i.e. is it a context-free tree language? If it is, this can still be used for protocol verification because $R^*(E) \cap \mathcal{L}_I = \emptyset$ is decidable, provided $\mathcal{L}_I$ is still regular. This paper attempts to answer this question.

[1] For more information on cryptographic protocol, the reader can refer to [16].

[2] Innermost is useless when Restriction 2 is satisfied. Without "innermost", Restriction 3 would imply Restriction 2, and we want them to be independent.

[3] I.e. $\forall l \to r \in R$, $\forall p \in Pos(r)$, ($r(p)$ is an innermost defined-function in $r \implies r|_p = f(r_1, \ldots, r_n)$) where $r_1, \ldots, r_n$ are variables or ground constructor-terms.

Before giving the results, some more notions need to be introduced. We say that a tree language is *NT-bounded* if it is generated by a NT-bounded grammar, which means that the number of nested non-terminals in trees generated by the grammar is bounded. Note that regular languages are NT-bounded. A context-free tree language is said *top-context-free*[4] if it is generated by a top-context-free grammar, i.e. whose production right-hand-sides contain only one non-terminal, located on top. Note that top-context-free languages are context-free and NT-bounded.

We also define some other restrictions:

5. Left-hand-sides of rewrite rules are linear.
6. Recursive[5] rewrite rules are not consuming, i.e. their left-hand-sides are of the form $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are variables.
7. E is the set of instances of a fixed linear term t, by arbitrary NT-bounded context-free languages of constructor-terms.
8. E is the set of instances of a fixed **non-linear** term t that contains only one defined-function occurring on top, by instantiating the linear variables of t by arbitrary NT-bounded context-free languages of constructor-terms, and by instantiating the non-linear variables of t by arbitrary top-context-free languages of constructor-terms.

Still assuming that R is constructor-based, the results are:

- For all $i \in \{1, 2, 3\}$, if restrictions 1, 2, 3, 4 are satisfied except Restriction i then $R^*(E)$ is not context-free in the general case.
- If Restrictions 1, 2, 5, 6 and (7 or 8) are satisfied, then $R^*(E)$ is context-free and NT-bounded.
- If Restrictions 1, 2, 6, 7 are assumed (i.e. Restriction 5 is not assumed anymore), and even if Restriction 3 is assumed in addition, then $R^*(E)$ is not context-free in the general case.
- If Restrictions 1, 2, 5 and (7 or 8) are assumed (i.e. Restriction 6 is not assumed anymore), and even if Restriction 3 is assumed in addition, then $R^*(E)$ is not context-free in the general case.

The positive results are obtained by building a context-free grammar that generates $R^*(E)$.

2 Preliminaries

2.1 Terms and Positions

We denote respectively by $\mathcal{C}$, $\mathcal{F}$, $\mathcal{X}$ the sets of *constructors, defined-functions* and *variables.* For $s \in \mathcal{C} \cup \mathcal{F}$, $ar(s)$ denotes the arity of s. In the following, we write $f \in \mathcal{F}^{\backslash n}$ for $f \in \mathcal{F}$ and $ar(f) = n$. A term is *linear* if no variable occurs more than once. A *ground term* is a term that does not contain variables. $T(\Sigma, \mathcal{X})$ is the set of terms defined on the signature $\Sigma = \mathcal{F} \cup \mathcal{C}$, T_Σ is the set of ground terms, and $T_\mathcal{C}$ is the set of ground *constructor-terms* (terms that contain only constructors). Let t, t' be terms. We denote by $Var(t)$ the set of variables of t, by $Pos(t)$ the set of positions of t, by $PosF(t)$ the set of positions of defined-functions of t, and by $\overline{Pos}(t)$ the set of positions of non-variable symbols of t. A position p is a list of integers whose length is denoted by $|p|$. For positions p, p', $p \geq p'$ means that p

[4] Initially introduced by A. Arnold and M. Dauchet [2], and called *co-regular*. It is called *top-context-free* in [9].

[5] $l \to r$ is recursive means that: $l \to_R^* \alpha$ where there exists a position p of α s.t. $\alpha(p) = l(\epsilon)$.

is located below p', i.e. $p = p'.v$ for some position v, whereas $p \| p'$ means that p and p' are incomparable, i.e. $\neg(p \geq p') \wedge \neg(p' \geq p)$. Let $p \in \overline{Pos}(t)$, $t|_p$ is the subterm of t at position p, and $t(p)$ is the top symbol of $t|_p$. $t[t']_p = t[p \leftarrow t']$ is the term obtained from t by replacing the subterm at position p by t'. A term $C \in T(\Sigma, \mathcal{X})$ s.t. $Var(C) = \{x_1, \ldots, x_n\}$ is called a *context*, and $C[t_1, \ldots, t_n]$ denotes the term obtained from C by replacing each x_i by the term t_i.

2.2 Rewrite Systems and Descendants

A *term rewrite system (TRS)* is a pair (Σ, R) where R is a finite set of rewrite rules $l \to r$ where $l, r \in T(\Sigma, \mathcal{X})$ and $Var(r) \subseteq Var(l)$. In the following of the paper we write only R for a rewrite system. *lhs* stands for left-hand-side, *rhs* for right-hand-side. R is *constructor-based* if every lhs l of R is of the form $l = f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$ and $t_1, \ldots, t_n$ contain only constructors and variables. The rewrite relation $\to_R$ is defined as follows: $t \to_R t'$ (or, $t \to_{[p, l \to r]} t'$) if there exists $p \in \overline{Pos}(t)$, a rule $\langle \to r$ and a substitution σ s.t. $t|_p = l\sigma$ and $t' = t[r\sigma]_p$. $\to_R^*$ denotes the reflexive-transitive closure of $\to_R$. t' is a *descendant* of t if $t \to_R^* t'$. If E is a set of terms, $R^*(E)$ denotes the set of descendants of elements of E. t is *irreducible* if $\neg(\exists t' \mid t \to_R t')$. The set of irreducible ground terms is denoted by $IRR(R)$. t' is a *normal-form* of t if $t \to_R^* t'$ and t' is irreducible. The set of normal-forms of elements of E is denoted by $R^!(E)$. Thus, $R^!(E) = R^*(E) \cap IRR(R)$.

2.3 Recursive and Consuming Rewrite Rules

Definition 1. Let R be a TRS. Let $f,\ g \in \mathcal{F}$. The relation $>$ on $\mathcal{F}$ is defined as follows:

$$f > g \iff \exists l \to r \in R,\ l(\epsilon) = f \wedge \exists p \in PosF(r),\ r(p) = g$$

Definition 2. let R be a TRS and $l \to r \in R$. $l \to r$ is *recursive* in R if

$$\exists g \in \mathcal{F}\ (g \in F(r)\ \wedge\ g >^* l(\epsilon))$$

where $>^*$ is the reflexive-transitive closure of $>$, and $F(r)$ are the defined-functions appearing in r.

Lemma 1. *Let R be a TRS and $l \to r,\ l_1 \to r_1, \ldots,\ l_i \to r_i \in R$. Let t be a term s.t. $PosF(t) = \{\epsilon\}$.*

$$\textit{If } t \to_{l \to r} t_1 \to_{l_1 \to r_1} t_2 \ldots \to_{l_n \to r_n} t_{n+1} \textit{ and } l \to r \textit{ is not recursive, then}$$
$$\forall i,\ l_i \to r_i \neq l \to r \textit{ (i.e } l \to r \textit{ can be used only once).}$$

Proof. $l \to r$ being non-recursive, by applying rules on functions of r, we can never generate the symbol $l(\epsilon)$. So, $\forall i,\ \forall p_i \in posF(t_i),\ t_i(p_i) \neq l(\epsilon)$.

Definition 3. Let R be a TRS and $l \to r \in R$.
$l \to r$ is *not consuming* if l is of the form $f(x_1, \ldots, x_n)$, where $f \in \mathcal{F}^{\backslash n}$ and $x_1, \ldots, x_n$ are variables.

2.4 Tree Grammars

A *tree grammar* $\mathcal{G}$ is a quadruple $(\mathcal{N}, \Sigma, S, P)$ where $\mathcal{N}$ is a finite set of non-terminals, Σ a signature, $S \in \mathcal{N}$ the axiom, and P a finite set of production rules. In the regular case, the production rules are of the form: $A \to t$ where $A \in \mathcal{N}$ and $t \in T(\Sigma \cup \mathcal{N})$ and in the context-free case: $A(x_1, \ldots, x_n) \to t$ where A is a n-ary non-terminal, $x_1, \ldots, x_n$ are distinct variables and $t \in T(\Sigma \cup \mathcal{N} \cup \{x_1, \ldots, x_n\})$[6]. Note that there are only 0-ary

[6] Note that t is not necessarily linear.

non-terminals in the regular case. The language $\mathcal{L}(\mathcal{G})$ generated by $\mathcal{G}$ is the set of terms derivable from S. A set of terms E is a regular (resp. context-free) language if there is a regular (resp. context-free) grammar $\mathcal{G}$ such that $E = \mathcal{L}(\mathcal{G})$. Grammar 1.a below generates the regular tree language $\{f(s^*(a))\}$ while Grammar 1.b generates the context-free tree language $\{f(s^n(a), s^n(b))\}$.

Example 1.

a. $S \to f(B)$
$B \to s(B) \mid a$

b. $S \to F(a, b)$
$F(x, y) \to F(s(x), s(y)) \mid f(x, y)$

Figure 1 recalls known properties.

	L_1	L_2	
$L_1 \cup L_2$	reg.	reg.	reg.
	c-free	c-free	c-free
$L_1 \cap L_2$	reg.	reg.	reg.
	reg.	c-free	c-free
	c-free	c-free	~~c-free~~
$\overline{L_1}$	reg.		reg.
$t \in L_1$	reg. or c-free		decidable
$L_1 = \emptyset$	reg. or c-free		decidable

Figure 1. Properties

In a grammar, the notion of recursive production is defined in the same way as for rewrite rules.

Definition 4. A context-free tree grammar is said *NT-bounded* if the transitions whose rhs's contain nested non-terminals, are not recursive.

Lemma 2. *Let $\mathcal{G}$ be a NT-bounded context-free tree grammar. The number of nested non-terminals in trees generated by $\mathcal{G}$ is bounded.*

Recall that top-context-free languages are defined in the introduction.

Lemma 3. *[2] Let $L = \{f(t, t) \mid t \in L', f \in \Sigma^{\backslash 2}\}$. L is context-free iff L' is top-context-free. And in this case, L is also top-context-free.*

Lemma 4. *[9] Let Σ be a signature. T_Σ is top-context-free iff Σ contains only symbols of arity at most one or no constant symbol.*

3 Negative Results

The negative results come from the counter-examples below, and the following remark: if R is left-linear, the set of irreducible terms $IRR(R)$ is regular. Therefore, if $R^*(E)$ is context-free, then $R^!(E) = R^*(E) \cap IRR(R)$ should also be context-free. More generally, if $R^*(E)$ is context-free and T is regular, then $R^*(E) \cap T$ should also be context-free.

Counter-example 1 Restrictions 1 to 4 are satisfied, except Restriction 1.
Let $C = \{c^{\backslash 2},\ a^{\backslash 0}\}$ and $T_\mathcal{C}$ be the set of terms on $\mathcal{C}$.
Let $R = \{f(x) \to c(x, x)\}$ and $E = \{f(s) \mid s \in T_\mathcal{C}\}$. Then, $R^!(E) = \{c(s, s) \mid s \in T_\mathcal{C}\}$.
From Lemmas 3 and 4, we can conclude that $R^!(E)$ is not context-free.

Counter-example 2 Restrictions 1 to 4 are satisfied, except Restriction 3.

$$R = \{ f(x) \to p(f(s(x))),\ f(x) \to g(x, a),\ g(s(x), y) \to s(g(x, s(y))) \},\ E = \{ f(a) \}. \text{ Then, } R^!(E) = \{ p^n(s^n(g(a, s^n(a)))) \mid n \in \mathbb{N} \}.$$

It is not context-free because generating the right-hand branch $p^n(s^n(g(s^n(a))))$ needs to count n three times.

Remark 1. If Restrictions 1 to 4 are satisfied, except Restriction 2, we can simulate the previous counter-example by using the same starting language and the following TRS :

$$R = \{ f(x) \to p(f(h(x))),\ f(x) \to g(x, a),\ g(s(x), y) \to s(g(x, h(y))),\ h(x) \to s(x) \}$$

Counter-example 3 Restrictions 1, 2, 3, 6, 7 are assumed (Restriction 5 is not assumed). Let $R = \{ f(c(x, x)) \to x \}$ and $t = f(y)$. Let $L = \{ c(s^*(p^n(r^k(a))), s^n(p^k(r^*(a)))) \mid n, k \in \mathbb{N} \}$.

Let us consider $E = t[y \leftarrow L]$. The tree language L is recognized by the following NT-bounded context-free grammar :

$S \to F(a, a)$

$F(x, y) \to F(x, r(y)) \mid F_1(x, y) \qquad F_1(x, y) \to F_1(r(x), p(y)) \mid F_2(x, y)$

$F_2(x, y) \to F_2(p(x), s(y)) \mid F_3(x, y)\ F_3(x, y) \to F_3(s(x), y) \mid c(x, y)$.

Let $T_{\mathcal{C}}$ be the set of terms on $\mathcal{C} = \{c, s, p, r, a\}$. Obviously, $T_{\mathcal{C}}$ is regular.

Hence, $R^*(E)$ is not context-free since $R^*(E) \cap T_C = \{ s^n(p^n(r^n(a))) \mid n \in \mathbb{N} \}$ is not context-free.

Counter-example 4 Restrictions 1, 2, 3, 5 and 7 are assumed (Restriction 6 is not assumed). Let :

$$R = \{ i(c(x, y, z)) \xrightarrow{r_0} f(x, y, z),\ f(s(x), y, z) \xrightarrow{r_1} s(f(x, y, z)),\ f(a, y, z) \xrightarrow{r_1'} g(a, y, z),$$

$$g(x, p(y), z) \xrightarrow{r_2} p(g(x, y, z)),\ g(x, a, z) \xrightarrow{r_2'} h(x, a, z),\ h(x, y, r(z)) \xrightarrow{r_3} r(h(x, y, z)) \}$$

Let us consider $E = \{ \begin{array}{ccc} & i & \\ & | & \\ & c & \\ / & | & \backslash \\ s^n & p^n & r^n \\ | & | & | \\ a & a & a \end{array} \mid n \in \mathbb{N}\}$. Obviously, $R^!(E) = \{ \begin{array}{ccc} & s^n & \\ & | & \\ & p^n & \\ & | & \\ & r^n & \\ & | & \\ & h & \\ / & | & \backslash \\ a & a & a \end{array} \mid n \in \mathbb{N}\}$ is not context-free. Hence, $R^*(E)$ is not context-free.

Remark 2. If Restrictions 1, 2, 3, 5 and 8 are assumed, we add the rule $j(x, y) \to i(x)$ to the previous counter-example and consider

$$E = \{ \begin{array}{cc} \multicolumn{2}{c}{j} \\ / & \backslash \\ x & x \end{array} \theta \mid \theta : x \leftarrow \{ \begin{array}{ccc} & c & \\ / & | & \backslash \\ s^n & p^n & r^n \\ | & | & | \\ a & a & a \end{array} \mid n \in \mathbb{N}\}\}$$

4 Positive Results

Let R be a constructor-based TRS satisfying Restrictions 1, 2, 5, 6, and let E satisfying Restrictions 7 or 8 be the starting tree language, generated by the NT-bounded context-free grammar $\mathcal{G}_E = (\mathcal{N}_{\mathcal{G}_E}, \Sigma, S_{\mathcal{G}_E}, P_{\mathcal{G}_E})$.

In this section, we give some definitions needed to compute $R^*(E)$. Recall that we want to construct a tree grammar that generates $R^*(E)$. We proceed step by step:
Suppose that $\{p_1, \ldots, p_n\}$ are outermost defined-functrions positions of E. Being incomparable positions, they can be treated independantly. Finding $R^*(E)$ consists in computing descendants of elements of E at positions $\geq p_i$, $\forall i \in \{1 \ldots n\}$ (so, we introduce $R^*_{\geq p_i}(E)$). We use the notation $\geq p_i$ in the case of there is at least one defined-function position below p_i. If it is not the case, it is sufficient to compute $R^*_{p_i}(E)$ (terms obtained by rewriting E at position p_i in any steps, i.e. at position p_i plus possibly at defined-functions positions issued of rewriting steps) since there is no nested defined-functions in rhs's.

4.1 Definitions

We first need to introduce how to construct a grammar that generates E in which we substitute some sub-languages $L_i = E|_{p_i}$ by other languages L'_i.

Definition 5. Let t be a fixed linear term, and let $Var(t) = \{x_1, \ldots, x_k\}$. Let $L_1, \ldots,\ L_k$ be context-free tree languages generated by the grammars $\mathcal{G}_{L_1}, \ldots,\ \mathcal{G}_{L_k}$. Let $p_1, \ldots,\ p_n \in \overline{Pos}(t)$ s.t. $\forall i,\ j\ \ (i \neq j \Rightarrow p_i || p_j)$. Let $L'_1, \ldots,\ L'_n$ be context-free tree languages generated by the grammars $\mathcal{G}_{L'_1},\ \ldots,\ \mathcal{G}_{L'_n}$.

The language of instances of t, whose sub-terms at positions $p_1, \ldots, p_n$ have been replaced, is defined as follows:

$$E[p_1 \leftarrow L'_1] \ldots [p_n \leftarrow L'_n] = \{(t\sigma)[p_1 \leftarrow s_1] \ldots [p_n \leftarrow s_n] \mid \forall i,\ x_i\sigma \in L_i\ \wedge\ \forall j,\ s_j \in L'_j\}$$

By renaming, if necessary, the non-terminals, we can suppose that the set of non-terminals of $\mathcal{G}_{L_1}, \ldots,\ \mathcal{G}_{L_k}, \mathcal{G}_{L'_1},\ \ldots,\ \mathcal{G}_{L'_n}$ are disjoint.
We denote by $S_{\mathcal{G}_{L_i}}$ (resp. $P_{\mathcal{G}_{L_i}}$) the axiom (resp. the set of productions) of $\mathcal{G}_{L_i}$.
The context-free tree grammar $\mathcal{G}_{E[p_1 \leftarrow L'_1] \ldots [p_n \leftarrow L'_n]}$ that generates $E[p_1 \leftarrow L'_1] \ldots [p_n \leftarrow L'_n]$,

has the axiom $S_{\mathcal{G}_{E[p_1 \leftarrow L'_1]\dots[p_n \leftarrow L'_n]}} = T_\epsilon$, and the set of productions:

$$\begin{aligned} P_{\mathcal{G}_{E[p_1 \leftarrow L'_1]\dots[p_n \leftarrow L'_n]}} = & P_{\mathcal{G}_{L_1}} \cup \dots \cup P_{\mathcal{G}_{L_k}} \cup P_{\mathcal{G}_{L'_1}} \cup \dots \cup P_{\mathcal{G}_{L'_n}} \\ & \{T_q \to t(q)(T_{q.1}, \dots, T_{q.ar(t(q))}) \mid q \in \overline{Pos}(t) \ \wedge \ \forall i, \ \neg(q \geq p_i)\} \\ & \{T_{p_i} \to S_{\mathcal{G}_{L'_i}} \mid i \in \{1, \dots, n\}\} \\ & \{T_q \to S_{\mathcal{G}_{L_j}} \mid q \in Pos(t) \ \wedge \ t(q) = x_j \ \wedge \ \forall i, \neg(q \geq p_i)\} \end{aligned}$$

Remark 3. - The particular case $n = 0$ defines the language E of the instances of t by elements of $L_1, \dots, L_k$.

- If $\mathcal{G}_{L_1}, \dots, \mathcal{G}_{L_k}, \mathcal{G}_{L'_1}, \dots, \mathcal{G}_{L'_n}$ are NT-bounded, $\mathcal{G}_{E[p_1 \leftarrow L'_1]\dots[p_n \leftarrow L'_n]}$ is also NT-bounded.

Definition 6. Let $p \in \overline{Pos}(t)$, then we define $E|_p = \{s|_p \mid s \in E\}$.
We obtain a grammar that generates $E|_p$ by replacing the axiom of $\mathcal{G}_E$ by T_p.

Now, we consider the descendants obtained by rewriting below or at position p ($R^*_{\geq p}$), and the descendants obtained by rewriting starting from position p (R^*_p).

Definition 7. Let L be a language and p be a position. We define:

$$R^*_{\geq p}(L) = \{s' \mid \exists s \in L, \ s \to^*_{[u_1, \dots, \ u_n]} s' \ \wedge \ \forall i, \ u_i \geq p\}.$$

Definition 8. $t \to^+_{[p, rhs's]} t'$ means that t' is obtained by rewriting t at position p, plus possibly at positions coming from the rhs's.
Formally, there exist some intermediate terms $t_1, \dots, t_n$ and some sets of positions $P(t), P(t_1), \dots, P(t_n)$ s.t.

$$t = t_0 \to_{[p_0, l_0 \to r_0]} t_1 \to_{[p_1, l_1 \to r_1]} \dots \to_{[p_{n-1}, l_{n-1} \to r_{n-1}]} t_n \to_{[p_n, l_n \to r_n]} t_{n+1} = t'$$

where

- $p_0 = p$ and $P(t) = \{p\}$,
- $\forall j, \ p_j \in P(t_j)$,
- $\forall j, \ P(t_{j+1}) = P(t_j) \backslash \{p' \mid p' \geq p_j\} \cup \{p_j.w \mid w \in PosF(r_j)\}$.

Remark 4. $P(t_j)$ contains only defined-function positions. Since there are no nested defined-functions in rhs's, $p, p' \in P(t_j)$ implies $p \| p'$.

Definition 9. Given a language L and a position p, we define $R^*_p(L)$ as follows

$$R^*_p(L) = L \cup \{t' \mid \exists t \in L, t \to^+_{[p, rhs's]} t'\}$$

Example 2. Let $R = \{f(x) \to s(x), \ g(x) \to s(h(x)), \ h(x) \to p(f(x))\}$ where $F = \{f, \, g, \, h\}$ and $C = \{s, \, a\}$. The symbols(s) that are eligible for rewriting, are underlined:

$$R^*_1(\{f(\underline{h}(g(a)))\}) = \{f(\underline{h}(g(a)))\} \cup \{f(p(\underline{f}(g(a))))\} \cup \{f(p(s(g(a))))\}$$

$Succ_t(p)$ are the nearest defined-function positions of t located below p. Formally:

Definition 10. Let $p \in Pos(t)$.

$$Succ_t(p) = \{p' \in PosF(t) \mid p' > p \wedge \forall q \in Pos(t) \ (p < q < p' \Rightarrow q \notin PosF(t))\}$$

4.2 Computing $R^*(E)$

The following theorem and lemma give a recursive algorithm (using $R^*_{\geq p}$) to build a NT-bounded context-free grammar that generates $R^*(E)$, from NT-bounded context-free grammars that generate R^*_p.
The proofs are given in [15].

Theorem 1. *Let t be a linear term, and E be the language of instances of t.*

$$\begin{array}{ll} R^*(E) = R^*_{\geq\epsilon}(E) & \textit{if } t(\epsilon) \in \mathcal{F} \\ \qquad E[p_1 \leftarrow R^*_{\geq p_1}(E)|_{p_1}] \dots [p_n \leftarrow R^*_{\geq p_n}(E)|_{p_n}] & \textit{otherwise} \\ & \textit{with } Succ_t(\epsilon) = \{p_1, \dots,\ p_n\}. \end{array}$$

Lemma 5. *Let t be a linear term, and E be the language of instances of t. Let $p \in PosF(t)$.*

$$\begin{array}{ll} R^*_{\geq p}(E) = R^*_p(E[p_1 \leftarrow R^*_{\geq p_1}(E)|_{p_1}] \dots [p_n \leftarrow R^*_{\geq p_n}(E)|_{p_n}]) & \textit{if } Succ_t(p) = \{p_1, \dots,\ p_n\} \\ \qquad R^*_p(E) & \textit{if } Succ_t(p) = \emptyset \end{array}$$

The following lemma gives an algorithm to build a NT-bounded context-free grammar that generates R^*_p, from a NT-bounded context-free grammar that generates R^*_ϵ.

Lemma 6. *Let $L'_1, \dots, L'_n$ be context-free tree languages.*

- *If $Succ_t(p) = \{p_1, \dots,\ p_n\} \neq \emptyset$, then*
  $$R^*_p(E[p_1 \leftarrow L'_1] \dots [p_n \leftarrow L'_n]) = E[p \leftarrow R^*_\epsilon(E|_p[p_1 - p \leftarrow L'_1] \dots [p_n - p \leftarrow L'_n])]$$
- *If $Succ_t(p) = \emptyset$, then $R^*_p(E) = E[p \leftarrow R^*_\epsilon(E|_p)]$*

Remark 5. If t contains only one defined-function, occurring at the root, we have $R^*(E) = R^*_\epsilon(E)$. Moreover, we show in Section 4.3 that we can compute $R^*_\epsilon(E)$ provided E is a NT-bounded context-free language. Consequently, the second positive result (assuming t is not linear) comes from the following lemma.

Lemma 7. *Let t be a term. Let E be the set of instances of t s.t. :*

- *The linear variables of t are instantiated by NT-bounded context-free tree languages, and*
- *the non-linear ones are instantiated by top-context-free tree languages.*

Then E is a NT-bounded context-free tree language.

4.3 Computing R^*_ϵ

The Maximal Depth of consumable symbols : $Depth(R)$

Thanks to Restriction 6, in a term t s.t. $PosF(t) = \{\epsilon\}$, constructors occurring below a certain depth are not used when rewriting t in several steps : they are not consumable by rewriting.

Example 3. Let $R = \{\, h(s(x)) \overset{ru_1}{\to} x,\ g(p(x)) \overset{ru_2}{\to} h(x) \,\}$.

When rewriting $g(p(s(a)))$ by ru_2 and ru_1, p, s are consumed, whereas a is not consumed.

$Depth(R)$ as defined below is actually the maximal depth of consumable symbols (see the completeness proof in [15]).

Definition 11. Let R be a TRS satisfying restriction 6, and $l \to r \in R$. We define $Depth(l) = Max(\{|p| \mid p \in \overline{pos}(l)\})$ and

$$Depth(R) = \sum_{l \to r \in R} Depth(l)$$

Remark 6. Only consuming rules matter when computing $Depth(R)$, because if $l \to r$ is not consuming, then $Depth(l) = 0$.

Explanations and Example

Recall that R be a constructor-based TRS satisfying Restrictions 1, 2, 5, 6, and let E satisfying Restrictions 7 or 8 be the starting tree language, generated by the NT-bounded context-free grammar $\mathcal{G}_E = (\mathcal{N}_{\mathcal{G}_E}, \Sigma, S_{\mathcal{G}_E}, P_{\mathcal{G}_E})$. Let $p \in PosF(t)$ and $L = E|_p$. The algorithm for building a context-free grammar $\mathcal{G}_{R^*_\epsilon L} = (\mathcal{N}_{\mathcal{G}_{R^*_\epsilon L}}, \Sigma, S_{\mathcal{G}_{R^*_\epsilon L}}, P_{\mathcal{G}_{R^*_\epsilon L}})$ generating $R^*_\epsilon(L)$ starts with the productions of $\mathcal{G}_L$, and adds new productions with 0-ary non-terminals of the form A_t, where t is a tree that may contain non-terminals of $\mathcal{G}_L$, s.t. $t(\epsilon) \in \mathcal{F}$ and $\mathcal{L}(A_t) = R^*(\{t' \mid t \to^*_{\mathcal{G}_L} t'\})$. The algorithm also uses n-ary non-terminals of the form A_t, and in this case t contains in addition constants $\perp_1, \dots, \perp_n$, and productions are added s.t. ($t_1, \dots, t_n$ are arbitrary trees that may contain non-terminals of $\mathcal{G}_L$) $\mathcal{L}(A_t(t_1, \dots, t_n)) = R^*(\{t' \mid t[\dots, \perp_i \leftarrow t_i, \dots] \to^*_{\mathcal{G}_L} t'\})$.

The role of A_t's is as follows. Since $\mathcal{G}_{R^*_\epsilon L}$ has to be context-free, we cannot create a production like $A_t(t_1, \dots, t_n) \to t'$ to simulate a rewrite step issued from the term $t[\dots, \perp_i \leftarrow t_i, \dots]$ (only $A_t(x_1, \dots, x_n) \to t'$ is allowed). However, $Depth(R)$ gives the depth of symbols that may be needed when rewriting. Then, when we deal with A_t, we first focus on the depth of symbols in t. When a symbol is deeper than $Depth(R)$, we get rid of the corresponding sub-term by moving it into arguments of A_t. To define the depth of a symbol, we must not take into account the non-terminals that occur above it, since a non-terminal is replaced by nothing if it is derived by a collapsing production.

Example 4. Let R be the TRS given by the following set of rewrite rules:

$$\{\, g(x, y) \to c(h(x), f(y)),\ f(x) \to s(f(s(x))),\ h(s(x)) \to a \,\}$$

Then, $Depth(R) = 1$. Obviously, R statisfies restrictions 1, 2, 5, 6.
Let a term $t = g(x, a)$. Obviously, t is linear. And let

$$E = \{\, g(s^n(a), a) \mid n \in \mathbb{N}\} = \{t\sigma \mid x\sigma = \{s^n(a) \mid n \in \mathbb{N}\}\}$$

generated by the set of production rules:

$$P_{\mathcal{G}_E} = \{S \to g(B, a),\ B \to a \mid s(B)\}.$$

The language $\{s^n(a) \mid n \in \mathbb{N}\}$ that instanciates the linear term t is a context-free tree language. Then, E satisfies restrictions 7.

Here, E contains only one defined-function. So, $R^*_\epsilon(E) = R^*(E)$. Then, in the following, we use $R^*(E)$ instead of $R^*_\epsilon(E)$.
Let us denote by $S_{\mathcal{G}_{R^*_\epsilon L}}$ the axiom of $\mathcal{G}_{R^*_\epsilon L}$: we add $S_{\mathcal{G}_{R^*_\epsilon L}} \to A_{g(B,a)}$ since $S_{\mathcal{G}_L}$ is the axiom of the starting grammar and $S_{\mathcal{G}_L} \to g(B,a)$. No symbol is deeper than 1 in $g(B,a)$, then we try to rewrite $g(B,a)$ and derive B. We add the productions:

$$A_{g(B,a)} \to c(A_{h(B)}, A_{f(a)}) \mid A_{g(a,a)} \mid A_{g(s(B),a)}.$$

Four new non-terminals have appeared. We have to deal with them. As previously, we try to rewrite and derive $h(B)$, $f(a)$ and $g(a,a)$. We add the productions:

$$A_{h(B)} \to A_{h(s(B))} \mid A_{h(a)},\ A_{f(a)} \to s(A_{f(s(a))}) \text{ and } A_{g(a,a)} \to c(A_{h(a)}, A_{f(a)})$$

Let us remark that we do not rewrite and derive $(g(s(B),a))$. Indeed, it contains some symbol (here B) that are deeper than $Depth(R)$. We get rid of it by adding the production:

$$A_{g(s(B),a)} \to A_{g(s(\perp),a)}(B)$$

Four new non-terminals have appeared. $h(a)$ cannot be rewritten nor derived.
$g(s(\perp),a)$ can be rewritten. We add the production:

$$A_{g(s(\perp),a)}(x) \to c(A_{h(s(\perp))}(x), A_{f(a)})$$

We have to deal with $A_{h(s(B))}$ and $A_{f(s(a))}$.
We add the productions:

$$A_{h(s(B))} \to A_{h(s(\perp))}(B) \text{ and } A_{f(s(a))} \to A_{f(s(\perp))}(a)$$

Two new non-terminals have appeared. $h(s(\perp))$ and $f(s(\perp))$ can be rewritten. We add the productions:

$$A_{h(s(\perp))}(x) \to a \text{ and } A_{f(s(\perp))}(x) \to sA_{f(s(s(\perp)))}(x).$$

One new non-terminal have appeared. We have to deal with $A_{f(s(s(\perp)))}$ We add the production:

$$A_{f(s(s(\perp)))}(x) \to A_{f(s(\perp))}(s(x)).$$

Moreover, for each non-terminal A_t where $t = C[\perp_1, \ldots, \perp_n]$, we add the production

$$A_t(x_1, \ldots, x_n) \to C[x_1, \ldots, x_n].$$

No more new non-terminals. The algorithm stops. The reader can checks that the resulting grammar really generates:

$$\{\ g(s^n(a), a),\ c(h(s^n(a)), s^p(f(s^p(a)))),\ c(a, s^p(f(s^p(a))))\ \mid n,p \in \mathbb{N}\}$$

5 Conclusion

In this paper, we have studied the behaviour of descendants of a set of terms over a rewriting system. Since obtaining regular descendants has already been investigated in the past, we have studied languages of higher level in the Chomsky hierarchy (so more expressive languages) : Context-free tree languages.

We have seen that the set of descendants of a language depends on the form of the rewriting rules and the starting language too. So, some restrictions are necessary.

We have shown by an example how to construct a grammar that generates the set of descendants of elements of a set E. The reader can find a general algorithm that permits to construct a grammar that generates the set $R^*(E)$ in the full version of the paper [15].

References

1. M. Abadi. Security protocols and specifications. In *Second International Conference, FOSSACS'99*, volume 1578 of *LNCS*, pages 1–13. Springer-Verlag, March 1999.
2. A. Arnold and M. Dauchet. Un théorème de duplications pour les forêts algébriques. In *Journal of Computer and System Sciences*, pages 13:223–244, 1976.
3. M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Fifth Annual IEEE Symposium on Logic Computer Science*, pages 242–248. IEEE Computer Society Press, Philadelphia, Pennsylvania, 1990.
4. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings of 9th Conference RTA, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998.
5. T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *Proceedings 17th International CADE, Pittsburgh (Pen., USA)*, volume 1831 of *LNAI*. Springer-Verlag, 2000. (extended version in Technical Report RR-3921, Inria 2000).
6. R. Gilleron. Decision problems for term rewrite systems and recognizable tree languages. In *STACS*, volume 480 of *LNCS*, pages 148–159. Springer-Verlag, 1991.
7. J. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In Beverly Sanders Dominique Méry Beverly Sanders, editor, editor, *Fifth International Workshop, FMPPTA 2000*, volume 1800 of *LNCS*. Springer-Verlag, 2000.
8. J. Goubault-Larrecq. Vérification de protocoles cryptographiques : la logique à la rescousse! In editor J.Goubault-Larrecq, editor, *1st international workshop on communications security on the internet, SECI'02, Tunis, Tunisia*, pages 119–152, INRIA, 2002, Sept.
9. D. Hofbauer, M. Huber, and G. Kucherov. Some results on top-context-free tree languages. In *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming*, volume 787 of *Lectures Notes in Computer Science*, pages 157–171. Springer-Verlag, 1994.
10. F. Jacquemard. Decidable approximations of term rewrite systems. In editor H. Ganzinger, editor, *Proceedings 7th Conference RTA, New Brunswick (USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1996.
11. D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. In *6th International Static Analysis Symposium*, volume 1694 of *LNCS*. Springer-Verlag, 1999. (appeared in Science of Computer Programming, 47(2-3):177-202, 2003).
12. P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proceedings of the 6th international conference LPAR*, number 1705 in Lecture Notes in Artificial Intelligence (LNAI), Tbilisi, Republic of Georgia, 1999. Springer Verlag.
13. P. Réty and J. Vuotto. Regular Sets of Descendants by Leftmost Strategy. In *Second International WRS, Copenhagen (Denmark)*, 2002. (appeared in ENTCS, 70(6), 2002).
14. P. Réty and J. Vuotto. Regular Sets of Descendants by some Rewrite Strategies. In *Proceedings of 13th Conference RTA, Copenhagen (Denmark)*, LNCS. Springer, 2002.
15. P. Réty and J. Vuotto. Context-free Tree Languages for Descendants. Research Report RR-LIFO-2004-04, LIFO, Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, BP 6759, F-45067 Orléans Cedex 2, 2004.
http://www.univ-orleans.fr/SCIENCES/LIFO/prodsci/rapports/RR/RR2004/RR-2004-04.ps.gz

16. B. Schneier. *Applied Cryptography : Protocols, algorithms, and source code in C.* John Wiley and sons edition, 1996.
17. H. Seki, T. Takai, F. Youhei, and Y. Kaji. Layered Transducing Term Rewriting System and its Recognizability Preserving Property. In *13th International Conference RTA*, volume 2378 of *Lectures Notes in computer Science*. Springer-Verlag, 2002.
18. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In L. Bachmair, editor, *Proceedings 11th Conference RTA, Norwich (UK)*, volume 1833 of *LNCS*, pages 246–260. Springer-Verlag, 2000.

ACTAS: A System Design for Associative and Commutative Tree Automata Theory

Hitoshi Ohsaki[1,2] and Toshinori Takai[3]

[1] PRESTO – Japan Science and Technology Agency (JST)
[2] National Institute of Advanced Industrial Science and Technology (AIST)
ohsaki@ni.aist.go.jp

[3] CREST – Japan Science and Technology Agency (JST)
takai@ni.aist.go.jp

Abstract ACTAS is an integrated system for manipulating associative and commutative tree automata (AC-tree automata for short), that has various functions such as for Boolean operations of AC-tree automata, computing rewrite descendants, and solving emptiness and membership problems. In order to deal with high-complexity problems in reasonable time, over- and under-approximation algorithms are also equipped. Such functionality enables us automated verification of safety property in infinite state models, that is helpful in the domain of, e.g. network security, in particular, for security problems of cryptographic protocols allowing an equational property. In runtime of model construction, for the analysis of state space expansion, intermediate status of the computation can be viewed as a numerical data table, and the line graphs are dynamically generated. Besides, a graphical user interface of the system provides us a user-friendly environment for handy use.

1 Introduction

Tree automata are the counterpart of finite automata for strings, in the sense that it inherits most of the properties holding for finite automata. It is known that *tree languages* recognized by tree automata are closed under Boolean operations and most of the decidability results are positive [4]. The tree automata framework is useful in dealing with trees (i.e. terms), and several verification techniques based on the tree automata framework have been studied [5,7]. For instance, Kaji *et. al* pointed out in [8] that some important cryptographic protocols are modeled by *term rewriting systems* (TRS for short, [2]) and tree automata, and moreover, the positive decidability results and closure properties of tree automata allow us to design an automated deduction technique for reasoning about the security problems.

In fact, verification tools for security protocols have been developed based on tree automata framework [1,3]. Genet and Viet Triem Tong provided a tree automata library, called Timbuk [5,6], in which associative and commutative properties of functions symbols are treated by using approximation. Rule-based approaches allowing associativity and commutativity have been also investigated, e.g. in [9].

Let us briefly explain below how to handle the model checking problem for infinite state transition systems in practice by using term rewriting and tree automata. We suppose that a TRS $\mathcal{R}$ over the signature $\mathcal{F}$ specifies the transition relation of a transition system $\mathcal{M}$. We say $\mathcal{M}$ admits the transition step $s \to_{\mathcal{M}} t$ under an initial state space L if (1) $s = C[l\sigma]$ and $t = C[r\sigma]$ for some rewrite rule $l \to r$ in $\mathcal{R}$, context C and substitution σ, and (2) there is a state s_0 in L such that $s_0 \to_{\mathcal{M}}^* s$, where $\to_{\mathcal{M}}^*$ is the reflexive and transitive closure of $\to_{\mathcal{M}}$. One should notice that $\to_{\mathcal{M}} \subseteq \to_{\mathcal{R}}$. Namely, the domain of the system $\mathcal{M}$ is the set of all ground terms over $\mathcal{F}$, and the state space of $\mathcal{M}$ to be verified is the reachable states from L by $\mathcal{R}$. We suppose that the initial state space L can be represented by some tree automaton $\mathcal{A}$, in such a way that $t \in L$ if and only if t is accepted by $\mathcal{A}$. So, in this setting, given a rewrite system and a tree automaton specifying each of $\mathcal{M}$ and L, the reachable state space of a transition system is considered to be defined. In the paper, we denote by $\mathcal{L}(\mathcal{A})$ the set of elements accepted by a tree automata $\mathcal{A}$, and by $[\rightarrow^*_{\mathcal{R}}](\mathcal{L}(\mathcal{A}))$ the set of reachable states.

Let P be some subset of the domain of $\mathcal{M}$, that consists of states to which we do not allow $\mathcal{M}$ to admit the transition step from any initial state in L. For instance in the network protocols, P is the set of private information, L is the initial knowledge of the intruder, and $\mathcal{R}$ is the intruder's possible operations. So the information obtainable by the intruder can be represented by $[\rightarrow^*_{\mathcal{R}}](L)$, and thus, the intersection of P and $[\rightarrow^*_{\mathcal{R}}](L)$ contains a private information that is reachable (i.e. obtainable) somehow by the intruder. In other words, the non-emptiness of the intersection indicates that the protocol is *not* secure.

However, the set $[\rightarrow^*_{\mathcal{R}}](L)$ of reachable states is not a regular tree language even if L is a regular tree language. Even worse, it is not computable in general. A tree language L is called *regular* if there exists a tree automata $\mathcal{A}$ such that L is recognized by $\mathcal{A}$. To overcome the above problem there have been several studies, such as: (1) to find a subclass, i.e. sufficient conditions, of $\mathcal{R}$ in which regularity is preserved, and (2) to extend the tree automata framework so that a wider class of tree languages can be handled. Decidable subclasses of such TRS that effectively preserve regularity have been investigated in [15,16]. Regarding the second approach, it is known that regularity is not AC-closed. In other words, the AC-closure of a regular tree language is no longer regular. A binary function symbol f in a signature $\mathcal{F}$ is *associative and commutative* if the following axioms are assumed:

$$f(x, f(y, z)) = f(f(x, y), z) \qquad f(x, y) = f(y, x)$$

The AC-closure of a tree language L is, given a subset $\mathcal{F}_{\mathsf{AC}}$ of the binary function symbols in $\mathcal{F}$, a set $\{t \mid \exists s \in L.\ s =_{\mathsf{AC}} t\}$. Here $=_{\mathsf{AC}}$ denotes the equivalence relation induced by AC-axioms of all function symbols in $\mathcal{F}_{\mathsf{AC}}$. The above negative observation reveals that for modeling a cryptographic protocol allowing equational property like Diffie-Hellman key exchange protocol, the reachable state space can not be handled by the standard tree automata.

In this research we take the second approach. We proposed in [13] an extension of tree automata, called *equational tree automata*. We also showed in [10,11] that under certain useful equational axioms, e.g. associativity and/or commutativity, tree languages accepted by the equational tree automata are closed under Boolean operations. Moreover, in this extended framework the previous Diffie-Hellman key exchange protocol can be handled, and even the verification process is automatable [12].

The AC-tree automata simulator (ACTAS) is a tool for the computation of tree automata allowing that some of the binary function symbols are associative and commutative. A screen shot of this system is presented in Fig. 1.

The class of AC-tree automata is effectively closed under union and intersection, and the membership and emptiness problems are decidable. In regular case, the emptiness test is solvable in linear time. The decidability result of emptiness problem for *non*-regular case is also positive, however, it is not manageable in the sense of real computation. It can be observed by the fact that the reachability of a Petri-net instance is known to be EXPSPACE-hard and non-regular AC-tree automata are in some sense a generalization of Petri-nets. Therefore we designed in ACTAS over- and under-approximation algorithms, for efficiently computing rewrite descendants $[\rightarrow^*_{\mathcal{R}}](\mathcal{L}(\mathcal{A}/\mathsf{AC}))$ of a given AC-tree automaton $\mathcal{A}/\mathsf{AC}$ and TRS $\mathcal{R}$.

Through the cryptographic protocol examples in this paper, we explain a verification technique, based on AC-tree automata framework, using the above functionality in ACTAS.

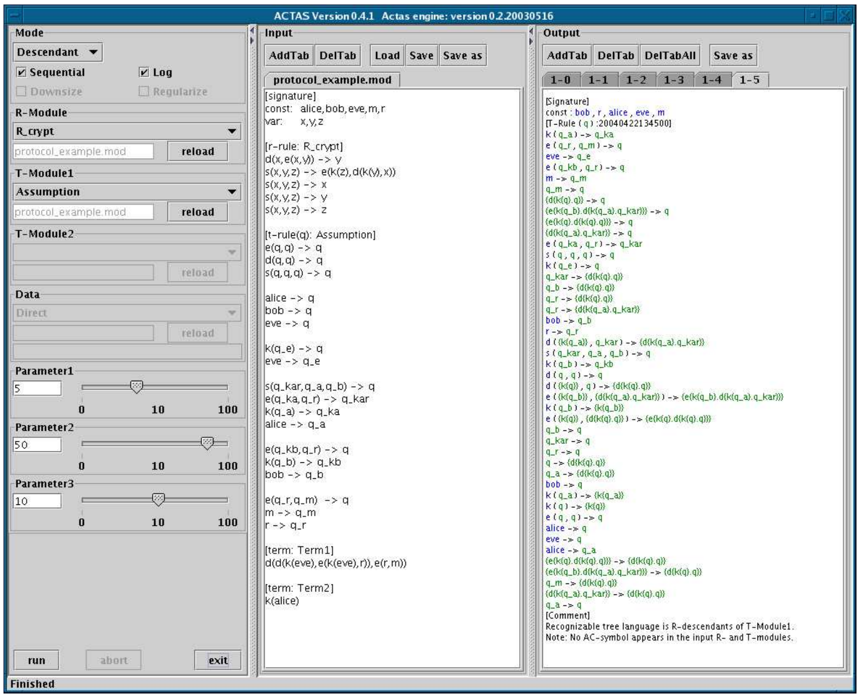

Figure 1. Control panel of ACTAS

2 AC-Tree Automata

We begin this section by introducing AC-tree automata. We then explain how to operate ACTAS as a tool for for manipulating AC-tree automata and even as a tool supporting automated verification.

A *tree automaton* (TA for short) $\mathcal{A}$ is a 4-tuple $(\mathcal{F}, \mathcal{Q}, \mathcal{Q}_{fin}, \Delta)$, whose components are the *signature* $\mathcal{F}$, i.e. a finite set of function symbols with fixed arities, a finite set $\mathcal{Q}$ of special constant symbols, called *states*, with $\mathcal{F} \cap \mathcal{Q} = \varnothing$, a subset $\mathcal{Q}_{fin}$ of $\mathcal{Q}$ whose elements are called *final states*, and a finite set Δ of *transition rules* in one of the following forms:

$f(p_1, \ldots, p_n) \to q_1$	(TYPE 1)
$f(p_1, \ldots, p_n) \to f(q_1, \ldots, q_n)$	(TYPE 2)
$p_1 \to q_1$	(TYPE 3)

such that $f \in \mathcal{F}$ with $\mathsf{arity}(f) = n$ and $p_1, \ldots, p_n, q_1, \ldots, q_n \in \mathcal{Q}$. In TYPE 2 the root function symbols of the left- and right-hand sides must be the same. Transition rules in TYPE 3 are called *ϵ-rules* ("epsilon-rules"). A TA $\mathcal{A}$ is called *regular* if Δ consists only of rules in TYPE 1. Rules of TYPE 2 are not treated in [4]. Under consideration of equational properties, however, TYPE 2 is essential in the sense that, e.g. recognizable tree languages of our definition have a bijective correspondence to the word language hierarchy [10]. An efficient algorithm for the intersection of such AC-tree automata, presented in [13], is also one of the advantages.

A transition move $\to_{\mathcal{A}}$ is the rewrite relation $\to_{\Delta}$ by taking Δ as a TRS Δ over the signature $\mathcal{F} \cup \mathcal{Q}$. A ground term t over $\mathcal{F}$ is *accepted* if $t \to_{\mathcal{A}}^{*} q_f$ for some $q_f \in \mathcal{Q}_{fin}$. The set of terms accepted by a tree automaton $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$. A tree language L, that is a subset of all ground terms, is *recognizable* if there is a tree automata $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$.

```
1: [signature]
2: AC: f
3: const: a,b
4:
5: [t-rule(q): A]
6: a -> q_a
7: b -> q_b
8: f(q_a,q_b) -> q
9: f(q,q) -> q
```

Figure 2. An example of AC-tree automaton in ACTAS

An equational tree automaton is a pair of a tree automaton $\mathcal{A}$ and an equational theory $\mathcal{E}$, denoted as $\mathcal{A}/\mathcal{E}$. The transition move is defined by the relation $\rightarrow_{\mathcal{A}}$ modulo $\mathcal{E}$. An AC-tree automaton is an equational tree automaton whose equational theory is the associativity and commutativity axioms for some of the binary function symbols in $\mathcal{F}$. The basic properties of AC-tree automata are stated below:

Theorem 1. *[11,13] (1) The class of tree languages accepted by AC-tree automata are closed under union and intersection. (2) The class of tree languages accepted by* regular *AC-tree automata are closed under Boolean operations (union, intersection, and complementation). (3) The membership problem and the emptiness problem for AC-tree automata are decidable.* □

We consider the tree automaton $\mathcal{A}$ with the following transition rules

$$\mathsf{a} \rightarrow q_{\mathsf{a}},\ \mathsf{b} \rightarrow q_{\mathsf{b}},\ \mathsf{f}(q_{\mathsf{a}}, q_{\mathsf{b}}) \rightarrow q,\ \mathsf{f}(q, q) \rightarrow q$$

and the final state q. Suppose f is associative and commutative, then the AC-tree automaton $\mathcal{A}/\mathsf{AC}$ accepts such trees t that

$$|t|_{\mathsf{a}} = |t|_{\mathsf{b}}$$

i.e. the number of occurrences of a is the same as the number of occurrences of b in the same tree t. One should notice that the above language is not recognizable with any tree automata.

In ACTAS the above example $\mathcal{A}/\mathsf{AC}$ is specified as shown in Fig. 2. The signature is specified by declaring AC-symbols and constant function symbols. The other constant symbols are recognized as state symbols. The tree automaton $\mathcal{A}$ is specified in the module named `A`, by listing the transition rules (6–9th lines). The argument `q` of the predefined symbol `t-rule` is the final state of `A`.

At the current implementation ACTAS is equipped with the following functions for Boolean operations (1)–(2) and rewrite descendants computation (3), and two solvers for membership and emptiness problems (4)–(5):

1. Given two AC-tree automata $\mathcal{A}/\mathsf{AC}$ and $\mathcal{B}/\mathsf{AC}$, construct an AC-tree automaton $\mathcal{C}/\mathsf{AC}$ such that $\mathcal{L}(\mathcal{C}/\mathsf{AC}) = \mathcal{L}(\mathcal{A}/\mathsf{AC}) \cup \mathcal{L}(\mathcal{B}/\mathsf{AC})$.
2. Given two AC-tree automata $\mathcal{A}/\mathsf{AC}$ and $\mathcal{B}/\mathsf{AC}$, construct an AC-tree automaton $\mathcal{C}/\mathsf{AC}$ such that $\mathcal{L}(\mathcal{C}/\mathsf{AC}) = \mathcal{L}(\mathcal{A}/\mathsf{AC}) \cap \mathcal{L}(\mathcal{B}/\mathsf{AC})$.
3. Given an AC-tree automaton $\mathcal{A}/\mathsf{AC}$ and a TRS $\mathcal{R}$ whose rewrite rules do not contain AC-function symbols, construct AC-tree automaton $\mathcal{C}/\mathsf{AC}$ such that $\mathcal{L}(\mathcal{C}/\mathsf{AC}) = [\rightarrow^*_{\mathcal{R}}](\mathcal{L}(\mathcal{A}/\mathsf{AC}))$.

```
[signature]
AC: f
const: 0

[r-rule: R]
0 -> s(s(0))
[t-rule(q): A1]
0 -> zero
s(zero) -> one
zero -> q
f(one,one) -> q
f(q,q) -> q
[t-rule(q): A2]
0 -> q
s(q) -> one
s(one) -> q
f(one,one) -> q
f(q,q) -> q
```

Figure 3. Computing rewrite descendants of AC-tree automata

4. Given an AC-tree automaton $\mathcal{A}/\mathsf{AC}$ and a term t, determine if $t \in \mathcal{L}(\mathcal{A}/\mathsf{AC})$.
5. Given an AC-tree automaton $\mathcal{A}/\mathsf{AC}$, determine if $\mathcal{L}(\mathcal{A}/\mathsf{AC}) = \varnothing$,

The computation results obtained by the operations (1)–(3) can be re-used, as new inputs. For automated verification, the above function (3) is useful in order to construct models.

We consider the example in Fig. 3. The TRS $\mathtt{R}$ consists of the single rewrite rule $\mathtt{0} \to \mathtt{s}(\mathtt{s}(\mathtt{0}))$. The tree automaton $\mathtt{A1}$ accepts a tree t if $t = \mathtt{0}$, $t = \mathtt{f}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{0}))$, or $t = \mathtt{f}(t_1, t_2)$ such that t_1, t_2 are accepted by $\mathtt{A1}$. Under the assumption that $\mathtt{f}$ is associative and commutative, the above language coincides with $L = \{t \mid |t|_{\mathtt{s(0)}} \text{ is even }\}$. Thus a term t is reachable by $\mathtt{R}$ from some term t' in L if and only if t satisfies that $t = C[s_1, \ldots, s_n]$ such that C consists of $\mathtt{f}$, $\mathsf{root}(s_i) = \mathtt{s}$ or $\mathtt{0}$, and $\sum_{i=1}^{n} [\![s_i]\!]$ is even, where $[\![\mathtt{0}]\!] = 0$ and $[\![\mathtt{s}(t)]\!] = [\![t]\!] + 1$. Roughly speaking, t is an element in $[\to^*_{\mathtt{R}}](\mathcal{L}(\mathtt{A1}/\mathsf{AC}))$ if the sum of natural numbers occurring in t is even. For instance, $\mathtt{f}(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{f}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{0}))), \mathtt{f}(\mathtt{s}(\mathtt{0}), \mathtt{0}))$ belongs to $[\to^*_{\mathtt{R}}](\mathcal{L}(\mathtt{A1}/\mathsf{AC}))$ that is represented by an AC-tree automaton. Actually, this language is accepted by the AC-tree automaton $\mathtt{A2}/\mathsf{AC}$. An advantage of ACTAS is that by using the embedded functions, we can construct such AC-tree automata automatically.

Nevertheless, one can observe that the bounded computation is often required in constructing AC-tree automata that accept *rewrite descendants*, because (a) the language $[\to^*_{\mathcal{R}}](\mathcal{L}(\mathcal{A}/\mathsf{AC}))$ is *not* computable in general, and (b) even if $[\to^*_{\mathcal{R}}](\mathcal{L}(\mathcal{A}/\mathsf{AC}))$ is computable under a certain condition, it may not be recognizable with AC-tree automata. The two cases correspond to non-terminating computation.

From this observation, PARAMETERS 1–3 are arranged in the control panel. (See the lower left-hand corner in Fig. 1). PARAMETER 1 restricts the number of the execution of the outermost-loop in the algorithm: For instance, PARAMETER 1 being 0 in Function (3) checks whether a given ACTAS code is syntactically correct. By setting PARAMETER 1 to be n ($\geqslant 1$), we can execute the rewrite descendant computation only of n-loops.

For computing over- or under-approximated result, we select positive integers for PARAMETERS 2 and 3. For instance, if non-left-linear rewrite rules like $f(x, x) \to x$ are included in the rewrite system, we need to check in the algorithm of Function (3) whether tree automata $\mathcal{A}_1 = (\mathcal{F}, \mathcal{Q}, \{p_1\}, \Delta)$ and $\mathcal{A}_2 = (\mathcal{F}, \mathcal{Q}, \{p_2\}, \Delta)$ satisfy

$$\mathcal{L}(\mathcal{A}_1/\mathsf{AC}) \cap \mathcal{L}(\mathcal{A}_2/\mathsf{AC}) \neq \varnothing$$

for some p_1, p_2 in $\mathcal{Q}$ with $p_1 \neq p_2$. The values of PARAMETERS 2 and 3 restrict the search depth and width of the decision procedure of the above question. But if PARAMETERS 2–3 are the maximum 100, upper-bound limitation is ignored. That in turn results in the exact solution if the computation terminates. Hence, by selecting appropriate positive integers for PARAMETERS 1–3, one can obtain under-approximated results of $[\to^*_{\mathcal{R}}](\mathcal{L}(\mathcal{A}/\mathsf{AC}))$ in reasonable time. On the other hand, both PARAMETERS 2 and 3 being 0 turn out over-approximated results.

3 Cryptographic Protocol Verification

We explain below how to verify network protocols by using ACTAS. In the protocol illustrated in Fig. 4, we write $E(x, y)$ for a message y encrypted by some key x, and $K(x)$ for a principal x's secret key. The goal of this protocol is to send a secret message m from alice to bob without losing the secrecy. Hence m is encrypted with another secret key (nonce) r, and thus r is also encrypted and transfered to bob. In this network communication, alice first sends a triple of $E(K(\mathsf{alice}), \mathsf{r})$, alice the sender's ID, and bob the receiver's ID. Then server reacts to this request by sending back $E(K(\mathsf{bob}), \mathsf{r})$ to alice. At the final step alice sends the pair of $E(K(\mathsf{bob}), \mathsf{r})$ and $E(\mathsf{r}, \mathsf{m})$, to bob. The latter component is generated by encrypting m by r. The receiver bob can retrieve the clear-text m by decrypting $E(K(\mathsf{bob}), \mathsf{r})$ first.

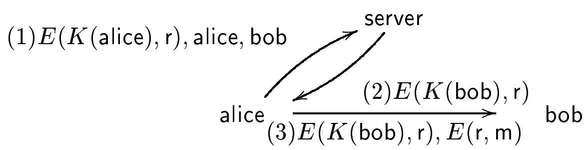

Figure 4. A cryptographic protocol

Now we assume that an intruder eve has the following abilities:

1. If eve knows x and $E(x, y)$, then eve also knows y.
2. eve knows how to apply encryption and decryption functions E and D, i.e. if eve knows x and y, eve can construct $E(x, y)$ and $D(x, y)$.
3. eve knows its own secret key $K(\mathsf{eve})$ and all principles names $\mathsf{alice}, \mathsf{bob}$ and eve.
4. eve can wiretap the network channels, i.e. eve knows all information flowing in the network of Fig. 4.

To detect the security flaw (otherwise, to ensure the secrecy of the protocol), we verify the protocol by using TRS and tree automata. We first model by a tree automaton the initial knowledge of the intruder eve. We then generate the set of states reachable from the initial knowledge by the following TRS:

$$\mathcal{R}_{\mathsf{crypt}} = \{D(x, E(x, y)) \to y\}$$

The TRS $\mathcal{R}_{\mathsf{crypt}}$ corresponds to the above intruder's ability 1. The other assumptions 2–4, which are the intruder's initial knowledge and available operations, can be represented by the tree automaton ($\mathcal{A}_{\mathsf{initial}}$), that is shown in Fig. 5.

The tree automaton $\mathcal{A}_{\mathsf{initial}}$ accepts a term t if and only if t is obtainable by the intruder without using the encryption-decryption axiom $\mathcal{R}_{\mathsf{crypt}}$. The set $[\to^*_{\mathcal{R}_{\mathsf{crypt}}}](\mathcal{L}(\mathcal{A}_{\mathsf{initial}}))$ of reachable states corresponds to the fixpoint of the intruder's knowledge. By computing rewrite descendants (Function (3) in ACTAS), we have a tree automaton $\mathcal{A}_{\mathsf{fixpoint}}$ that satisfies $\mathcal{L}(\mathcal{A}_{\mathsf{fixpoint}}) = [\to^*_{\mathcal{R}_{\mathsf{crypt}}}](\mathcal{L}(\mathcal{A}_{\mathsf{initial}}))$ if there exists. Therefore, by solving membership constraint (Function (4) in ACTAS) $\mathsf{m} \in \mathcal{L}(\mathcal{A}_{\mathsf{fixpoint}})$?, it can be determined whether or not the protocol is secure against wiretapping.

```
[signature]
const:  alice,bob,eve,m,r
var:    x,y

[r-rule: R_crypt]
d(x,e(x,y)) -> y

[t-rule(q): A_initial]
d(q,q) -> q
e(q,q) -> q

alice -> q
bob -> q
eve -> q
k(q_e) -> q
eve -> q_e

e(q_ka,q_r) -> q
k(q_a) -> q_ka
alice -> q_a

e(q_kb,q_r) -> q
k(q_b) -> q_kb
bob -> q_b

e(q_r,q_m)  -> q
m -> q_m
r -> q_r
```

Figure 5. Specification code of cryptographic protocol assuming wiretapping only

```
[signature]
const:  alice,bob,eve,m,r
var:    x,y,z

[r-rule: R_crypt2]
d(x,e(x,y)) -> y
s(x,y,z) -> e(k(z),d(k(y),x))
s(x,y,z) -> x
s(x,y,z) -> y
s(x,y,z) -> z

[t-rule(q): A_initial2]
d(q,q) -> q
e(q,q) -> q
s(q,q,q) -> q

alice -> q
bob -> q
eve -> q
k(q_e) -> q
eve -> q_e

s(q_kar,q_a,q_b) -> q
e(q_ka,q_r) -> q_kar
k(q_a) -> q_ka
alice -> q_a

e(q_kb,q_r) -> q
k(q_b) -> q_kb
bob -> q_b

e(q_r,q_m)  -> q
m -> q_m
r -> q_r
```

Figure 6. Specification code of cryptographic protocol assuming active attack

Along the similar construction scheme, we can detect that the same protocol is *not* secure against impersonation. The associated tree automaton and TRS is illustrated in Fig. 6, that represents the previous protocol example in which intruder's active attack is assumed. By allowing that every principal (including the intruder eve) sends a request to server, we add the rewrite rule

`s(x,y,z) -> e(k(z),d(k(y),x))`

and the transition rule `s(q,q,q) -> q` to the previous code. In the left-hand side `s(x,y,z)` of the rewrite rule, variables `x`, `y`, `z` are (intended to) assigned to encrypted data, sender's ID and receiver's ID, respectively. The result of this rewriting step is a server's reply, that is (assumed to be) received by a sender. More precisely, the sender receives an encrypted message $\mathtt{e(k(}s_1\mathtt{),d(k(}s_2\mathtt{),}s_3\mathtt{))}$, that is once decrypted with a sender's key at server site

loop	♯(t-rule)	♯(state)	time (sec)
0	18	9	—
1	30	16	3
2	41	16	12
3	44	16	22
4	44	16	32

* The computation is saturated at 3rd loop.

Figure 7. The numbers of transition rules and state symbols at each loop

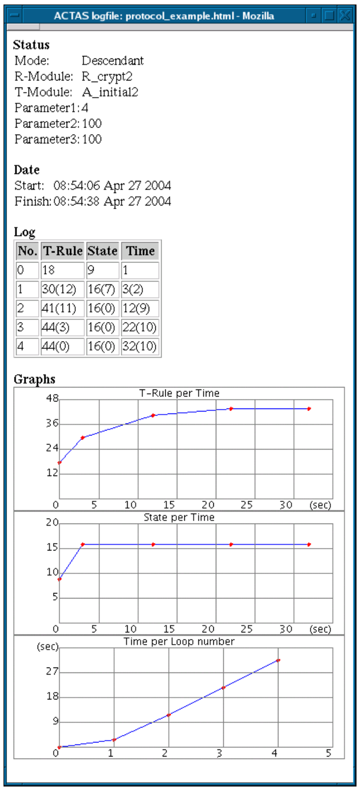

Figure 8. Intermediate status of the computation in HTML format

and the result is encrypted with a receiver's key. We assume also that eve can decompose any data of the form $\mathsf{s}(t_1, t_2, t_3)$, and this situation is represented by the other three rewrite rules.

In the experiment, by using Function (3) of ACTAS with PARAMETER 1 to be 4 or the greater (and the others to be arbitrary positive integers), we obtain an under-approximated result. The numbers of transition rules and state symbols at each loop are shown in Fig. 7. This table together with the line graphs is generated automatically. We can save the displayed data as HTML format files (Fig. 8).

The resulting tree automaton accepts the secret message m, and thus, we know that the protocol is not secure. Actually, the protocol allows the following security flaw: The intruder eve first sends the tuple of $E(K(\mathsf{alice}), \mathsf{r})$, alice, eve to server. These elements are included in eve's initial knowledge due to Assumptions 3–4. Then eve obtains $E(K(\mathsf{eve}), \mathsf{r})$ as a response from server. By Assumptions 2–3, eve constructs $D(D(K(\mathsf{eve}), E(K(\mathsf{eve}), \mathsf{r})), E(\mathsf{r}, \mathsf{m}))$ that gives rise to m, because of $\mathcal{R}_{\mathsf{crypt}}$ that is Assumption 1.

4 Diffie-Hellman Key-Exchange Protocol

The protocol illustrated below is called Diffie-Hellman key-exchange algorithm (e.g. Section 22.1, [14]):

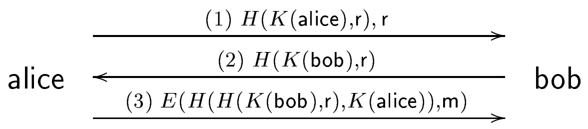

Figure 9. Diffie-Hellman key-exchange algorithm

In the figure $H(x, y)$ stands for the composition of data x and y, that is an integer y^x (mod p where p is given) in the real situation. To simplify the property of H, we assume in this model that H is implemented as $H(x, y) = p^{x+y}$ (mod q where p, q are given). A secret key of a principal x is denoted by $K(x)$. The goal of this protocol is without losing the secrecy to share a session key by exchanging some data between the initiator (alice in the example) and the receiver (bob), and then to send from the initiator to receiver a message encrypted by the session key. The protocol consists of the three steps: alice first chooses a number r and sends it to bob together with an integer $H(K(\mathsf{alice}), \mathsf{r})$. We suppose that no one else can retrieve $K(\mathsf{alice})$ only from $H(K(\mathsf{alice}), \mathsf{r})$. At the second step bob returns $H(K(\mathsf{bob}), \mathsf{r})$ to alice. Because of the exponentiation in the implementation of H, one can assume that H is associative and commutative:

$$H(x, H(y, z)) = H(H(x, y), z) \qquad H(x, y) = H(y, x)$$

Due to the first two steps, alice generates $H(H(K(\mathsf{bob}), \mathsf{r}), K(\mathsf{alice}))$ by combining $H(K(\mathsf{bob}), \mathsf{r})$ and $K(\mathsf{alice})$. Similarly, bob also generates $H(H(K(\mathsf{alice}), \mathsf{r}), K(\mathsf{bob}))$ such that:

$$H(H(K(\mathsf{alice}), \mathsf{r}), K(\mathsf{bob})) \;=_{\mathsf{AC}}\; H(H(K(\mathsf{bob}), \mathsf{r}), K(\mathsf{alice})).$$

Hence bob obtain the message m as follows.

$$\begin{array}{ll} D(H(H(K(\mathsf{alice}), \mathsf{r}), K(\mathsf{bob})), E(H(H(K(\mathsf{bob}), \mathsf{r}), K(\mathsf{alice})), \mathsf{m})) & =_{\mathsf{AC}} \\ D(H(H(K(\mathsf{bob}), \mathsf{r}), K(\mathsf{alice})), E(H(H(K(\mathsf{bob}), \mathsf{r}), K(\mathsf{alice})), \mathsf{m})) & \to_{\mathcal{R}_{\mathsf{crypt}}} \mathsf{m} \end{array}$$

The assumption of the protocol can be specified as the ACTAS code like in Fig. 10. In this setting we suppose that (i) the intruder eve can wiretap the network channels, but (ii) eve does not actively attack to the protocol. Even if the binary function symbol H is associative and commutative, the AC-rewrite descendants can be computed by using the same algorithm of [15], because H does not appear in rewrite rules. But since the left-hand side of $\mathcal{R}_{\mathsf{crypt}}$ has multiple occurrences of the variable x, the intersection-emptiness problem for AC-tree automata has to be dealt with in the algorithm. The intersection of AC-tree automata can be computed in ACTAS efficiently. However, the resulting AC-tree automata are no longer AC-regular in this efficient construction, and to solve the emptiness problem for non-regular AC-tree automata is EXPSPACE-hard [11].

In fact, when computing the exact solution fully automatically, it is nearly non-terminating computation. So we apply for this example the over-approximation algorithm (by

```
[signature]
AC: h
const: alice,bob,eve,m,r
var: x,y

[r-rule: R_crypt]
d(x,e(x,y)) -> y

[t-rule(q): Assumption]
d(q,q) -> q
e(q,q) -> q
h(q,q) -> q

alice -> q
bob -> q
eve -> q

k(q_e) -> q
eve -> q_e
h(q_ka,q_r) -> q
k(q_a) -> q_ka
alice -> q_a
r -> q_r

r -> q

h(q_kb,q_r) -> q
k(q_b) -> q_kb
bob -> q_b

e(q_kabr,q_m) -> q
h(q_kbr,q_ka) -> q_kabr
h(q_kb,q_r) -> q_kbr
m -> q_m
```

Figure 10. Diffie-Hellman key-exchange protocol (assuming wiretapping only)

setting PARAMETER 2 to be 0) and the under-approximation (by choosing appropriate positive integers for PARAMETERS 2 and 3). Unfortunately, the over-approximated result, that is an AC-tree automaton accepting some superset of eve's obtainable knowledge, accepts m. It does not mean security flaw of the protocol. But at the current implementation there is no option to refine the over-approximated result. From the under-approximation results, the security flaw of the protocol is not detected either.

Despite of the above experiment, we observe that this protocol example can be handled in the AC-tree automata framework. Let $\mathcal{A}_0/\mathsf{AC}$ be the AC-tree automaton initially provided, and $\mathcal{L}(\mathcal{A}_0(q)/\mathsf{AC})$ a tree language that is accepted by $\mathcal{A}_0/\mathsf{AC}$ with the final state q. Then, for each state symbols p and q of $\mathcal{A}_0/\mathsf{AC}$, the intersection-emptiness of $\mathcal{L}(\mathcal{A}_0(p)/\mathsf{AC})$ and $\mathcal{L}(\mathcal{A}_0(q)/\mathsf{AC})$ is solvable, because $\mathcal{L}(\mathcal{A}_0(p)/\mathsf{AC})$ is finite if $p \neq \mathsf{q}$, where q is the final state symbol of the AC-tree automaton in Fig. 10. Membership function in the system assists us to perform the above test. This implies that all the possible combinations of p and q are computable in reasonable time. Moreover, by using this result, we can easily examine whether new transition rules are generated. In fact, no new transition rule can be constructed for $\mathcal{A}_0/\mathsf{AC}$ and $\mathcal{R}_{\mathsf{crypt}}/\mathsf{AC}$. Therefore $\mathcal{A}_0/\mathsf{AC}$ is already the fixpoint, namely, the protocol is secure against wiretapping.

Regarding the active attack by assuming impersonation, the security flaw of the protocol is noted in [14].

Acknowledgments. The authors thank the three anonymous referees for their numerous comments and suggestions to improve the early version of the paper.

References

1. A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò and L. Vigneron: *The AVISS Security Protocol Analysis Tool*, Proc. of 14th CAV, Copenhagen (Denmark), LNCS 2404, pp. 349–353, Springer-Verlag, 2002.
2. F. Baader and T. Nipkow: *Term Rewriting and All That*, Cambridge University Press, 1998.

3. Y. Chevalier and L. Vigneron: *Automated Unbounded Verification of Security Protocols*, Proc. of 14th CAV, Copenhagen (Denmark), LNCS 2404, pp. 324–337, Springer-Verlag, 2002.
4. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi: *Tree Automata Techniques and Applications*, 2002. Draft available from `http://l3ux02.univ-lille3.fr/tata/`
5. T. Genet and F. Klay: *Rewriting for Cryptographic Protocol Verification*, Proc. of 17th CADE, Pittsburgh (PA), LNCS 1831, pp. 271–290, Springer-Verlag, 2000.
6. T. Genet and V. Viet Triem Tong: *Reachability Analysis of Term Rewriting Systems with Timbuk*, Proc. of 8th LPAR, Havana (Cuba), LNAI 2250, pp. 691–702 Springer-Verlag, 2001.
7. H. Hosoya, J. Vouillon and B.C. Pierce: *Regular Expression Types for XML*, Proc. of 5th ICFP, Montreal (Canada), SIGPLAN Notices 35(9), pp. 11–22, ACM Press, 2000.
8. Y. Kaji, T. Fujiwara and T. Kasami: *Solving a Unification Problem under Constrained Substitutions Using Tree Automata*, Journal of Symbolic Computation 23, pp. 79–117, 1997.
9. José Meseguer: *Software Specification and Verification in Rewriting Logic*, Proc. of NATO Advanced Study Institute on Models, Algebras and Logic of Engineering Software, Computer and Systems Sciences 191, pp. 133–193, IOS Press, 2003
10. H. Ohsaki, H. Seki and T. Takai: *Recognizing Boolean Closed A-Tree Languages with Membership Conditional Rewriting Mechanism*, Proc. of 14th RTA, Valencia (Spain), LNCS 2706, pp. 483–498, Springer-Verlag, 2003.
11. H. Ohsaki and T. Takai: *Decidability and Closure Properties of Equational Tree Languages*, Proc. of 13th RTA, Copenhagen (Denmark), LNCS 2378, pp. 114–128, Springer-Verlag, 2002.
12. H. Ohsaki and T. Takai: *A Tree Automata Theory for Unification Modulo Equational Rewriting*, Proc. of 16th UNIF, Copenhagen (Denmark), 2002. Draft available from `http://staff.aist.go.jp/hitoshi.ohsaki/unif2002.ps.gz`
13. H. Ohsaki: Beyond Regularity: *Equational Tree Automata for Associative and Commutative Theories*, Proc. of 15th CSL, Paris (France), LNCS 2142, pp. 539-553, Springer-Verlag, 2001.
14. B. Schneier: *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Second Edition, John Wiley & Sons, 1996.
15. T. Takai, Y. Kaji and H. Seki: *Right-Linear Finite-Path overlapping Term Rewriting Systems Effectively Preserve Recognizability*, Scienticae Mathematicae Japonicae. To appear.
16. T. Takai, H. Seki, Y. Fujinaka and Y. Kaji: *Layered Transducing Term Rewriting System and Its Recognizability Preserving Property*, IEICE Transactions on Information and Systems E86-D(2), pp. 285–295, 2003. Information about IEICE Transactions is found at `http://www.ieice.org`

Rule-based Programs describing Internet Security Protocols*

Yannick Chevalier and Laurent Vigneron

LORIA - UHP-UN2-CNRS
Campus Scientifique, B.P. 239
54506 Vandœuvre-lès-Nancy Cedex, France
{chevalie,vigneron}@loria.fr

Abstract We present a low-level specification language used for describing real Internet security protocols. Specifications are automatically generated by a compiler, from TLA-based high-level descriptions of the protocols. The results are rule-based programs containing all the information needed for either implementing the protocols, or verifying some security properties. This approach has already been applied to several well-known Internet security protocols, and the generated programs have been successfully used for finding some attacks.

1 Introduction

Internet is becoming everyday a more widely used medium for electronic commerce. This development is hampered by the natural insecurity of communications, as it is not possible to guarantee that some data exchanged is not listened by someone else, or even that it really originated from the claimed sender. This lack of security leads to the development of *security protocols*, that is small messages sequences, after which the author provides some properties to the user, such as the correct identification of the users (called *agents*) and the privacy of some data pieces.

There has been a significant amount of work toward the specification of security protocols in the recent years [23,1,17,7,19]. However, a large part of this work, including our own, is applied only to *toy protocols* in the *Alice&Bob* notation, *i.e.* as a linear scenario describing the messages exchanged.

Our main goal is to successfully handle complex protocols such as those under discussion at the IETF [18]. To this end, a new High-Level Protocol Specification Language (HLPSL) was developed in the AVISPA project, having in mind the constructions often found in the specification of these protocols [8]. We have written a compiler transforming a protocol specification in this language to a set of rewrite rules. We present in this article not the compiler itself, but the encoding of high-level properties in rewrite rules. We believe that rule-based systems are a natural framework to encode the properties encountered when studying cryptographic protocols.
We do not discuss in this paper about verification methods. For more information concerning them, see [12,10] for instance.

This paper is organized as follows: we first describe shortly the high-level specification language (Section 2); in Section 3, we describe how the initial specification is translated into a rule-based program, corresponding to a low-level specification; then, we list the examples of real Internet security protocols that have already been successfully compiled (Section 4), their rule-based specification being used by several verification tools. In the conclusion, we

* This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-39252 AVISPA project, and the RNTL 03V360 Prouvé project.

compare our compiler with the MuCAPSL-MuCIL translator [13], based on the powerful language MuCAPSL [20].

2 Specifying Protocols and Intruder

In this section, we first present our objectives concerning protocols specifications, recalling what should be the properties of specification languages. Such objectives are achieved by the use of two languages: a high-level language that we describe shortly (see [8] for a complete description and semantics), to be used by protocol designers; and a low-level language (described in a further section) to be used by engineers.

2.1 Specification Languages

Studying security protocols is a very important domain nowadays. This is often done in a three steps process. First, protocols are specified in a high-level, easily understandable language. Then, this specification is analyzed to ensure that there are no trivial errors. If no flaws are found, the protocol is verified in a time-consuming last step. We are interested in this paper in the translation from the high-level language used in the first step to a language suitable for analysis.

A lot of high-level specification languages have been defined, some very simple (such as those based on the Alice&Bob notation [7,19]) and some dedicated to a specific tool [17,23]. But all of them either have a very limited expressiveness, or need a high level of expertise, or both.

Our aim in studying security protocols is based on the following objectives concerning the specification language: we want to consider real Internet protocols and to define a language that can be used by industrials; this language has to be able to express many security properties and has to have a clear semantics; in addition, it has to provide a basis for automated analysis.

These objectives are motivated by the fact that protocols specifications have to be used as documentations: in general, protocols are described in long documents (for example provided by the IETF); this makes them difficult to understand, and may lead to different interpretations according to the objectives of the reader (to implement the protocol, to verify it, or simply to understand it). Moreover, the underlying scientific foundations have to be clear in a protocol specification. This is very important for knowing if the protocol is easy to implement or not.

For summing up, the requirements for a high-level protocol specification language are:

- Simplicity and comprehension: specifications have to be easy to write, to read and to understand.
- Flexibility: a modification in the protocol should not mean to rewrite the whole specification.
- Non-ambiguity: the semantics of the language should be clear enough for avoiding ambiguous interpretations.
- Modularity: a specification has to be modular; this permits to share some modules between several protocols, and possibly to hide some parts of the protocol.
- Expressiveness: this is the most difficult criteria to satisfy; it can be decomposed into the following points:

– Control flow: the language has to provide some primitives for controlling the reception and emission of messages, to describe a negotiation phase, for instance.
– Knowledge of the intruder and agents: the user has to be able to manage the knowledge of the participants to the protocol, and that of the intruder.
– Cryptographic primitives: the language has to permit the use of fresh information, such as nonces (*numbers used once*) or keys, of hash functions, and also of signatures.
– Complex initial states: sometimes, protocols do not start from scratch, and assume that some preliminary actions have already been done; so the language has to permit the use of complex initial states.
– Complex message types: in most languages messages are generally built with simple primitives, such as encryption, decryption, pairing, but some more complex data structures may be needed for describing internal data structures or messages of roles (e.g. sets, lists, records).
– Algebraic properties: in some protocols, the mechanisms for encryption and decryption or for creating keys is details and involves the use of algebraic operators, such as exclusive-or or exponentiation; such operators have to be recognized by the specification language, because they satisfy some properties that may be considered for implementing the protocols.

Because none of the existing languages satisfies all these crucial requirements, we have decided, with our partners of the AVISPA project (Siemens AG Münich, DIST Genova and ETH Zürich), to define a new specification language.

2.2 An Expressive Specification Language

We will illustrate our new specification language [8] with the well-known Needham and Schroeder Public Key (NSPK) protocol. This example is usually considered as very simple and far away from real protocols. But the version that can be seen in most papers is a simplified one. Our aim being to consider all the options that may rise during the execution of a protocol, we will consider a more complex variant of the NSPK protocol: the NSPK Key Server (NSPK-KS). This protocol is given as follows, using an Alice&Bob-based notation:

$$
\begin{array}{l}
\textit{if } A \textit{ does not know } K_B, \\
\qquad \begin{array}{ll} A \to S: & A, B \\ S \to A: & \{B, K_B\}_{K_S^{-1}} \end{array} \\
A \to B: \quad \{N_A, A\}_{K_B} \\
\textit{if } B \textit{ does not know } K_A, \\
\qquad \begin{array}{ll} B \to S: & B, A \\ S \to B: & \{A, K_A\}_{K_S^{-1}} \end{array} \\
B \to A: \quad \{N_A, N_B\}_{K_A} \\
A \to B: \quad \{N_B\}_{K_B}
\end{array}
$$

The main originality of this protocol is that agents A and B, needing to know the public key of each other for running NSPK, can ask it to a server S if they do not already have it. This means that some steps of the protocol are conditional.

Such a protocol is impossible to specify in other high-level dedicated protocol specification languages, because none of them permits to easily define such guarded transitions.

Modular specification using roles. Our specification language is modular: protocols are not given as a sequence of messages, but as a set of roles. There are basic roles, each one representing the behavior of one agent in the protocol. There are also composed roles, representing the composition of other roles or their instantiations to be considered.
Informally, basic roles correspond to an Alice&Bob description with control; composed roles correspond to the use of CSP-like operators.

In our example, there are three basic roles: Alice, initiator of the protocol and named A; Bob, responder, named B; and Server, S.

Each role is an independent process, with external information given as parameter, and a local environment.

```
role Server(S: agent, Ks: public_key,
            KeyMap: (agent.public_key) set,
            SND,RCV: channel (dy)) played_by S def=
  local A: agent, B: agent, Kb: public_key
  knowledge(S) = { inv(Ks) }
  transition
    step0.  RCV(A'.B') /\ in(B'.Kb', KeyMap)
      =|> SND({B'.Kb'}inv(Ks))
end role
```

The local environment is given by a list of local variables, and a list of knowledge (for example, S knows the inverse of the public key Ks, i.e. the corresponding private key).

The messages are exchanged via channels (SND and RCV), parameterized by their level of security that corresponds to the model of the intruder to be used for them: dy stands for the standard Dolev-Yao model [15] (no specific protection); ota stands for the over-the-air model (no diverted message). For a role, there may be several channels for sending and receiving messages, depending on their security level and on the concerned agents.

Composed roles are used for describing how to combine roles: this is possible to run roles in parallel or in sequence. For example, in the following NSPK role, Alice and Bob roles are run in parallel, and in as many instances as required. The Server role does not appear in this composed role because there will be only one, and it will be launched in another composed role, in parallel with NSPK.

```
role NSPK(SC, RC, S_SRV, R_SRV: agent -> channel (dy),
            Ks: public_key,
            Instances: (agent.agent.public_key.public_key) set,
            KeySet: agent -> (agent.public_key) set) def=
  composition
    /\_{ in(A.B.Ka.Kb, Instances) }
            Alice(A,B,Ka,Ks,KeySet(A),SC(A),RC(A),S_SRV(A),R_SRV(A))
            /\ Bob(A,B,Kb,Ks,KeySet(B),SC(B),RC(B),S_SRV(B),R_SRV(B))
end role
```

Control flow: guarded transitions. The main part of a role is the description of a transition system. Its originality is that transitions are not ordered: they are of the form condition =|> action, where condition and action are multisets of facts; a transition can be applied as soon as its condition is satisfied. So, in a role, several transitions can be

applicable at the same time.
We illustrate the use of guarded transitions with the role Alice.

```
role Alice(A,B: agent, Ka,Ks: public_key,
          KeyRing: (agent.public_key) set,
          SND_B,RCV_B,SND_S,RCV_S: channel (dy)) played_by A def=
   local State: nat, Na: text(fresh),
          Nb: text, Kb: public_key
   init State = 0
   knowledge(A) = { inv(Ka) }
   transition
      step1a.  State = 0
            /\ in(B'.Kb', KeyRing)
            /\ RCV_B(start)
          =|> State' = 2
            /\ SND_B({Na'.A}Kb')
      step1b.  State = 0
            /\ not(in(B.Kb',KeyRing))
            /\ RCV_B(start)
          =|> State' = 1
            /\ SND_S(A.B)
      step2.   State = 1
            /\ RCV_S({B.Kb'}inv(Ks))
          =|> State' = 2
            /\ KeyRing' = cons(B.Kb',KeyRing)
            /\ SND_B({Na'.A}Kb')
      step3.   State = 2
            /\ RCV_B({Na.Nb'}Ka)
          =|> State' = 3
            /\ SND_B({Nb'}Kb)
end role
```

The condition can contain comparisons, Boolean expressions over lists or sets, messages receptions. The action can contain messages sendings, assignments of variables.
A transition is a change of state, *primed variables* representing the values of the variables in the next state. So primed variables can be assigned in the right-hand side of transitions. However, if the new value of a variable is learned in the left-hand side of a transition (in a received message, in a comparison, or in a set expression, for examples), then its primed name is used (see for example step0 of role Server).

The definition of roles is based on a rich type system. Many types are available for describing protocols: agent, channel, text, message, public key, symmetric key, Boolean, integer, hash function, enumeration. Some variables of these types may be "fresh", i.e. their value is generated at running time; this happens when the primed variable appears only in the right-hand side of a transition, not assigned.
This is also possible to use type constructors: function, pair, list, set. And some algebraic operators can be used for representing cryptography properties: *xor*, *exp*.

Verification of properties. In the specification, this is possible to precise a goal section, indicating a list of properties to be checked. The supported properties are:

- secrecy: some information (keys, nonces, messages,...) have to remain secret, i.e. an intruder should not get this information;
- authentication: two roles identify each other w.r.t. an information that they send to each other; this property exists in two versions: weak authentication and strong authentication, the second one proposing a protection against the replay of a protocol.

These properties are kinds of macros, without the need to add some information in the transitions of roles.
This is however possible to specify LTL formulas that will not be interpreted by the compiler, and to add some user-defined facts in the transitions that will be carried by the compiler.

A more complete description of this high-level language and of its semantics as TLA formulas is given in [8].

3 Towards a Rule-based Program

Specifications of protocols are compiled into a rule-based program. During this compilation phase, the syntax and the semantics of the initial specification are verified. If some goals are specified, the compiler can either generate one program containing all the properties to be checked, or it can generate one program for each goal.

The generated program contains basically three parts: rules describing the intruder's behavior; rules describing role transitions and compositions[1]; the initial state, describing the instances to be considered of the protocol.

All this information is divided in several files: a prelude file containing all the protocol independent information, and at least one protocol specific file. These files represent a complete and detailed low-level specification of the initial protocol, where all variables and constants are typed: this is a rule-based program, a rule being of the form:

step rule_name (list_of_variables_involved) :=
left_hand_side => right_hand_side

The left-hand side and right-hand side of a rule are multisets of terms. So, the multiset constructor '.' is associative and commutative. This permits to handle the *non-determinism* when matching the current state of the protocol against the left-hand side of a rule. This also permits to consider the run in *parallel* of several instances of the protocol [19].

3.1 General Information

The prelude file contains all the general information necessary for obtaining a self-contained program. This information is divided into several sections that we are going to describe.

[1] Note that currently we only consider parallel compositions; sequential compositions are accepted in the hight-level specifications, but not yet converted into rules.

Type symbols. The list of type names is given in this section.

section typeSymbols:

agent, text, symmetric_key, public_key, function, table,
message, fact, nat, set, protocol_id

Signature. This section contains the subtyping information; for example, agents, keys and nonces are subtypes of messages.

section signature:

message > agent
message > nonce
message > symmetric_key
message > public_key

In addition, each primitive used for constructing messages is declared, such as:

pair : message ∗ message −> message
crypt : message ∗ message −> message
scrypt : message ∗ message −> message
inv : message −> message
apply : message ∗ message −> message

corresponding to pair construction, asymmetric and symmetric encryption of a message with a key, key inverse (this is a notation, not an applicable algorithm), and function application to a message.
More advanced primitives are also declared, such as intruder's knowledge, belonging constraints, and goal facts:

iknows : message −> fact
contains : message ∗ message −> fact
secret : message ∗ agent −> fact
witness : agent ∗ agent ∗ protocol_id ∗ message −> fact

Declaration of variables. The type of each variable used in this prelude file (see the following sections) is declared in this section.

section types:

F,K,M,M1,M2,M3 : message

Note that all of them are declared of type message for sake of generality. For example, K is used as a key, but in case of symmetric encryption it could be a compound message.

Equational properties. Protocols specifications are often based on some hypotheses over the message construction or the cryptography. This section permits to list the equational properties considered. For example, messages are built by concatenating sub-messages, forming tuples. But for a more simple representation in the rules, tuples will be represented by pairing; this choice is correct if pairing is associative.
Another example concerns the keys used for encrypting messages: given a public (resp. private) key k, its corresponding private (resp. public) key is denoted $inv(k)$; a consequence is that the inverse of the inverse of a key is the key itself.

section equations:

pair(M1,pair(M2,M3)) = pair(pair(M1,M2),M3)
inv(inv(K)) = K

For some protocols, the perfect cryptography hypothesis[2] is relaxed by describing how to generate some keys using the Diffie-Hellman exponentiation [21,9],

exp(exp(M1,M2),M3) = exp(exp(M1,M3),M2)
exp(exp(M1,M2),einv(M2)) = M1

or by describing the encryption mechanism using exclusive-or.

xor(M1,xor(M2,M3)) = xor(xor(M1,M2),M3)
xor(M1,M2) = xor(M2,M1)
xor(M,M) = 0
xor(0,M) = M

Intruder model. The intruder is described by a set of messages that it knows and by rules over this set. We describe the behavior of an intruder, following the standard Dolev-Yao model [15], independently of the protocol considered. This general behavior is first the ability to generate messages from its knowledge:

section intruder:

step gen_pair (M1,M2) :=
    iknows(M1).iknows(M2) => iknows(pair(M1,M2))
step gen_crypt (M1,M2) :=
    iknows(K).iknows(M) => iknows(crypt(K,M))
step gen_scrypt (M1,M2) :=
    iknows(K).iknows(M) => iknows(scrypt(K,M))
step gen_apply (M1,M2) :=
    iknows(F).iknows(M) => iknows(apply(F,M))

The intruder may also analyze messages in its knowledge, for trying to get new information by decomposing them, if possible:

step ana_pair (M1,M2) :=
    iknows(pair(M1,M2)) => iknows(M1).iknows(M2)
step ana_crypt (K,M) :=
    iknows(crypt(K,M)).iknows(inv(K)) => iknows(M)
step ana_scrypt (K,M) :=
    iknows(scrypt(K,M)).iknows(K) => iknows(M)

Finally, the intruder is able to generate fresh information. This is described by the following rule, where the left-hand side is empty, the right-hand side corresponds to the addition of a message M in the intruder's knowledge, and the arrow of the rule contains the information that M has to be generated at running time: its value has to be an unused value of the type of M.

step generate (M) :=
    =[*exists* M]=> iknows(M)

All this information in independent of the protocol to be considered. This independence guarantees an objective and general description of the intruder's behavior.

[2] An encrypted message can only be decrypted by the adequate key.

3.2 Protocol Information

We describe in this section how a high-level specification is translated into a rule-based program, and illustrate it with the NSPK-KS protocol.

The high-level specification of a protocol is mainly a list of roles of two kinds: basic roles, each one representing the behavior of a participant; composed roles, describing the environment of the basic roles, i.e. how to compose them and which instantiations to consider.

In the resulting program, a basic role, which is initially presented as a module, is then considered as a state. The environment roles will permit us to generate initial role states; the transitions in a basic role will describe how to change the state of that role.
What was the local environment of a basic role becomes a list of parameters of the state. However, in the current version of the compiler, only one kind of channels is considered: channel on which the Dolev-Yao model of the intruder is applied. So, for avoiding useless complex notation in the generated rules, all sent messages are directly added to the knowledge of the intruder (iknows(...)).

Signature. The generated program contains a section with the signature of each role state primitive, representing the internal data structure of the role. Note that the first argument of a role state is the name of its player. This information may be useful for tools that have to use this program, in particular if they want to manage the knowledge of the agents.

section signature:

```
state_Bob: agent * agent * public_key * public_key * set * nat * text * text
    * public_key * nat -> fact
state_Alice: agent * agent * public_key * public_key * set * nat * text * text
    * public_key * nat -> fact
state_Server: agent * public_key * set * agent * agent * public_key * nat
    -> fact
```

In those role states, natural numbers are used as labels for distinguishing steps, and they are also used for ensuring the uniqueness of agents.

Declarations. Then, all the variables and constants used in the program are declared. Note that, as in the high-level language, variables always start with a capital letter, and constants with a small letter or with a digit.

section types:

```
nb, na : protocol_id
kb, ka, ks, ki, Ka, Kb, Ks, Dummy_Ka, Dummy_Kb, dummy_pk : public_key
CID, CID2, CID1, State, 0, 1, 2, 3, 4, 5, 6 : nat
Nb, Na, Dummy_Nb, Dummy_Na, dummy_nonce : text
MGoal, KeySet, start : message
AGoal, b, a, s, A, B, S, i, Dummy_B, Dummy_A, dummy_agent : agent
Instances, KeyMap, KeyRing, local_62, local_89, local_104, local_111, local_116 : set
```

Initialization. The initialization of the protocol is put in a specific section. It contains the initial states of basic roles, obtained by flattening the composed roles; note that some parameters of those states correspond to variables that were declared locally (and not

initialized) in the roles, so they are initialized with specific constants (dummy_...). The knowledge of the intruder is also initialized in this section, using the knowledge declared for it in the composed roles, and the knowledge necessary for playing its assigned roles in the instantiations. The start message is also put in the intruder's knowledge.
A *state of the protocol* is a set of roles states and the set of knowledge of the intruder. Note that in the generated program, the intruder is never assigned as player of a state role. This is due to the Dolev-Yao intruder model: each message sent by an agent is directly added to the knowledge of the intruder; so, an agent gets a message by taking it in the intruder's knowledge; and as the intruder is able to decompose and compose messages, it can build messages it is supposed to build when playing a honest role.

```
section inits:
  initial_state init1 :=
    contains(pair(i,ki),local_62).iknows(local_62).
    iknows(ki).iknows(inv(ki)).
    iknows(ks).iknows(a).iknows(i).
    iknows(start).
    state_Server(s,ks,local_89,dummy_agent,dummy_agent,dummy_pk,2).
    state_Alice(a,b,ka,ks,local_104,0,dummy_nonce,dummy_nonce,dummy_pk,4).
    state_Bob(b,a,kb,ks,local_111,0,dummy_nonce,dummy_nonce,dummy_pk,5).
    state_Alice(a,i,ka,ks,local_116,0,dummy_nonce,dummy_nonce,dummy_pk,6).
    contains(pair(a,ka),local_89).contains(pair(b,kb),local_89).
    contains(pair(i,ki),local_89).
    contains(pair(a,ka),local_104).
    contains(pair(b,kb),local_111).
    contains(pair(a,ka),local_116)
```

The initial state is therefore a term corresponding to the set of the initial role states and the set of the intruder's initial knowledge.

In the example given above, only constants are used. However, in the high-level specification, variables can be used for describing that some information is *shared* by several role states. For example, a will play twice the role Alice, and a variable could have been used for storing its key set. The translation of this in the initialization section would have been to used only one constant instead of local_104 and local_116.

Rules. The main section of the generated program is the section of rules. Each rule correspond to a state transition for one of the basic roles. For example, in NSPK-KS, the server has only one possible transition:

```
section rules:
  step step_0 (S,Ks,KeyMap,Dummy_A,Dummy_B,Dummy_Kb,A,B,Kb,CID) :=
    state_Server(S,Ks,KeyMap,Dummy_A,Dummy_B,Dummy_Kb,CID).
    iknows(pair(A,B)).
    contains(pair(B,Kb),KeyMap)
    =>
    state_Server(S,Ks,KeyMap,A,B,Kb,CID).
    iknows(crypt(inv(Ks),pair(B,Kb)))
```

In each rule, the left-hand side contains the general pattern of the role state and the facts representing conditions for firing the transition; the awaited message has to be in the intruder's knowledge. After automatic diversion, the reply is immediately put into the intruder's knowledge. So, in the right-hand side, there is the new role state plus some facts describing new knowledge for the intruder or the modification of the value of a complex variable (a set, for example).

The translation of step1b and step2 of role Alice generates the following two rules:

```
step step_2 (A,B,Ka,Ks,KeyRing,Na,Nb,Dummy_Kb,CID) :=
    state_Alice(A,B,Ka,Ks,KeyRing,0,Na,Nb,Dummy_Kb,CID).
    iknows(start).
    not(contains(pair(B,Kb),KeyRing))
    =>
    state_Alice(A,B,Ka,Ks,KeyRing,1,Na,Nb,Dummy_Kb,CID).
    iknows(pair(A,B))

step step_3 (A,B,Ka,Ks,KeyRing,Dummy_Na,Nb,Dummy_Kb,Na,Kb,CID) :=
    state_Alice(A,B,Ka,Ks,KeyRing,1,Dummy_Na,Nb,Dummy_Kb,CID).
    iknows(crypt(inv(Ks),pair(B,Kb)))
    =[exists Na]=>
    state_Alice(A,B,Ka,Ks,KeyRing,2,Na,Nb,Kb,CID).
    iknows(crypt(Kb,pair(Na,A))).
    contains(pair(B,Kb),KeyRing).
    secret(Na,A).secret(Na,B)
```

In the last one, the nonce Na has to be created at running time (this is a fresh information). The notation =[*exists* Na]=> means that a fresh value will have to be generated each time that this rule is applied.
In that rule, there are also terms for indicating that Na is supposed to remain secret, only shared by agents A and B. This information has been added because the secrecy of Na has been required as goal property in the high-level specification.
In both transitions, the old state contains some variables named Dummy_Kb, for example. Such variables capture the old value of the corresponding variable (e.g. Kb) when either a new value will be assigned in the new state (e.g. in step_3), or when the name of this variable has to be used in the transition without considering its value (e.g. in step_2).

Goals. The last section is devoted to the description of goal properties. For example, if the secrecy of a term has to be checked, the goal section will be:

```
section goals:
  goal secrecy_of (MGoal,AGoal) :=
    secret(MGoal,AGoal).
    iknows(MGoal).
    not(secret(MGoal,i))
```

This description means that the secrecy property is not satisfied if a message MGoal, declared as a secret shared by agent AGoal, is in the intruder's knowledge, and the intruder (whose name is i) is not supposed to share it.

Similar goals are automatically generated for describing authentication properties.

4 Use of the Resulting Rule-based Programs

In this section, we list some of the protocols that we have already been able to analyze with our compiler. We also cite the tools that are using the generated rule-based programs for trying to find some attacks.
Note that in this paper, we have not illustrated our compiler with one of the industrial security protocols cited in the following because most of them are complex, involving many roles. The NSPK-KS protocol has the advantage of being based on a protocol known by everybody, but this variant is an excellent example for illustrating the importance of the environment of each agent, and of the control that has to be set.
We also do not list the standard toy protocols of the Clark-Jacob library [11]; they can of course be considered by our compiler.

4.1 Already Handled Protocols

In order to assess the compilation process, we have built a list of protocols from various sources. The aim here is to demonstrate that our compiler is able to handle a wide variety of protocols, known as important in different areas. We have selected both low-level protocols, such as TLS, as well as high-level ones, such as UMTS-AKA. There are also protocols already recommended by the IETF as well as protocols still under work.

To handle these protocols does not mean just to be able to specify them in our high-level language. What we have detailed in the previous sections is that we have defined a methodology for analyzing such specifications and for translating them into low-level specifications. All the subsequently given protocols have been automatically compiled, their specification being as verbatim as possible from their informal definition.

Core security mechanism [6]. Transport Layer Security (TLS) [14]. This two-part protocol aims at providing low-level privacy and data integrity. We have currently modeled and compiled the TLS record protocol.

Useful but not core mechanism [6]. Kerberos, ChapV2. The first one is well-spread and well-known. ChapV2 is an extension by Microsoft of the CHAP protocol used for PPP authentication.

Authentication mechanism for the Internet. In the survey [22], several protocols were recommended for authentication with password over the Internet. Among these, we have already analyzed the protocols EKE, EKE2, SPEKE and the SRP protocols. Among them, the *EKE protocol family is a family of zero-knowledge protocols for authentication on a password. We have also successfully analyzed the UMTS-AKA (Authentication and Key Agreement UMTS) protocol. The SRP protocol was designed by *Siemens* and presented at the IETF.

Protocols under development. We also work, in conjunction with Siemens, on the analysis of protocols under development. Among these, we have already analyzed the AAA MobileIP protocol, which is a sub-protocol of Mobile IP. See [16] for a description of the protocol its goal. Note that the Mobile IP protocol is a collection of protocols in development since 1998. We hope to contribute to this development by speeding up part of its analysis. We have also analyzed the IKEv2 main protocol.

Other sources. Finally, we have considered protocols from different sources, such as the ISO/IEC public key protocols. We have also analyzed the two party signature RSA protocol [5].

4.2 Verifying the Rule-based Programs

The programs that have been generated for the protocols listed in the previous section have been studied by several tools:

- OFMC [4]: an on-the-fly model checker developed at ETH, Zürich;
- SATMC [3]: a SAT-based model checker developed at DIST, Genova;
- CL-Atse [24]: a constraint logic-based protocol analyzer developed at LORIA, Nancy.

The first results obtained are still preliminary ones, but some attacks have been found on several protocols, some of them being new ones (see [8]).

This connection of three very different tools, done for the AVISPA project, demonstrates the flexibility and expressiveness of the rule-based specifications of protocols that we generate automatically.

5 Conclusion

We have presented in this paper a low-level framework for expressing protocol specifications. The security protocols are initially described with a simple, flexible, modular, non-ambiguous and expressive high-level language [8]. The generated specifications are rule-based programs with very detailed information: a full typing of variables, constants and primitives; a precise description of role transitions and of the initial state; the independent description of the intruder's model.

The generated rules permit to consider both the parallelism of the agents, and the non-determinism when applying a rule [19].

The compiler described is written in OCaml and documented with OCamlWeb (140 pages). It has been defined for the AVISPA project, for considering real industrial Internet security protocols. The first experiments generate very good results, so we are going to continue its development for being able to handle even more protocols. This is a considerable improvement compared to the first compiler that was realized for the AVISS project [2], where only simple protocols could be considered [11].

The only other known attempt to go beyond Alice&Bob notation in a clear high-level language is the MuCAPSL-MuCil translator [13]. Initial specifications are written in MuCAPSL [20], a new version of the CAPSL language [7] dedicated to the specification of group communication protocols. While CAPSL was an Alice&Bob notation language, MuCAPSL is completely different: it is based on roles, and provides a large scale of primitives and data structures; each role is typed (a type contains some attributes, functions, ...) and contains a sequence of instructions, including `DO...UNTIL` loops.
Comparing MuCAPSL with our specification language, this is clear that MuCAPSL offers many more data structures, primitives and instructions. However, it is unclear how to specify that some roles share some information, or how many instances of a role are created, since there is no environment role and roles do not have parameters. In addition, with our language, the transitions are guarded and are not ordered: their application is not deterministic, and transitions can be applied several times if possible; this is not the case with

MuCAPSL. So, this is not clear how the example given in this paper, NSPK Key Server, could be specified in MuCAPSL. About properties to check, MuCAPSL, like our language, proposes secrecy and authentication.
The MuCAPSL-MuCIL translator generates a specification in MuCIL that contains the declaration of the used symbols, and a multiset of conditional rewriting rules.

For concluding, MuCAPSL-MuCIL and our compiler have both their own advantages, and if our specification language is more simple, this is because we have designed it with the objective to provide it to industrial partners. It is powerful enough for specifying rather easily most of the Internet security protocols. And the aim of our compiler is to provide rule-based specifications of protocols to industrials for helping them to implement the protocols that they design, and also for verifying those protocols, by plugging any kind of verification tool, as it has already been done with AVISPA for three very different tools.

Acknowledgements: we thank the referees for their relevant remarks that have helped us to improve this paper.

References

1. M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.
2. A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In E. Brinksma and K. Guldstrand Larsen, editors, *Computer-Aided Verification, CAV'02*, LNCS 2404, pages 349–354, Heidelberg, 2002. Springer. URL of the AVISS and AVISPA projects: `http://www.avispa-project.org`.
3. A. Armando and L. Compagna. Abstraction-driven SAT-based Analysis of Security Protocols. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, LNCS 2919, pages 257–271, Santa Margherita Ligure, Italy, May 2003. Springer.
4. D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In Einar Snekkenes and Dieter Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer, 2003.
5. M. Bellare and R. Sandhu. The security of a family of two-party RSA signature schemes. Technical Report 2001/060, Cryptography ePrint Archive, 2001. `http://eprint.iacr.org/2001/060/`.
6. S. Belovin. Report of the iab security architecture workshop, April 1998.
7. Common Authentication Protocol Specification Language. `http://www.csl.sri.com/~millen/capsl/`.
8. Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. An high level protocol specification language suited for industrial security-sensitive protocols. Technical report, DIST, Genova, Italy, 2004. Submitted.
9. Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the Security of Protocols with Diffie-Hellman Exponentiation and Products in Exponents. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science, FST TCS'03*, LNCS 2914, page ??? Springer, December 2003.
10. Y. Chevalier and L. Vigneron. Strategy for Verifying Security Protocols with Unbounded Message Size. *Journal of Automated Software Engineering*, 11(2):141–166, April 2004.
11. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.
12. G. Delzanno and P. Ganty. A Survey on the State-of-the-art Methods for the Automatic Verification of Security Protocols. In *Proc. 1st Workshop on Issues in Security and Petri Nets (WISP)*, Eindoven, The Netherlands, 2003.
13. G. Denker and J. Millen. Modeling Group Communication Protocols using Multiset Term Rewriting. In *4th Int. Workshop on Rewriting Logic and its Applications (WRLA 2002)*, Pisa, Italy, 2002. (invited talk). ENTCS vol. 71 (2003), Elsevier Science Publishers.
14. T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, January 1999. Status: Proposed Standard.

15. D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
16. S. Glass, T. Hiller, S. Jacobs, and S. Perkins. RFC 2977: Mobile IP Authentication, Authorization, and Accounting Requirements, October 2000. Status: Informational.
17. G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
18. IETF: The Internet Engineering Task Force. `http://www.ietf.org`.
19. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR 2000*, LNCS 1955, pages 131–160. Springer, 2000.
20. J. Millen and G. Denker. MuCAPSL. In *DISCEX III, DARPA Information Survivability Conference and Exposition*, pages 238–249. IEEE Computer Society, 2003.
21. J. K. Millen and V. Shmatikov. Symbolic Protocol Analysis with Products and Diffie-Hellman Exponentiation. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW'03)*, pages 47–61. IEEE Computer Society Press, 2003.
22. E. Rescorla. A Survey of Authentication Mechanisms, October 2003. Draft. `http://www.ietf.org/internet-drafts/draft-iab-auth-mech-03.txt`.
23. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop CSFW'95*, pages 98–107. IEEE Computer Society Press, 1995.
24. M. Turuani. *Sécurité des Protocoles Cryptographiques : Décidabilité et Complexité*. Phd thesis, Université Henri Poincaré – Nancy 1, December 2003.

Principles of Chemical Programming

J.-P. Banâtre[1], P. Fradet[2] and Y. Radenac[1]

[1] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{jbanatre,yradenac}@irisa.fr
[2] INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 Montbonnot, France
Pascal.Fradet@inria.fr

Abstract The chemical reaction metaphor describes computation in terms of a chemical solution in which molecules interact freely according to reaction rules. Chemical models use the multiset as their basic data structure. Computation proceeds by rewritings of the multiset which consume elements according to reaction conditions and produce new elements according to specific transformation rules. Since the introduction of Gamma in the mid-eighties, many other chemical formalisms have been proposed such as the CHAM, the P-systems and various higher-order extensions. The main objective of this paper is to identify a basic calculus containing the very essence of the chemical paradigm and from which extensions can be derived and compared to existing chemical models.

1 Introduction

The chemical reaction metaphor has been discussed in various occasions in the literature. This metaphor describes computation in terms of a chemical solution in which molecules (representing data) interact freely according to reaction rules. Chemical models use the multiset as their basic data structure. Computation proceeds by rewritings of the multiset which consume elements according to reaction conditions and produce new elements according to specific transformation rules.

To the best of our knowledge, the Gamma formalism was the first "chemical model of computation" proposed as early as in 1986 [2] and later extended in [3]. A Gamma program is a collection of reaction rules acting on a multiset of basic elements. A reaction rule is made of a condition and an action. Execution proceeds by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable state is reached that is to say when no more reactions can take place. Figure 1 gives three short examples illustrating the Gamma style of programming. The reaction *max* computes the maximum element of a non empty set. The reaction replaces

$$\begin{array}{ll} max & = \textbf{replace } x, y \textbf{ by } x \textbf{ if } x > y \\ primes & = \textbf{replace } x, y \textbf{ by } y \textbf{ if } multiple(x, y) \\ maj & = \textbf{replace } x, y \textbf{ by} \{\} \textbf{ if } x \neq y \end{array}$$

Figure 1. Examples of Gamma programs

any couple of elements x and y such that $x > y$ by x. This process goes on till a stable state is reached, that is to say, when only the maximum element remains. The reaction *primes* computes the prime numbers lower or equal to a given number N when applied to the multiset of all numbers between 2 and N ($multiple(x, y)$ is true if and only if x is multiple of y). The majority element of a multiset is an element which occurs more than $card(M)/2$ times in the multiset. Assuming that such an element exists, the reaction *maj* yields a multiset which only contains instances of the majority element just by removing pairs of distinct elements. Let us emphasize the conciseness and elegance of these programs.

Nothing had to be said about the order of evaluation of the reactions. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

Gamma makes it possible to express programs without artificial sequentiality. By artificial, we mean sequentiality only imposed by the computation model and unrelated to the logic of the program. This allows the programmer to describe programs in a very abstract way. In some sense, one can say that Gamma programs express the very idea of an algorithm without any unnecessary linguistic idiosyncrasies. The interested reader may find in [3] a long series of examples (string processing problems, graph problems, geometry problems, ...) illustrating the Gamma style of programming and in [1] a review of contributions related to the chemical reaction model.

Later, the idea was developed further into the CHAM [4], higher-order multiset rewriting [13], the hmm-calculus [8], the P-systems [14], etc. Although built on the same basic paradigm, these proposals have different properties and different expressive powers. This article is an attempt to identify the basic principles behind chemical models.

In Section 2, we exhibit a minimal chemical calculus, from which all other "chemical models" can be obtained by addition of well-chosen features. Basically, this minimal calculus, incorporates the γ-reduction which expresses the very essence of the chemical reaction and the associativity and commutativity rules which express the basic properties of chemical solutions. This calculus is then enriched in Section 3 with conditional reactions and, further, the possibility of rewriting atomically several molecules. These extensions give rise to four possible chemical calculi. Section 4 shows how existing chemical models relate and compare to the basic calculi presented previously. Section 5 suggests several research directions and concludes.

2 A minimal chemical calculus

In this section, we introduce a higher-order calculus, the γ_0-*calculus*, that can be seen as a formal and minimal basis for the chemical paradigm (in much the same way as the λ-calculus is the formal basis of the functional paradigm).

2.1 Syntax

The fundamental data structure of the γ_0-calculus is the multiset (a collection which may contain several copies of the same element). Elements can move freely inside the multiset and react together to produce new elements. Computation can be seen either intuitively, as chemical reactions of elements agitated by Brownian motion, or formally, as higher-order associative and commutative (AC) rewritings of multisets.

The syntax of γ_0-terms (also called *molecules*) is given in Figure 2. A γ-abstraction is

$$
\begin{array}{rcll}
M & ::= & x & ;\ \textit{variable} \\
 & | & (\gamma\langle x\rangle.M) & ;\ \gamma\textit{-abstraction} \\
 & | & (M_1, M_2) & ;\ \textit{multiset} \\
 & | & \langle M\rangle & ;\ \textit{solution}
\end{array}
$$

Figure 2. Syntax of γ_0-molecules

a reactive molecule which consumes a molecule (its argument) and produces a new one (its body). Molecules are composed using the AC multiset constructor ",". A solution encapsulates molecules (*e.g.,* multiset) and keeps them separate. It serves to control and isolate reactions. To avoid notational clutter, we omit outermost parentheses, parentheses in multisets and we assume that γ-abstractions associate to the right. For example, the γ-abstraction $(\gamma\langle x\rangle.(x,(x,(\gamma\langle y\rangle.y))))$ will be written $\gamma\langle x\rangle.x, x, \gamma\langle y\rangle.y$.

The γ_0-calculus bears clear similarities with the λ-calculus. They both rely on the notions of (free and bound) variable, abstraction and application. A λ-abstraction and a γ-abstraction both specify a higher-order rewrite rule. However, λ-terms are tree-like whereas the AC nature of the application operator "," makes γ_0-terms multiset-like. Associativity and commutativity formalizes Brownian motion and make the notion of solution necessary, if only to distinguish between a function and its argument.

2.2 Semantics

The conversion rules and the reduction rule of the γ_0-calculus are gathered in Figure 3. Chemical reactions are represented by a single rewrite rule, the γ-reduction, which applies

$$
\begin{array}{llll}
(\gamma\langle x\rangle.M),\langle N\rangle & \longrightarrow_\gamma \; M[x:=N] & \text{if } \mathit{Inert}(N) \vee \mathit{Hidden}(x,M) & ;\ \gamma\text{-}\mathit{reduction} \\
\gamma\langle x\rangle.M & \equiv \;\; \gamma\langle y\rangle.M[x:=y] & \text{with } y \text{ fresh} & ;\ \alpha\text{-}\mathit{conversion} \\
M_1, M_2 & \equiv \;\; M_2, M_1 & & ;\ \mathit{commutativity} \\
M_1,(M_2,M_3) & \equiv \;\; (M_1,M_2),M_3 & & ;\ \mathit{associativity}
\end{array}
$$

Figure 3. Rules of the γ_0-calculus

a γ-abstraction to a solution. A molecule $(\gamma\langle x\rangle.M),\langle N\rangle$ can be reduced only if

Inert(N): the content N of the solution argument is a closed term made exclusively of γ-abstractions or exclusively of solutions (which may be active),

or *Hidden*(x,M): the variable x occurs in M only as $\langle x\rangle$. Therefore $\langle N\rangle$ can be active since no access is done to its contents.

So, a molecule can be extracted from its enclosing solution only when it has reached an inert state. This is an important restriction that permits the ordering of rewritings. Without this restriction, the contents of a solution could be extracted in any state and the solution construct would lose its purpose.

Consider, for example, the following molecules:

$$
\begin{array}{l}
\omega \equiv \gamma\langle x\rangle.x,\langle x\rangle \\
\Omega \equiv \omega,\langle\omega\rangle \\
I \;\equiv \gamma\langle x\rangle.\langle x\rangle
\end{array}
$$

Clearly, Ω is an always active (non terminating) molecule and I an inert molecule (the identity function in normal form). The molecule $\langle\Omega\rangle,\langle I\rangle,\gamma\langle x\rangle.\gamma\langle y\rangle.x$ reduces as follows:

$$\langle\Omega\rangle,\langle I\rangle,\gamma\langle x\rangle.\gamma\langle y\rangle.x \longrightarrow \langle\Omega\rangle,\gamma\langle y\rangle.I \longrightarrow I$$

The first reduction is the only one possible: the γ-abstraction extracts x from its solution and $\langle I\rangle$ is the only inert molecule ($\mathit{Inert}(I) \wedge \neg\mathit{Hidden}(x,\gamma\langle y\rangle.x)$). The second reduction is possible only because the active solution $\langle\Omega\rangle$ is not extracted but removed $(\neg Inert(\Omega) \wedge Hidden(y, I))$

A molecule is in normal form if all its molecules are inert. We say that two molecules M_1 and M_2 are syntactically equivalent (and we write $M_1 \equiv M_2$), if they can be rewritten into each other using α-conversion and AC rules.

As usual, rules can be applied in parallel as long as they apply to disjoint redexes. So, there can be several reactions at the same time for disjoint sub-terms and/or within nested solutions and/or inside γ-abstractions.

2.3 Expressivity

The γ_0-calculus is more expressive than the λ-calculus since it can easily express non-deterministic programs. For example, let A and B two distinct normal forms:

$$\begin{array}{ccc}
(\gamma\langle x\rangle.\gamma\langle y\rangle.x), \langle A\rangle, \langle B\rangle & \equiv & (\gamma\langle x\rangle.\gamma\langle y\rangle.x), \langle B\rangle, \langle A\rangle \\
\downarrow_\gamma & & \downarrow_\gamma \\
(\gamma\langle y\rangle.A), \langle B\rangle & & (\gamma\langle y\rangle.B), \langle A\rangle \\
\downarrow_\gamma & & \downarrow_\gamma \\
A & \not\equiv & B
\end{array}$$

On the other hand, the λ-calculus can easily be encoded within the γ_0-calculus. Figure 4 gives here a possible encoding for the strict λ-calculus using the function $[\![\cdot]\!]$ which takes a λ-term and returns its translation as a γ-term. The standard call-by-name λ-calculus

$$\begin{array}{lcl}
[\![x]\!] & \stackrel{\text{def}}{=} & x \\
[\![\lambda x.E]\!] & \stackrel{\text{def}}{=} & \gamma\langle x\rangle.[\![E]\!] \\
[\![E_1\ E_2]\!] & \stackrel{\text{def}}{=} & \langle[\![E_1]\!]\rangle,\ \gamma\langle f\rangle.f, \langle[\![E_2]\!]\rangle
\end{array}$$

Figure 4. Translating λ-terms into γ_0-molecules

can also be encoded but the translation is slightly more involved. This comes from the strict nature of the γ_0-calculus which enforces the argument to be inert/reduced before the reaction can take place.

As in the λ-calculus, recursion, integers, booleans, data structures, arithmetic, logical and comparison operators can be defined within the γ_0-calculus. We do not give their precise definitions in this article since they are similar to their definitions as λ-terms. From now on, we will give our examples assuming that these constructs have been defined. In particular, we will use pairs (written $a{:}b$) and recursive definitions to define n-shot abstractions (which re-introduce themselves after each reaction). For example, the molecule performing the product of all integers in its solution can be defined as:

$$pi = \gamma\langle x\rangle.\gamma\langle y\rangle.\langle x * y\rangle, pi$$

The reactive molecule pi takes two integers and replaces them by their product and a copy of itself. For example:

$$pi, \langle 2\rangle, \langle 3\rangle, \langle 2\rangle \longrightarrow_\gamma \ldots \longrightarrow_\gamma pi, \langle 12\rangle \longrightarrow_\gamma \gamma\langle y\rangle.\langle 12 * y\rangle, pi$$

3 Two fundamental extensions

The γ_0-calculus is a quite expressive higher-order calculus. However, compared to the original Gamma [3] and other chemical models [8,13,14], it lacks two fundamental features:

- *Reaction condition.* In Gamma, reactions are guarded by a condition that must be fulfilled in order to apply them. Compared to γ_0 where inertia and termination are described syntactically, conditional reactions give these notions a semantic nature.
- *Atomic capture.* In Gamma, any fixed number of elements can take part in a reaction. Compared to a γ_0-abstraction which reacts with one element at a time, a n-ary reaction takes atomically n elements which cannot take part in any other reaction at the same time.

These two extensions are orthogonal and enhance greatly the expressivity of chemical calculi. Strictly speaking, these features do not permit to express a larger class of programs (the γ_0-calculus is Turing-complete). But, they do add expressivity in the sense that they are not syntactic sugar and can only be expressed using a global re-organization of programs.

3.1 Conditional reaction

In the γ_c-calculus, abstractions hold conditions. The condition of an abstraction must be satisfied before the reaction occurs. The syntax of the γ_c molecules is given in Figure 5. The reaction condition is modeled by M_0 which must evaluate to a special constant **true**

$$\begin{array}{rll} M ::= & x & \textit{; variable} \\ \mid & \gamma\langle x\rangle\lfloor M_0\rfloor.M_1 & \textit{; conditional } \gamma\textit{-abstraction} \\ \mid & (M_1, M_2) & \textit{; multiset} \\ \mid & \langle M\rangle & \textit{; solution} \end{array}$$

Figure 5. Syntax of γ_c-molecules

before the reaction occurs. The γ_c-reduction is formalized as follows:

$$\frac{\mathit{Inert}(N) \vee \mathit{Hidden}(x,(M_0,M_1)) \qquad M_0[x := N] \xrightarrow{*}_c \mathbf{true}}{(\gamma\langle x\rangle\lfloor M_0\rfloor.M_1),\ \langle N\rangle \longrightarrow_c M_1[x := N]}$$

where the molecule **true** is a given constant ($e.g.,\ \gamma\langle x\rangle\lfloor x\rfloor.x$).

Clearly, the γ_c-calculus embeds the γ_0-calculus: the abstractions of γ_0 correspond to the abstractions of γ_c with the condition **true**. Inert γ_c-molecules are molecules where no solution satisfies the reaction condition of any γ-abstraction. So, as opposed to γ_0, inert molecules in the γ_c-calculus can mix solutions and abstractions as long as no condition is satisfied. Inertia, as well as termination, becomes a semantic notion.

Consider the task of ceiling a collection of integers by 9. This can be expressed in γ_c by the following recursive molecule:

$$\mathit{ceil} = \gamma\langle x\rangle\lfloor x > 9\rfloor.\langle 9\rangle,\ \mathit{ceil}$$

and, for example,

$$\mathit{ceil},\ \langle 10\rangle,\ \langle 3\rangle,\ \langle 11\rangle \xrightarrow{*}_\gamma \mathit{ceil},\ \langle 9\rangle,\ \langle 3\rangle,\ \langle 9\rangle$$

Conditions can be used to encode type checking and pattern-matching. For example, assuming pairs $(x{:}y)$ (with the access functions Fst and Snd) and a tag (constant) Int, we can encode typed integers by $Int{:}x$. A γ-abstraction matching an integer as argument can be written as:

$$\gamma\langle i\rangle\lfloor Fst\ i = Int \wedge M_0\rfloor.M_1$$

It is easy to define a convenient and expressive pattern language to match integers, booleans, solutions, γ-abstractions, etc. For example, the γ-abstraction above could also be written:

$$\gamma(Int{:}x)\lfloor M_0\rfloor.M_1$$

There is no simple and local way to encode γ_c in γ_0. The encoding implies a global reorganization of γ_0 programs. A possible encoding consists in a γ_0 program interpreting γ_c programs. All γ_c elements are isolated in solutions with their description (type, value) and γ_0-abstractions simulate the semantics of γ_c (this can be done since γ_0 is Turing-complete). The algorithm must check that the reaction condition of all γ-abstractions is false before it terminates.

3.2 Atomic capture

In the γ_n-calculus, abstractions can capture several elements atomically. The syntax of the molecules is given in Figure 6. The n-ary abstraction can occur if it finds n solutions, otherwise no reaction takes place. Of course, an element cannot participate in several reactions simultaneously (mutual exclusion). The γ_c-reduction is formalized as follows:

$$\begin{array}{rll} M ::= & x & ;\ \textit{variable} \\ \mid & \gamma(\langle x_1\rangle,\ldots,\langle x_n\rangle).M & ;\ \textit{n-ary } \gamma\textit{-abstraction} \\ \mid & (M_1, M_2) & ;\ \textit{multiset} \\ \mid & \langle M\rangle & ;\ \textit{solution} \end{array}$$

Figure 6. Syntax of γ_n-molecules

$$\frac{\forall\ 1 \le i \le n\ \ Inert(N_i) \vee Hidden(x_i, M)}{(\gamma(\langle x_1\rangle,\ldots,\langle x_n\rangle).M),\ \langle N_1\rangle,\ldots,\langle N_n\rangle \longrightarrow_n M[x_i := N_i]}$$

For example, the addition and product of a collection of integers can be defined as binary recursive γ-abstractions:

$$\begin{aligned} sigma &= \gamma(\langle x\rangle,\langle y\rangle).\langle x+y\rangle,\ sigma \\ pi &= \gamma(\langle x\rangle,\langle y\rangle).\langle x*y\rangle,\ pi \end{aligned}$$

The following example describes one possible execution where one addition and one multiplication have been performed (many other executions are possible):

$$sigma,\ pi,\ \langle 2\rangle,\ \langle 3\rangle,\ \langle 4\rangle\ \xrightarrow{*}_n\ \langle 20\rangle,\ sigma,\ pi$$

Consider the previous example but with $sigma$ and pi defined as unary γ_0-abstractions. When there remains only two elements, $sigma$ and pi could each take one element and would keep waiting for a second. These conflicts (which can also be seen as deadlocks) can only be avoided with the ability of taking several elements atomically.

This feature is not syntactic sugar. As with γ_c, a possible way to encode the atomic capture, is to isolate all γ_n elements in solutions and to emulate using γ_0 reactions the semantics of γ_n. Abstractions of γ_n are encoded with their arity and the emulator should test the presence of enough elements before triggering the reaction.

4 Chemical calculi and related chemical models

The previous extensions are orthogonal and can be combined. For example, the γ_0-calculus can be extended using reaction conditions and atomicity capture. We denote the resulting calculus γ_{cn}. The γ-calculi are depicted in Figure 7 where arrows stand for "can be extended into" or "is less expressive than". In the following, we relate well-known chemical models to γ_0 and γ_{cn}.

$$\begin{tikzcd}
& \gamma_{cn} & \\
\gamma_c \arrow[ur] & & \gamma_n \arrow[ul] \\
& \gamma_0 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 7. γ-calculi.

4.1 The γ_0-calculus and related models

Our minimal chemical calculus is quite close to Berry and Boudol's concurrent λ-calculus (referred to here as the γ_{tb}-calculus) introduced after the chemical abstract machine (CHAM) in [4]. The γ_{tb}-calculus relies also on variables, abstractions, an AC application operator and solutions. However, to distinguish between the γ-abstraction and its argument, it adds the notion of positive ions (denoted M^+). The γ-abstractions are negative ions (denoted $x^- M$) which can react only with positive ions:

$$\beta\text{-reaction:} \;\; (x^- M), N^+ \to M[x := N]$$

In fact, no reaction can occur within a positive ion and so arguments are passed unchanged to abstractions. Furthermore, an additional reduction law, the *hatching rule*, extracts an inert molecule M from a solution $\langle M \rangle$:

$$\text{hatching:} \;\; \langle W \rangle \rightleftharpoons W \qquad \text{if } W \text{ is inert}$$

In the γ_0-calculus, these two notions are replaced by the strict γ-reduction. In particular, hatching can be written explicitly as

$$(\gamma\langle x \rangle.x), \langle M \rangle$$

which extracts M from its solution when it becomes inert. Even if the γ_0-calculus looks simpler than the γ_{tb}-calculus, it seems that they cannot be translated easily into each other (*e.g.,* by a translation defined on the syntax rules). They appear to be call-by-value (γ_0) and call-by-name (γ_{tb}) versions of similar ideas.

4.2 The γ_{cn}-calculus and related models

In the γ_{cn}-calculus, abstractions have a reaction condition and the ability to take several molecules atomically. Their syntax becomes:

$$\gamma(\langle x_1\rangle, \ldots, \langle x_n\rangle)\lfloor M_0\rfloor.M_1$$

The associated reduction rule mixes γ_c-reduction and γ_n-reduction :

$$\frac{\forall\ 1 \leq i \leq n \quad \mathit{Inert}(N_i) \vee \mathit{Hidden}(x_i, (C, M)) \quad C[x_i := N_i] \xrightarrow{*}_{cn} \mathbf{true}}{(\gamma(\langle x_1\rangle, \ldots, \langle x_n\rangle)\lfloor C\rfloor.M), \langle N_1\rangle, \ldots, \langle N_n\rangle \longrightarrow_{cn} M[x_i := N_i]}$$

The γ_{cn} model cumulates the expressive power of γ_c and γ_n. For example, the dining philosophers problem can be expressed in γ_{cn} as follows:

$$\begin{aligned} \mathit{Eat} &= \gamma(\langle \mathit{Fork}{:}f_1\rangle, \langle \mathit{Fork}{:}f_2\rangle)\lfloor f_2 = f_1 + 1 \ \mathit{mod}\ N\rfloor.\langle \mathit{Phi}{:}f_1\rangle, \mathit{Eat} \\ \mathit{Think} &= \gamma(\langle \mathit{Phi}{:}f\rangle)\lfloor \mathbf{true}\rfloor.\langle \mathit{Fork}{:}f\rangle, \langle \mathit{Fork}{:}(f + 1 \ \mathit{mod}\ N)\rangle, \mathit{Think} \end{aligned}$$

Initially the multiset contains only forks and the two recursive molecules. The Eat reaction looks for two adjacent forks $\langle \mathit{Fork}{:}f_i\rangle$ and "produces" an eating philosopher $\langle \mathit{Phi}{:}f\rangle$. This reaction needs the expressive powers of γ_n and γ_c: the two forks have to be adjacent (reaction condition of γ_c) and should be taken simultaneously (atomicity of γ_n) to prevent deadlocks. The Think reaction "transforms" an eating philosopher into two available forks.

Most of the existing chemical models have reaction conditions and the ability to take several molecules atomically. They are closely related to γ_{cn} even when they are first-order languages. We present here two first-order models (Gamma and the CHAM) and higher-order extensions (higher-order multiset rewriting and the hmm-calculus).

Gamma To the best of our knowledge, Gamma [2,3] is the first chemical model. It consists in a single multiset containing basic inactive molecules and external, conditional and n-ary reactions. Reactions are n-shot: they are applied until no reaction can take place. They are first-order: they are not part of the multiset and cannot be taken as argument or returned as result. Moreover, there is no nested solutions. Even if sub-solutions can be encoded, there is no notion of inertia in Gamma (only global termination). A standard Gamma program is easily expressed as a γ_{cn}-molecule made of a solution (inert because without any abstraction) representing the multiset and a collection of recursive γ-abstractions representing the reactions. Gamma has inspired many extensions (*e.g.,* composition operators [12]) and other chemical models. Most of these extensions and models remain related to γ_{cn}.

The Chemical Abstract Machines The chemical abstract machine [4] (CHAM) is a chemical approach introduced to describe concurrent computations without explicit control. It started from Gamma and added many features such as membranes, (sub)solutions, inertia and airlocks. Like Gamma, reactions are n-ary and n-shot rewrite rules which are not part of the multisets. The selection pattern in the left-hand side of rewrite rules can include constants which is a form of reaction condition. For example, in [4], the description of the operational semantics of the TCCS and CCS calculi contains a cleanup rule ($0 \rightharpoonup$) which removes molecules equal to 0. The CHAM would be equivalent to γ_{cn} if it was higher-order.

Higher-order extensions A first higher-order extension of Gamma has been proposed in [13]. The definition of Gamma involves two different kinds of terms: the program (set of rewrite rules) and multisets. The main extension of higher-order Gamma consists in unifying these two categories of expression into a single notion of configuration. A configuration contains a program and a list of named multisets. It is denoted by $[Prog,\ Var_1 = Multiset_1, \ldots, Var_n = Multiset_n]$. The program $Prog$ is a rewrite rule of the multisets (named Var_i) of the configuration. This model is an higher-order model because any configuration can handle other configurations through their program. It includes reaction conditions and n-ary rewrite rules. However, reactions are not first-class citizens since they are kept separate from multisets of data.

The hmm-calculus [8] (for *higher-order multiset machines*) is described as an extension of Gamma where reactions are one-shot and first-class citizens. An abstraction denoted by $\lambda\tilde{x}.M_1 \Leftarrow M_0$ describes a reaction rule: it takes several terms denoted by a tuple $\tilde{x}$, the term M_1 is the action and the term M_0 is the reaction condition. Like γ_{lb}, the hmm-calculus uses a call-by-name strategy. It needs an hatching rule to extract an inert molecule from its solution. Any reaction can occur within solutions and within abstractions. The hmm-calculus can be seen as a lazy version of the γ_{cn}-calculus, or as an extension of the γ_{lb}-calculus with conditional and n-ary reactions.

P-systems P-systems [14] are computing devices inspired from biology. It consists in nested membranes in which molecules react. Molecules can cross and move between membranes. A set of partially ordered rewrite rules is associated to each membrane. These rules describe possible reactions and communications between membranes of molecules. These features can be expressed in γ_{cn} by introducing two new notions. They do not add additional expressive power but they are convenient and interesting in themselves.

- The first needed notion is *universally quantified conditions*. Intuitively, a reaction condition C can be read "if it exists a solution that satisfies C …". Another kind of condition could also be considered: "if all solutions satisfy C …". This universally quantified condition can be expressed in γ_c. It amounts to testing the absence of a molecule satisfying $\neg C$. Using this mechanism, it is possible to specify a partial order between reactions as priorities. A high priority reaction should react before one with a lower priority. To encode priority, an abstraction should check that no abstraction with a higher priority can react, *i.e.,* that there is *no* elements in the solution that satisfy the conditions of the abstractions with a higher priority.
- The second notion is *porous solutions*. It is possible to define porous solutions which can be manipulated by γ-abstractions even when they are active. A porous solution made of the active molecule $X_1, \ldots, X_n$ can be encoded by $\langle \gamma\langle x \rangle.X_1, \ldots, X_n \rangle$. The body of the γ-abstraction is active but can be accessed by extracting it from its inert enclosing solution and by applying it to an argument. This feature can be used to represents the porous membranes of P-systems. This capability is also useful for example when modeling non-terminating reactive systems which interacts continuously with their environment. The reactive system is therefore represented by an always active porous solution and the environment by reactions adding (sending) and removing (receiving) elements in that solution.

Other models Our list of comparisons is not exhaustive and other models could have been considered. For example, Linda and its variants (particularly Bauhaus Linda [7]) are close to Gamma. Other work has been carried out about concurrent λ-calculus according to a chemical metaphor such as [11], or, for example, models from [9].

5 Conclusion

In this article, we have studied the fundamental features of the chemical programming paradigm. The γ_0-calculus embodies the essential characteristics (AC multiset rewritings) in only four syntax rules. Terms are multisets (built with the AC application operator ",") which can be nested (inside solutions). This minimal calculus has been shown to be expressive enough to express the λ-calculus and a large class of non-deterministic programs. However, it does not reflect closely existing chemical languages such as Gamma. Two extensions must be considered to achieve a comparable expressive power: reaction conditions and atomic capture. With appropriate syntactic sugar (recursion, constants, operators, pattern-matching, porous solutions, etc.), the γ_{cn}-calculus closely models most of the existing chemical programming models.

This work suggests several research directions. First, we should prove formally that our extensions really improve the expressive power of our minimal chemical calculus. The comparison of the expressive power of languages has been formally studied by Felleisen in [10]. He formalizes the intuitive notions of "syntactic sugar" and "expressive power". A new construct is considered as enhancing expressivity if its expression using the other constructs needs "a global reorganization of the entire program". A formal comparison of expressive power of different coordination languages has been carried out in [5]. This work compares different variants of Linda [6] with different models *à la* Gamma and with models featuring communication transactions. A similar approach could be taken to establish formally the pre-order of Figure 7. Our work could also be completed by providing formal translations of existing chemical models into the corresponding γ-calculus.

Another direction is to propose a realistic higher-order chemical programming language based on the γ_{cn}-calculus. It would consist in defining the already mentioned syntactic sugar, a type system, as well as expressive pattern and module languages.

References

1. Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In *Multiset Processing*, volume 2235 of *LNCS*, pages 17–44. Springer-Verlag, 2001.
2. Jean-Pierre Banâtre and Daniel Le Métayer. A new computational model and its discipline of programming. Technical Report RR0566, INRIA, September 1986.
3. Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM (CACM)*, 36(1):98–111, January 1993.
4. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
5. Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of coordination models. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594 of *LNCS*, pages 134–149, 1999.
6. Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
7. Nicholas Carriero, David Gelernter, and Lenore Zuck. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 66–76. Springer-Verlag, jul 1994.
8. David Cohen and Juarez Muylaert-Filho. Introducing a calculus for higher-order multiset programming. In *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 124–141, April 1996.
9. Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries – a review. *Artificial Life*, 7(3):225–275, 2001.

10. Matthias Felleisen. On the expressive power of programming languages. In *3rd European Symposium on Programming, ESOP'90*, volume 432, pages 134–151. Springer-Verlag, New York, N.Y., 1990.
11. Walter Fontana and Leo Buss. The arrival of the fittest: Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 56, 1994.
12. Chris Hankin, Daniel Le Métayer, and David Sands. A calculus of Gamma programs. In *Languages and Compilers for Parallel Computing, 5th International Workshop*, number 757 in Lecture Notes in Computer Science, pages 342–355. Springer-Verlag, August 1992.
13. Daniel Le Métayer. Higher-order multiset programming. In American Mathematical Society (AMS), editor, *Proc. of the DIMACS workshop on specifications of parallel algorithms*, volume 18 of *Dimacs Series in Discrete Mathematics*, 1994.
14. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

Strategy Construction in the Higher-Order Framework of TL

Victor L. Winter*

Department of Computer Science, University of Nebraska at Omaha, USA
vwinter@mail.unomaha.edu

Abstract When viewed from a strategic perspective, a labeled rule base in a rewriting system can be seen as a restricted form of strategic expression (e.g., a collection of rules strictly composed using the left-biased choice combinator). This paper describes higher-order mechanisms capable of dynamically constructing strategic expressions that are similar to rule bases. One notable difference between these strategic expressions and rule bases is that strategic expressions can be constructed using arbitrary binary combinators (e.g., left-biased choice, right-biased choice, sequential composition, or user defined). Furthermore, the data used in these strategic expressions can be obtained through term traversals.

A higher-order strategic programming framework called TL is described. In TL it is possible to dynamically construct strategic expression of the kind mentioned in the previous paragraph. A demonstration follows showing how the higher-order constructs available in TL can be used to solve several problems common to the area of program transformation.

1 Introduction

The concept of distributing data within a term structure is central to rewrite-based computation [12]. In [17] this problem is characterized and referred to as the *distributed data problem* (DDP). When the data to be distributed is independent of the input (i.e., constant for all input terms), simple strategies for distributing data can oftentimes be constructed statically. For example, consider constructing a strategy that will rewrite every integer in a term to the integer 2. Here the objective is to distribute the integer 2 throughout a term structure by rewriting every integer encountered. This is an example of data distribution involving data that is independent of any specific input term.

In contrast, consider constructing a strategy that will rewrite every integer in a term so that all integers are equal to the first integer in the term. For example, for a given term t if the first integer in t is 27 then all integers in t should be rewritten to 27. This is an example of data distribution involving data that is dependent on the input term. In the area of program transformation, the distribution of dependent data throughout a term is typically more common than the independent distribution of data. For example, variable renaming, function in-lining, and constant propagation all require the distribution of dependent data through a term structure.

Strategic/rewriting systems are often provided with extensions in order to enhance their ability to describe the distribution of data. Parameterization is one extension that is widely used as a mechanism for data distribution. For example, ASF+SDF [1] has been extended with a fixed collection of parameterizable traversal functions [5]. Another extension is to allow rule instances to be dynamically constructed using problem dependent data. In Stratego [14] for example, a primitive is provided making it possible to alter rule bases at runtime through the dynamic construction and destruction of rules. Such types of manipulations

* This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy. Victor Winter was also partially supported by NSF grant number CCR-0209187.

can also be realized in Maude [8][7] using its reflective capabilities, and also to some extent in Elan [3].

In this paper we look at higher-order extensions to strategic programming. Specifically we will describe how the higher-order rules, strategies, and traversals of a strategic programming language called TL can be used to effectively distribute (dependent) data throughout term structures. Interesting aspects of this approach are:

1. Within the structure of a source program, data aggregations (e.g., declaration lists, symbol tables, etc.) can be collected via higher-order traversals. During the course of such a traversal, the data elements forming an aggregation can be independently placed within individual strategies and rules (e.g., one data element per strategy). These strategies and rules capture the control necessary to then effectively use the data elements during a second traversal to achieve a desired program transformation (e.g., symbolic resolution in Java class files, function in-lining, and so on).
2. Structurally disjoint data can be collected incrementally via rules and strategies that lie beyond the second order. For example, two disjoint data structures can be manipulated via a third-order strategy.

Though TL is presently a theoretical framework, a restricted dialect of TL has been implemented in the HATS[1] system [20] and is freely available. All of the examples presented in this paper have been implemented in HATS.

The remainder of the paper is organized as follows. Section 2 gives an overview of TL. Section 3 takes an in-depth look at the inner workings of a strategic implementation of set union in TL. Section 4 looks at two manipulations common in the area of program transformation. Section 5 discusses some related work, and Section 6 concludes.

2 An Overview of TL

TL [17] is an *identity-based* higher-order strategic system for rewriting parse trees. We use the term identity-based to denote rewriting systems in which the failure of rule application to a term leaves the term unchanged. We use the term *failure-based* to denote systems where the unsuccessful application of a rule to a term yields a special failure value. In contrast to TL, the strategic programming systems Stratego [12] and Elan [2] are *failure-based*.

In TL, a domain (i.e., a term language) is defined using an Extended-BNF and terms also called *parse expressions* are described using a special notation. TL supports the constructs, combinators and strategic constants shown in Figure 1.

In addition to the constructs shown in Figure 1, TL also provides a number of one-layer generic traversals providing the ability to construct user-defined traversals. These constructs are not central to the topic of this paper and are therefore omitted. Instead we simply present a number of generic traversals that form part of the TL traversal library.

2.1 Term Notation

In TL, term structures are defined using a concrete syntax (e.g., an extended-BNF). This is somewhat a departure from most rewriting systems where term structures are traditionally defined using an abstract syntax. There are several advantages to using a concrete syntax

[1] Other than differences in syntax, the primary restriction is that the construction of user-defined strategies is not supported in HATS.

skip	A strategy constant that never applies
$lhs \to rhs$ if *condition*	A conditional first-order strategy (see Section 2.2 for more on *conditions*)
$lhs \to s^n$ if *condition*	A conditional strategy of order $n+1$
$s_1^n ; s_2^n$	Sequential composition
s_1^n <+ s_2^n	Left-biased choice
s_1^n +> s_2	Right-biased choice
$I(s^n)$	A unary combinator that does nothing
$fix(s^1)$	The fixed point application of the first-order strategy s^1
$transient(s^n)$	A unary combinator restricting the application of s^n

Figure 1. The basic constructs of TL

for defining *term* structures (which we will also refer to as *trees* when we need to distinguish them from the more traditional *terms*). In a practical setting, when using rewriting to manipulate programs, the conceptual gap between the concrete syntax of an industrial programming language and its abstract representation is oftentimes significant [15]. The width of this gap can reach the the point where it negatively impacts the ability to reason about the program manipulations one wants to achieve. Under such circumstances expressing rewriting rules and strategies directly in terms of the concrete syntax of the language is more appropriate.

Another advantage a concrete syntax offers is that the internal structure of a tree can be automatically completed using a parser. In contrast, due to the ambiguities inherent in abstract syntax definitions, the internal structure of a term, in its purest sense, cannot be automatically completed. In addition to easing the burden of term construction on the user, automatic completion of trees also assures that trees will always be well-formed entities as defined by a given concrete grammar. In contrast, abstract syntax-based frameworks typically require the internal structure of terms to be made explicit (by the user) within strategy and rule definitions.

We feel that the conceptual gap issue and the term/tree completion capability are significant enough distinctions to justify our departure from the traditional term nomenclature and representation. Tree representations have other advantages over terms, and the reverse is also true, but such discussions lie beyond the scope of this paper.

In TL, terms are defined in the following compact form which a parser can automatically complete. Let $G = (N, T, P, S)$ denote a context-free grammar where N is the set of nonterminals, T is the set of terminals, P is the set of productions, and S is the start symbol. Given an arbitrary symbol $B \in N$ and a string of symbols $\alpha = X_1 X_2 ... X_m$ where for all $1 \leq i \leq m : X_i \in N \cup T$, we say B derives α iff the productions in P can be used to expand B to α. Traditionally, the expression $B \stackrel{*}{\Rightarrow} \alpha$ is used to denote that B can derive α in zero or more expansion steps. Similarly, one can write $B \stackrel{+}{\Rightarrow} \alpha$ to denote a derivation consisting of one or more expansion steps.

In TL, we write $B[[\alpha']]$ to denote an *instance* of the derivation $B \stackrel{+}{\Rightarrow} \alpha$ whose resulting value is a parse tree having B as its root symbol. In TL, expressions of the form $B[[\alpha']]$ are referred to as *parse expressions*. In the parse expression $B[[\alpha']]$ the string α' is an *instance* of α because nonterminal symbols in α' are constrained through the use of subscripts. Subscripted nonterminal symbols are referred to as *schema variables* or simply *variables* for short. TL also considers a schema variable (e.g., B_i) to be a parse expression in its own right.

Figure 2 shows a BNF grammar fragment describing a small portion of an imperative language. The parse expressions $stmt[\![\ id_1\ =\ 5\ ]\!]$ and $stmt[\![\ id_2\ =\ 5\ ]\!]$ both describe instances of the derivation $stmt \stackrel{+}{\Rightarrow} id\ =\ 5$.

$$\begin{array}{lcl} stmt & ::= & assign \mid cond \mid ... \\ assign & ::= & lvalue \text{ "="}\ expr \\ ... & & \\ lvalue & ::= & id \\ expr & ::= & int \\ ... & & \end{array}$$

Figure 2. A concrete syntax fragment

Within a given scope all occurrences of schema variables having the same subscript denote the same variable. The purpose of placing subscripts on schema variables is to enable grammar derivations to be restricted with respect to one or more equality-oriented constraints. The difference between a nonterminal B and a schema variable B_i is that B is traditionally viewed as a set (or syntactic category) while B_i is a typed variable quantified over the syntactic category B.

When the dominating symbol and specific structure of a parse expression is unimportant the parse expression will be denoted by variables of the form $t, t_1, ...$ or variables of the form $tree, tree_1, tree_2$, and so on. Parse expressions containing no schema variables are called *ground* and parse expressions containing one or more schema variables are called *non-ground*. And finally, within the context of rewriting or strategic programming, *trees* as described here can and generally are viewed as *terms*. When the distinction is unimportant, we will refer to *trees* and *terms* interchangeably.

2.2 Rules

TL supports conditional labeled first-order rewrite rules of the form:

$$\text{label} : lhs \rightarrow rhs \text{ if } condition$$

where lhs and rhs are terms and the rule label and conditional part are optional components. Higher-order rules have the form:

$$\text{label} : lhs \rightarrow s^n \text{ if } condition$$

where s^n is an unlabeled (possibly higher-order) rule. When parsing higher-order rules, the $\rightarrow$ associates to the right. An abstract example of a second-order condition-free rule is:

$$r{:}lhs_1 \rightarrow lhs_2 \rightarrow rhs_2$$

In order to disambiguate the internal structure (e.g., conditional components) of higher-order rules one may enclose the righthand side of a rule in parenthesis.

$$\text{label} : lhs \rightarrow (s^n) \text{ if } condition$$

As a notational convenience, labeled higher-order rules without conditions may be written in curried form when appropriate. For example, a rule of the form:

$$r{:}x_1 \to x_2 \to x_3 \to x_4$$

can be equivalently written as:

$$r\ x_1{:}x_2 \to x_3 \to x_4$$

or can even be curried further and written as:

$$r\ x_1\ x_2{:}x_3 \to x_4$$

Rule Conditions The conditional portion of a rule is a *match-expression* consisting of one or more *match-equations*. The symbol $\ll$, adapted from the ρ-calculus [6], is used to denote first-order matching modulo an empty equational theory. Let t_2 denote a ground tree and let t_1 denote a parse expression which may contain one or more schema variables. A *match-equation* is denoted by $t_1 \ll t_2$. Equivalently we may also write $t_2 \gg t_1$. The symbols ll and $\gg$ are boolean valued operations that produce a substitution σ as a by-product (i.e., as an internal value that is not explicitly accessible to the user). A substitution σ binding schema variables to ground parse expressions is a solution to $t_1 \ll t_2$ if $\sigma(t_1) = t_2$ with $=$ denoting a boolean valued test for syntactic equality.

A *match-expression* is a boolean expression involving one or more match-equations. Match-expressions may be constructed using the standard boolean operators: $\wedge, \vee, \neg$. A substitution σ is a solution to a match-expression m iff $\sigma(m)$ evaluates to true using the standard semantics for boolean operators.

Rule Application The application of a conditional rewrite rule r to a tree t is expressed as $r(t)$ where r is either a label or an anonymous rule value e.g., $lhs \to s^n$. We adopt a curried notation in the style of ML where application is a left-associative implicit operator and parentheses are used to override precedence or may be optionally included to enhance readability. For example, $r\ t$ denotes the application of r to t and has the same meaning as $r(t)$.

2.3 Some First-Order Traversals from the TL Library

TL provides support for user-defined first-order traversals. TL also provides a number of standard generic first-order traversals. There are three degrees of freedom for a generic traversal: (1) whether a term is traversed bottom-up or top-down, (2) whether the children of a term are traversed from left-to-right or right-to-left, and (3) whether a standard *threaded* semantics or a *broadcast* semantics is used to propagate strategies within the traversal (see Section 2.6).

Figure 3 gives a list of the most commonly used generic traversals. The first traversal is TDL. This traversal will traverse the term it is applied to in a top-down left-to-right fashion. With the exception of TD_BR which is discussed in Section 2.6, the remaining entries in the table have similar descriptions. The last two traversals perform a fixed point computation with respect to a given traversal scheme.

Traversal	bottom-up	top-down	left-to-right	right-to-left	threaded	broadcast
TDL		√	√		√	
TDR		√		√	√	
TD_BR		√				√
BUL	√		√		√	
BUR	√			√	√	
FIX_TDL		√	√		√	
FIX_TDR		√		√	√	

Figure 3. General first-order traversals

2.4 Higher-Order Strategies

TL is a restricted higher-order strategic programming framework. TL is restricted because it only permits the application of higher-order strategies to ground terms. For example, strategies may not be applied to other strategies or rules as is allowed in the ρ-calculus [6]. In TL, the result of applying an order n strategy to a (ground) term is a strategy of order $n-1$.

From an operational perspective, a higher-order traversal traverses a term and applies a higher-order strategy s^n to every term encountered. Because the strategy being applied is of order n, the result of an application will be a strategy of order $n-1$. If a traversal visits m terms, then m strategies of order $n-1$ will be produced. Let s_1^{n-1}, s_2^{n-1} , ... , s_m^{n-1} denote the strategies resulting from such a traversal. In TL, a variety of binary strategic combinators can be used to combine the strategic results s_1^{n-1}, s_2^{n-1}, ... , s_m^{n-1} into a strategic expression (i.e., a strategy). Let $\oplus$ denote a binary combinator such as sequential composition, left-biased choice, right-biased choice, or a user-defined binary combinator. Higher-order traversals will combine these strategies into a strategic expression of the form:

$$s_1^{n-1} \oplus s_2^{n-1} \oplus ... \oplus \; s_m^{n-1}$$

There is one technical detail that has been omitted from the above explanation. In addition to combining strategies using a binary combinator, a higher-order traversal also uniformly applies a unary combinator τ to every resultant strategy. Thus, the actual strategy produced is:

$$\tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus ... \oplus \; \tau(s_m^{n-1})$$

In practice, the unary combinator that is most useful is the *transient* combinator with the I combinator playing the role of a default. The *transient* combinator is described in Section 2.5.

TL provides support for user-defined higher-order traversals. TL also provides a number of standard generic higher-order traversals. There are four degrees of freedom for a generic higher-order traversal: (1) whether a term is traversed bottom-up or top-down, (2) whether the children of a term are traversed from left-to-right or right-to-left, (3) which binary combinator should be used to compose the result strategies, and (4) which unary combinator should be used to wrap each resulting strategy.

Figure 4 gives a list of the most commonly used generic traversals. The first traversal in this list is *rcond_tdl*. This traversal will traverse the term it is applied to in a top-down left-to-right fashion. The result strategies will then be composed using the right-biased choice combinator and finally each result strategy will be wrapped in the unary combinator I. The remaining entries in the table have similar descriptions.

Travaersal	**bottom-up**	**top-down**	**left-to-right**	**right-to-left**	$\oplus$	τ
rcond_tdl		$\surd$	$\surd$		+>	I
rcond_tdr		$\surd$		$\surd$	+>	I
lcond_tdl		$\surd$	$\surd$		<+	I
lcond_tdr		$\surd$		$\surd$	<+	I
rcond_bul	$\surd$		$\surd$		+>	I
rcond_bur	$\surd$			$\surd$	+>	I
lcond_bul	$\surd$		$\surd$		<+	I
lcond_bur	$\surd$			$\surd$	<+	I
seq_tdl		$\surd$	$\surd$		;	I
seq_tdr		$\surd$		$\surd$	;	I
seq_bul	$\surd$		$\surd$		;	I
seq_bur	$\surd$			$\surd$	;	I

Figure 4. General higher-order traversals

2.5 The *transient* Combinator

The transient combinator is a very special combinator in TL. This combinator restricts a strategy so that it may be applied *at most once*. The "at most once" property characterizes the *transient* combinator and motivates the introduction of *skip* into the framework of TL. We define *skip* as a strategy whose application never succeeds.

Figure 5 gives some relationships between two abstract strategic constants ϵ and δ and the combinators <+ and ;. These relationships are considered from the perspective of a failure-based framework as well as an identity-based framework. In failure-based systems such as Stratego and ELAN, ϵ is typically called *id* or *identity* and δ is typically called *fail*. In the identity-based framework of TL, ϵ is called *id* and δ is called *skip*.

Strategy	Failure-Based Semantics	Identity-based Semantics
$\epsilon <+ s$	ϵ	ϵ
$s <+ \epsilon$	$s <+ \epsilon$	s
$\delta <+ s$	s	s
$s <+ \delta$	s	s
$\epsilon \; ; \; s$	s	s
$s \; ; \; \epsilon$	s	s
$\delta \; ; \; s$	δ	s
$s \; ; \; \delta$	δ	s

Figure 5. The semantics of *id*, *skip*, and *fail*

TL defines a strategy of the form $transient(s)$ as a strategy that *reduces* to the strategy *skip* if the application of the strategy s has been observed. Furthermore, only the innermost (i.e., closest enclosing) transient can observe the application of a strategy. This restriction is needed to prevent a cascading sequence of reductions for strategies containing nested transients.

Transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, a different strategy may be applied to every term encountered during a traversal. For example, let $int_1 \rightarrow int[[2]]$ denote a rule that rewrites an arbitrary integer to the value 2. If such a rule is applied to a term in a top-down fashion all of the integers in the term will be rewritten to 2. Now consider the following self-modifying transient strategy:

$$transient(int_1 \rightarrow int[[1]]) <\!\!+ \; transient(int_1 \rightarrow int[[2]]) <\!\!+ \; transient(int_1 \rightarrow int[[3]])$$

When applied to a term in a top-down fashion, this strategy will rewrite the first integer encountered to 1, the second integer encountered to 2, and the third integer encountered to 3. All other integers will remain unchanged.

2.6 Traversal Mechanisms

TL provides two types of term traversal: a *threaded* traversal and a *broadcasting* traversal. In a *threaded* traversal (e.g., TDL, TDR, BUL, BUR), terms are visited in sequential order and a single strategy is passed from term to term. A diagram showing the behavior of a threaded traversal can be seen in Figure 6.

In a *broadcasting* traversal (e.g., TDL_BR) a distinct copy of the strategy resulting from an application will be given to all of the children of a term. For example, the evaluation of the strategic expression $TDL\_BR(s)t$ will first apply the strategy s to the term t. Recall that in the most general case (i.e., when transients are present in the strategy), the result of such an application will alter both s as well as t. Let s' and t' respectively denote the strategy and term resulting from the application of s to t. Since TDL_BR is a broadcasting traversal, a distinct copy of s' will be applied to each of the sub-terms of t'. A diagram showing the behavior of a broadcasting traversal can be seen in Figure 7.

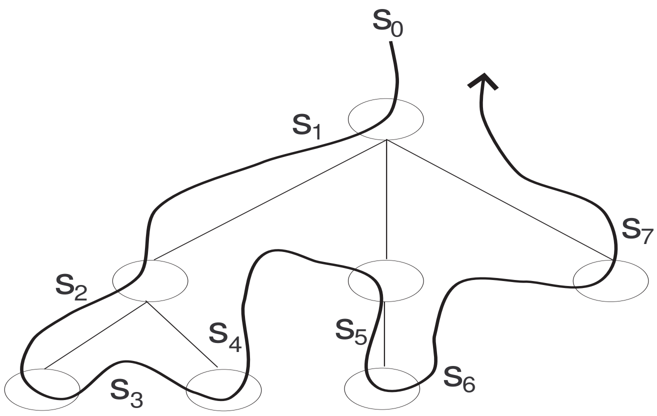

Figure 6. Diagram of the threaded traversal *TDL* from the perspective of strategy application

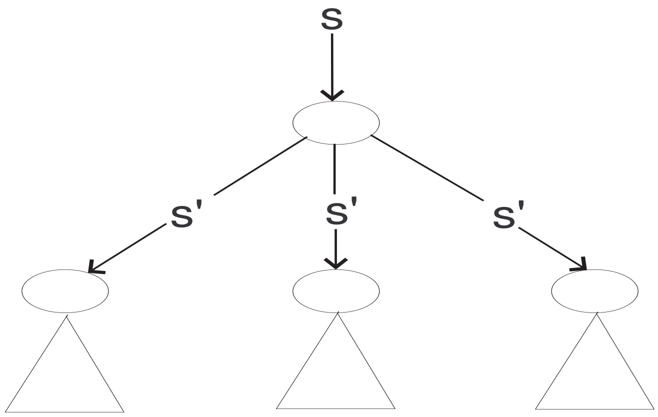

Figure 7. Diagram of the broadcasting traversal *TDL_BR* from the perspective of strategy application

3 A Benchmark: Set Union

We believe that set union has characteristics similar to a number of common transformational activities. For example, variations of set union can be used as the basis for variable renaming, data flow analysis, control flow analysis, symbolic resolution in Java class files [17], as well as field distribution and method table construction [18] in Java class files. Thus, because of its wide range of applicability, we consider set union to be a benchmark problem for a strategic programming system.

In this section we look at how the union benchmark can be solved in TL. Our approach is to lift basic operations on data (e.g., insertion of an element into a set, etc.) to the strategy level. For example, when implementing union, we wish to create a strategy that inserts a particular element into our union set only if the element does not already occur in the set. In TL the construction of these types of problem specific first-order strategies can be accomplished though higher-order strategies.

In Figure 8 a BNF grammar is given describing a language of set/sequence expressions. The meta-symbols of the grammar are ::=, (), |, <, >, “, and ”. The symbol () is used to denote the epsilon symbol, domain variables are enclosed in pointy brackets and terminal symbols are enclosed in quotes.

In Figure 9, *keep* and *add* are strategies realizing primitive operations on sets such as adding an element to an empty set. The strategy *union_s* is higher-order and defines a single computational step (e.g., a strategy that will “union” one element to a set). And finally, the strategy *make_union* performs its respective set operation by first properly instantiating *union_s* with respect to every element in set_1 and then applying the resulting strategy to the set_2.

<table>
<tr><td>set_expr</td><td>::= set set_op set | set</td></tr>
<tr><td>set</td><td>::= “{” es “}”</td></tr>
<tr><td>es</td><td>::= e es | ()</td></tr>
<tr><td>e</td><td>::= <id> | “(” <id> <id> “)”</td></tr>
<tr><td>set_op</td><td>::= “union”</td></tr>
</table>

Figure 8. A BNF describing set/sequence expressions

$keep\ e_1$ $: es[[e_1\ es_2]] \rightarrow es[[e_1\ es_2]]$
$add\ e_1$ $: es[[\ ]] \rightarrow es[[e_1]]$

$union\_s$ $: es[[\ e_1\ es_1\ ]] \rightarrow transient((keep\ e_1) <\!+ (add\ e_1))$

$make\_union : set\_expr[[set_1\ union\ set_2]] \rightarrow TDL(lcond\_tdl\ union\_s\ set_1)\ set_2$

Figure 9. Instantiation and application of second-order strategies to terms

3.1 A Closer Look at the Implementation of Union in TL

The strategic theme here is to decompose a set expression $\{a_1, a_2, ..., a_n\} \cup \{e_1, e_2, ..., e_m\}$ into a sequence of incremental strategies each of which can be used to evaluate an expression of the form: $S \cup \{e_i\}$. The higher-order strategy *union_s* generates such incremental strategies. Specifically, when given the context $es[[\ e_1\ es_1\ ]]$, *union_s* will extract the element e_1 and produce a *transient* strategy consisting of the conditional composition $keep(e_1) <\!+ add(e_1)$.

Building on *union_s* is the strategic expression $(lcond\_tdl\ union\_s\ set_1)$ which traverses set_1 producing the conditional composition of instances of *union_s*; one instance for each element in set_1. The resulting strategy is then applied to set_2 using the traversal TDL. Keeping this in mind, let us trace the strategic evaluation of the expression $set_1 \cup set_2$ where $set_1 = \{x_1\ x_2\ x_3\ x_4\}$ and $set_2 = \{y_1\ x_2\ x_3\ y_2\}$.

The result of $(lcond\_tdl\ union\_s\ set_1)$ and its application to the first term in set_2 are shown in Figures 10 through 14. Figure 10 shows the initial strategy applied to set_2. Figures 11 and 12 show how the strategy changes as it encounters (is applied to) the elements x_2 and x_3 respectively. The application of the strategy to the element y_2 has no effect and is shown in Figure 13. And finally, in Figure 14 the traversal reaches the end of set_2 at which time the element x_1 is added. Note that in this case, both the strategy and set_2 are changed by the application. In a similar fashion, x_4 is added yielding $\{y_1\ x_2\ x_3\ y_2\ x_1\ x_4\}$ as the final term and *skip* as the final strategy.

4 Adaptations to Common Transformational Objectives

In this section we look at TL solutions to two common transformational objectives that arise in the area of program transformation. We would like to mention that these examples were inspired from similar examples presented in [14].

Both examples are considered with respect to the grammar fragment defined in Figure 15. The meta-symbols of the grammar are ::=, (), |, <, >, “, ”, [, and]. The symbol () is used to denote the epsilon symbol, domain variables are enclosed in pointy brackets, terminal symbols are enclosed in quotes, and optional portions of productions are enclosed in square brackets.

4.1 Variable Renaming

In this example, we consider the variable renaming problem for a small block structured imperative language. (Note that the grammar given Figure 15 permits blocks to be nested).

$$\begin{array}{ll}
 & \{\overset{\Downarrow}{y_1}\ x_2\ x_3\ y_2\} \\
\quad \text{transient}(es[[x1\ es_2]] \to es[[x1\ es_2]] <\!\!+ es[[\ ]] \to es[[x1]]) & \\
<\!\!+ \text{transient}(es[[x2\ es_2]] \to es[[x2\ es_2]] <\!\!+ es[[\ ]] \to es[[x2]]) & \\
<\!\!+ \text{transient}(es[[x3\ es_2]] \to es[[x3\ es_2]] <\!\!+ es[[\ ]] \to es[[x3]]) & \\
<\!\!+ \text{transient}(es[[x4\ es_2]] \to es[[x4\ es_2]] <\!\!+ es[[\ ]] \to es[[x4]]) & \{\overset{\Downarrow}{y_1}\ x_2\ x_3\ y_2\}
\end{array}$$

Figure 10. Union with TDL traversal – The term y_1 in set_2 is unaffected

$$\begin{array}{ll}
\quad \text{transient}(es[[x1\ es_2]] \to es[[x1\ es_2]] <\!\!+ es[[\ ]] \to es[[x1]]) & \{y_1\ \overset{\Downarrow}{x_2}\ x_3\ y_2\} \\
\cancel{<\!\!+ \text{transient}(es[[x2\ es_2]] \to es[[x2\ es_2]] <\!\!+ es[[\ ]] \to es[[x2]])} & \{y_1\ \overset{\Downarrow}{x_2}\ x_3\ y_2\} \\
<\!\!+ \text{transient}(es[[x3\ es_2]] \to es[[x3\ es_2]] <\!\!+ es[[\ ]] \to es[[x3]]) & \\
<\!\!+ \text{transient}(es[[x4\ es_2]] \to es[[x4\ es_2]] <\!\!+ es[[\ ]] \to es[[x4]]) &
\end{array}$$

Figure 11. Union with TDL traversal – The term x_2 changes the strategy

$$\begin{array}{ll}
\quad \text{transient}(es[[x1\ es_2]] \to es[[x1\ es_2]] <\!\!+ es[[\ ]] \to es[[x1]]) & \{y_1\ x_2\ \overset{\Downarrow}{x_3}\ y_2\} \\
\cancel{<\!\!+ \text{transient}(es[[x2\ es_2]] \to es[[x2\ es_2]] <\!\!+ es[[\ ]] \to es[[x2]])} & \\
\cancel{<\!\!+ \text{transient}(es[[x3\ es_2]] \to es[[x3\ es_2]] <\!\!+ es[[\ ]] \to es[[x3]])} & \{y_1\ x_2\ \overset{\Downarrow}{x_3}\ y_2\} \\
<\!\!+ \text{transient}(es[[x4\ es_2]] \to es[[x4\ es_2]] <\!\!+ es[[\ ]] \to es[[x4]]) &
\end{array}$$

Figure 12. Union with TDL traversal – The term x_3 changes the strategy

$$\begin{array}{ll}
\quad \text{transient}(es[[x1\ es_2]] \to es[[x1\ es_2]] <\!\!+ es[[\ ]] \to es[[x1]]) & \{y_1\ x_2\ x_3\ \overset{\Downarrow}{y_2}\ \} \\
\cancel{<\!\!+ \text{transient}(es[[x2\ es_2]] \to es[[x2\ es_2]] <\!\!+ es[[\ ]] \to es[[x2]])} & \\
\cancel{<\!\!+ \text{transient}(es[[x3\ es_2]] \to es[[x3\ es_2]] <\!\!+ es[[\ ]] \to es[[x3]])} & \\
<\!\!+ \text{transient}(es[[x4\ es_2]] \to es[[x4\ es_2]] <\!\!+ es[[\ ]] \to es[[x4]]) & \{y_1\ x_2\ x_3\ \overset{\Downarrow}{y_2}\ \}
\end{array}$$

Figure 13. Union with TDL traversal – Processing the term y_2 has no effect

$$\begin{array}{ll}
 & \{y_1\ x_2\ x_3\ y_2\ \Downarrow\ \} \\
\cancel{\quad \text{transient}(es[[x1\ es_2]] \to es[[x1\ es_2]] <\!\!+ es[[\ ]] \to es[[x1]])} & \{y_1\ x_2\ x_3\ y_2\ x_1\ \} \\
\cancel{<\!\!+ \text{transient}(es[[x2\ es_2]] \to es[[x2\ es_2]] <\!\!+ es[[\ ]] \to es[[x2]])} & \\
\cancel{<\!\!+ \text{transient}(es[[x3\ es_2]] \to es[[x3\ es_2]] <\!\!+ es[[\ ]] \to es[[x3]])} & \\
<\!\!+ \text{transient}(es[[x4\ es_2]] \to es[[x4\ es_2]] <\!\!+ es[[\ ]] \to es[[x4]]) &
\end{array}$$

Figure 14. Union with TDL traversal – The term x_1 is added to the union

The TL solution makes use of a function *new*[2] that has the ability to generate unique variable names.

The code in Figure 16 highlights some of the issues that must be addressed when renaming variables in this setting. First, variables may be redeclared within a nested block. However, it is assumed that variables may not be redundantly declared within a given declaration list. Second, variable declarations may include an assignment to an initial expression which may contain occurrences of previously declared variables.

When dealing with declarations having initialization expressions, one must be careful to associate variables with their proper declarations. For example, in Figure 16 in the declaration $int\ x1 = x1 + 1$ in the inner block, the reference to the variable $x1$ occurring in the initialization expression $x1 + 1$ is actually a reference to the previous declaration of $x1$

[2] Because of its widespread use in program transformation, the function *new* is provided as a primitive in many transformation systems.

```
prog         ::=  block
block        ::=  "{" dec_list stmt_list "}"
dec_list     ::=  dec ";" dec_list | ()
dec          ::=  type id
                | type id "=" expr
                | "fun" id "(" id_list ")" "=" expr
type         ::=  "int" | "bool" | ...
stmt_list    ::=  stmt ";" stmt_list | ()
stmt         ::=  assign | block | ...
assign       ::=  id "=" expr
expr         ::=  cond | logical_expr
cond         ::=  "if" expr "then" expr "else" expr
logical_expr ::=  rel | ...
rel          ::=  expr "=" expr | E | ...
E            ::=  E "+" T | E "-" T | T
T            ::=  T "*" F | F "/" F | F
F            ::=  id | num | "(" expr ")" | id "(" expr_list ")" | ...
id_list      ::=  id [ "," id_list ] | ()
expr_list    ::=  actual [ "," expr_list | ()
actual       ::=  expr
id           ::=  <ident>
num          ::=  <integer>
...
```

Figure 15. A grammar fragment of a small block structured imperative language

in the outer block. Thus it would be incorrect to rename $int\ x1 = x1 + 1$ to $int\ new\_x1 = new\_x1 + 1$. Instead, the renaming should result in something like $int\ new\_x1 = x1 + 1$.

Another difficulty in this example results from the structure of a block as defined by the grammar. Specifically, a block has intentionally been defined to consist of a declaration list followed by a statement list. Note that renaming must occur both within the declaration list as well as the statement list.

```
{                              {
   int x1;                        int y4;
   int x2;                        int y5;
   int x3 = x1 + x2;              int y6 = y4 + y5;
   x1 = 5;                        y4 = 5;
   x2 = x1 + 5;                   y5 = y4 + 5;
   {                              {
     int x1 = x1 + 1;      ⟹       int y1 = y4 + 1;
     int z1 = x1 + x2;              int y2 = y1 + y5;
     int z2 = 4;                    int y3 = 4;
     x1 = x2 + z1 * z2;             y1 = y5 + y2 * y3;
   };                             };
   x1 = x2 + x1;                  y4 = y5 + y4;
}                              }
```

Figure 16. Considerations when renaming variables: A block before and after variable renaming

Figure 17 gives a TL implementation of variable renaming. An overview of our strategic approach to the variable renaming problem is as follows. Blocks are processed in an inside-out manner (i.e., nested blocks first). When a block is encountered, its declaration list

$restricted\ id_1\ id_2$	$:\ dec[[\ type_1\ id_1 = expr_1\ ]] \rightarrow dec[[\ type_1\ id_2 = expr_1\ ]]$
$unrestricted\ id_1\ id_2$	$:\ id_1 \rightarrow id_2$
gen_rename	$:\ dec_1 \rightarrow transient((restricted\ id_1\ id_2) <\!+ (unrestricted\ id_1\ id_2))$ if $id_2 \ll new \wedge (dec_1 \ll dec[[\ type_1\ id_1\ ]] \vee dec_1 \ll dec[[\ type_1\ id_1 = expr_1]])$
$rename$	$:\ block_1 \rightarrow TD\_BR(cond\_tdl\ gen\_rename\ dec\_list_1)\ block_1$ if $block_1 \ll block[[dec\_list_1\ stmt\_list_1]]$
var_rename	$:\ prog_1 \rightarrow BUL\ rename\ prog_1$

Figure 17. The strategies for renaming variables

will be traversed in a top-down fashion and a strategic expression will be constructed that is capable of renaming all variables within the block (variables occurring in both the declaration list as well as the statement list). Special care is taken to assure that variables occurring in *initializing expressions* (i.e., expressions on the right-hand sides of assignments in declarations) do not have their variables inappropriately renamed.

In the TL implementation shown in Figure 17 the strategies *restricted* and *unrestricted* are third-order strategies in curried form that when given a variable name id_1 and a corresponding fresh variable name id_2 will yield a first-order rule describing a specific kind of renaming. The strategy $restricted\ id_1\ id_2$ describes the rewriting that should occur when the declaration of id_1 is encountered. In particular, the declaration of id_1 should be renamed to id_2, but the initializing expression should remain untouched. The strategy $unrestricted\ id_1\ id_2$ describes the rewriting that should occur in all other cases.

Building on the *restricted* and *unrestricted* rules, is the higher-order strategy *gen_rename*. When applied to a declaration, *gen_rename* will create a transient of the form:

$$transient((restricted\ id_1\ id_2) <\!+ (unrestricted\ id_1\ id_2))$$

Note that this transient strategy can only be applied once and will perform either a restricted or unrestricted rename. During the course of a top-down traversal, the idea is to have this transient apply to the declaration which generated it after which it will reduce to *skip* for all subtrees of that declaration. If this can be accomplished, then any traversal that continues on to the initialization expression will leave all occurrences of the declared variable unchanged. In addition to this behavior, we would like the renaming to continue for the rest of the block (e.g., the remaining declarations and statements). It is precisely this behavior that can be accomplished by TD_BR.

One way of understanding the effect of TD_BR when used in conjunction with a transient is that TD_BR captures the notion of "*not below*" with respect to a tree structure. The notion of "*not below*" was first used in TAMPR [4]. For a given tree t and a given leaf x, let p denote a *path* from the root of t to the leaf x. Let s denote a first-order strategy (containing no transient combinators). The traversal $TD\_BR\ transient(s)\ t$ will apply s at most once on every *path* in t. For example, if s applies at a particular point in a path, then $transient(s)$ will reduce to *skip* after this application and will therefore not apply anywhere else on the path.

Given this understanding of the interaction between TD_BR and the *transient* combinator, let us consider the parse expression $dec\_list[[dec_1; dec\_list_1]]$. When applied to this term, the strategy $TD\_BR\ s$ will first apply s to $dec\_list[[dec_1; dec\_list_1]]$ yielding the strategy s'. A **copy** of the strategy s' is then broadcast to each of the children of $dec\_list[[dec_1; dec\_list_1]]$. In particular, both dec_1 and dec_list_1 will receive their own copy of s'. More specifically, let us consider what happens when s is $transient((restricted\, id_1\ id_2)$ $<+\ (unrestricted\, id_1\ id_2))$. In this case, the application of s to $dec\_list[[dec_1; dec\_list_1]]$ will leave s unchanged (i.e., $s = s'$). Next a copy of s' will be broadcast to both dec_1 and dec_list_1. If dec_1 is the declaration responsible for generating s', then s' will apply to dec_1 but will not apply to any subterm below dec_1 (e.g., the initializing expression in dec_1). In contrast, within dec_list_1 s' will continue attempting to apply and broadcast its own copy of s' to its children. This will enable the strategy $(unrestricted\, id_1\ id_2)$ within the transient s' to rename all remaining occurrences of id_1 to id_2 within the block which is what is desired.

And finally, in the strategy *var_rename* the traversal BUL causes all the blocks in a program to be renamed in an inside-out fashion.

4.2 Naïve Function In-lining

When performing function in-lining the goal is to replace a function call with an instance of its body. This body instance is obtained by substituting the formal parameters associated with the function definition by the actual parameters associated with the call. An example of in-lining is shown in Figure 18. In Figure 19, a TL implementation for performing naïve function in-lining is given. The strategy *fun_inline* is naïve because it does not consider problems that may arise as a result of recursive function calls or address issues resulting in the duplication of expressions corresponding to actual arguments.

The strategy *fun_inline* uses matching to split a block into its declaration list and statement list. The declaration list is then processed by the strategy *fun_dec* which creates an in-lining strategy for each function declaration and composes the results into a strategic expression. This strategic expression is then applied to the original declaration list in order to in-line all the function calls within the declaration list. Then this in-lined declaration list is again processed by the strategy *fun_dec*. This time the resulting strategy is applied to the statement list which has the effect of in-lining all function calls. The resulting statement list is then cleaned up (e.g., excess parenthesis are removed from expressions) by the strategy *remove_parens* whose implementation is not shown. Finally, the resulting statement list is substituted for the statement list in the original block.

The strategy *fun_dec* accomplishes its transformational objective through the help of the strategy *inline*. This strategy is given the name of a function id_1, its formal parameter list id_list_1, and its body $expr_1$ in curried form. With this information, the strategy *inline* is capable of rewriting a function call $F[[\ id_1(expr\_list_1)\ ]]$ to an appropriately in-lined body $F[[\ (expr_2)\ ]]$. It accomplishes this with the help of the strategy *zip*.

As a definition the strategy *zip* is simply a macro and serves no other purpose than to enhance the readability of the conditional portion of the *inline* strategy. Operationally, the body of *zip* will first perform a traversal on id_list the formal parameter list of a function. This traversal will create one *transient* strategy for each formal parameter id in id_list. Let s denote the resulting strategic expression. Next, a traversal on the actual parameter list $expr\_list$ is performed with the strategy s. This will result in a strategic expression consisting of a collection of rules of the form $F[[\ id_1\ ]] \to F[[\ (expr_1)\ ]]$, where id_1 is a formal parameter and $expr_1$ is a corresponding actual parameter. The transient combinator

```
{
  int z1;

  fun f1( x, y ) = x + y ;
  fun f2( x, y, z ) = x * y + z;
  fun f3(x) = f1(x,x);

  z1 = f1( f1(200,300), f1(400,500) );
  z1 = f1( 2, 3 ) + f2(22, 33, 44);
  z1 = f3(f3(3)) + f3(4);
}
```

$\Longrightarrow$

```
{
  int z1;

  fun f1( x, y ) = x + y ;
  fun f2( x, y, z ) = x * y + z;
  fun f3(x) = f1(x,x);

  z1 = 200 + 300 + 400 + 500;
  z1 = 2 + 3 + 22 * 33 + 44;
  z1 = 3 + 3 + 3 + 3 + 4 + 4;
}
```

Figure 18. An example of function in-lining

$formal\_to\_actual$	$: \ id_1 \rightarrow transient(actual[[expr_1]] \rightarrow F[[\ id_1\ ]] \rightarrow F[[\ (expr_1)\ ]])$
$zip\ id\_list_1\ expr\_list_1$	$: \ cond\_tdl(cond\_tdl\ formal\_to\_actual\ id\_list_1)\ expr\_list_1$
$inline\ id_1\ id\_list_1\ expr_1$	$: \ F[[\ id_1(expr\_list_1)\ ]] \rightarrow F[[\ (expr_2)\ ]]$ if $expr_2 \ll TDL(zip\ id\_list_1\ expr\_list_1)\ expr_1$
fun_dec	$: \ dec\_list[[fun\ id_1(id\_list_1) = expr_1; dec\_list_1]]$ $\rightarrow inline\ id_1\ id\_list_1\ expr_1$
$remove\_parens$	: ...
fun_inline	$: \ block[[dec\_list_1\ stmt\_list_1]] \rightarrow block[[dec\_list_1\ stmt\_list_3]]$ if $dec\_list_2 \ll TDL(cond\_tdl\ fun\_dec\ dec\_list_1)\ dec\_list_1 \wedge$ $stmt\_list_2 \ll TDL(cond\_tdl\ fun\_dec\ dec\_list_2)\ stmt\_list_1 \wedge$ $stmt\_list_3 \ll TDL\ remove\_parens\ stmt\_list_2$

Figure 19. A TL implementation of naïve function in-lining

mentioned previously is needed to assure that the proper correspondences between formals and actuals are created. When viewed collectively, the resulting rules are capable of rewriting formal parameters to actual parameters within the body of a function yielding an in-lined instance of that function.

5 Related Work

The higher-order nature of TL rules can be understood as a form of currried rewrite rule. In this context, curried arguments can be bound during the course of a higher-order generic traversal. The composition of strategies created during such generic traversal is related to a morphism. Specifically, the one-layer generic traversal combinators that are used to construct full traversals are similar but not identical to hylomorphisms over rose tree found in functional programming frameworks [10][11]. Similar observations have been made by others. For example, the catamorphism $fold\ b\ \oplus$ can be understood in strategic terms as performing a bottom-up term traversal on the structure of a list where the binary function $\oplus$ of the fold could be used to realize either a type-preserving rewriting function or a type-unifying accumulating function. This connection between catamorphisms and strategic driven term traversal is made in [9].

The ρ-calculus [6] is a failure-based rewriting framework in which matching modulo an equational theory provides the mechanism for the syntactic comparison of terms. In the ρ-calculus the distinction between a rule and a term to which a rule is applied is blurred. Both rules and terms are considered ρ-terms. This uniform treatment is reminiscent of the relationship between functions and terms in the λ-calculus. And, similar to the λ-calculus, in the ρ-calculus there are no restrictions regarding variable occurrences within a term. In particular, free variables may be introduced on the right-hand side of a rule. In fact, the right-hand side of a rule may itself be a rule, seamlessly opening the door to higher-order strategies.

In contrast to the ρ-calculus, TL is a restricted higher-order language. In TL, the name capture problem is sidestepped by the restriction that higher-order strategies only be applied to ground terms (and not to other strategies). Recall that ground terms do not contain (free) schema variables. As a result of this restriction, alpha-conversion, as it is defined in the λ-calculus is not required. In TL, all schema variables within a higher-order strategy fall within a single scope and must be (statically) distinguished accordingly within the definition.

The notion of creating problem specific instances of rules is a core capability of Stratego [14]. These dynamic rewrite rules are named rules that can be instantiated at runtime (i.e., dynamically) yielding a rule instance which is then added to the existing rule base. Dynamic rewrite rules are placed in the "where" portion of another rule and thus have access to information from their surrounding context. Similar to our approach, the input term itself is the driver behind the instantiation of rule variables. The lifetime of dynamic rules can be explicitly constrained in strategy definitions by the scoping operator { | ... |}.

Primary differences between the higher-order strategies in TL and the scoped dynamic rules described in [14] are the following:

1. TL higher-order strategies can be used as the basis of constructing *strategic expressions* that are created dynamically. The $\oplus$ and τ combinators provide the user explicit control over the combinators used to construct this strategy. Stratego views the dynamic instantiation of rules as a rule base (i.e., a strategy where rules are composed using the left-biased combinator and newly created rules are placed on the left-most end of the rule base). It would be interesting to extend the dynamic rule generation mechanism of Stratego to enable more control over the structure of dynamically generated rule bases. This idea has been recently proposed by Martin Bravenboer [16].
2. In Stratego, rule instances can be incrementally added and removed from a rule base. In TL, strategic expressions are created during the course of a separate pass(es) over a term structure. We believe that a separate pass is conceptually cleaner from the perspective of reasoning about the correctness of such structures. However, Stratego's incremental approach is more efficient and also allows a refined control over the contents of such rule bases. On the other hand, the *transient* combinator of TL also allows some degree of control over the contents of strategic expressions.
3. The incremental nature of Stratego's rule bases is similar to the operational or denotational environment models used to describe the semantics of scope. This facilitates thinking about the construction of rule bases in an incremental fashion. In TL, the user is strongly encouraged to think of strategic expressions in a more holistic manner [19].
4. Though the transient combinator has no direct analogy within scoped dynamic rewrite rules, its effects can be simulated in Stratego [16]. However, it is somewhat unclear whether a single approach/method can be used in Stratego to simulate all the behaviors resulting from the interaction between higher-order strategies and transients.

6 Conclusion

TL is based on the premise that higher-order rewriting provides a mechanism for dealing with the distribution of data conforming to the tenets of rewriting. In a higher-order framework, the distribution of data is expressed as a rule. Instantiation of such rules can be done using standard (albeit higher-order) mechanisms controlling rule application (e.g., traversal). Typically, a traversal-driven application of a higher-order rule will result in a number of instantiations. If left unstructured, these instantiations can be collectively seen as constituting a rule base whose creation takes place dynamically. However, such rule bases can encounter difficulties with respect to confluence and termination. In order to address this concern we also lift the notion of strategy construction to the higher-order as well. That is, instantiations are structured to form strategic expressions. Nevertheless, in many cases, simply lifting first-order control mechanisms to the higher-order does not permit the construction of strategic expressions that are sufficiently refined. This difficulty is alleviated though the introduction of the *transient* combinator. The interplay between transients and more traditional control mechanisms enables a variety of instances of the distributed data problem to be elegantly solved in a higher-order setting.

References

1. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.
2. P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. *An Overview of ELAN.* In C. Kirchner and H. Kirchner, eds., International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, France, 1998. Elsevier Science.
3. P. Borovansky, C. Kirchner, H. Kirchner, C. Ringeissen. *Rewriting with Strategies in ELAN: A Functional Semantics.* Int. J. Found. Comput. Sci. 12(1): 69-95 (2001)
4. J. M. Boyle, T. J. Harmer, and V. L. Winter. *The TAMPR program transformation system: Simplifying the development of numerical software.* In Arge, E., Pettersen, A. M., and Langtangen, H. P., editors, Modern Software Tools for Scientific Computing, pages 353–372. Birkhauser, 1997.
5. M.G.J. van den Brand, P. Klint, and J.J. Vinju. *Term Rewriting with Traversal Functions.* ACM Transactions on Software Engineering and Methodology (TOSEM), 12:2, pp 152-190, 2003.
6. H. Cirstea and C. Kirchner. *Intoduction to the rewriting calculus.* INRIA Research Report RR-3818, December 1999.
7. M. Clavel and J. Meseguer. *Reflection and strategies in rewriting logic.* In J. Meseguer, editor, Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier Science Publishers, 1996. Proceedings of the First International Workshop on Rewriting Logic and its Applications.
8. M. Clavel. *Reflection in Rewriting Logic.* CSLI Publications, 2000.
9. R. Lämmel, J. Visser, and J. Kort. *Dealing with Large Bananas.* In Johan Jeuring, editor, Workshop on Generic Programming, Ponte de Lima, July 2000. Technical Report, Universiteit Utrecht.
10. E. Meijer, M.M. Fokkinga, and R. Paterson. *Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire.* In J. Hughes, editor, FPCA'91: Functional Programming Languages and Computer Architecture, volume 523 of LNCS, pages 124-144. Springer-Verlag, 1991.
11. E. Meijer and J. Jeuring. *Merging Monads and Folds for Functional Programming.* In J. Jeuring and E. Meijer, editors, 1st International Spring School on Advanced Functional Programming Techniques, B astad, Sweden, volume 925 of Lecture Notes in Computer Science, pages 228–266. Springer-Verlag, Berlin, 1995.
12. E. Visser, Z. Benaissa, and A. Tolmach. *Building Program Optimizers with Rewriting Strategies.* Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98).
13. E. Visser and Z. Benaissa.*A Core Language for Rewriting.* Eds. C. Kirchner and H. Kirchner Electronic Notes in Theoretical Computer Science Vol. 15, Elsevier Science Publishers, 1998.
14. E. Visser. *Scoped dynamic rewrite rules.* In M. van den Brand and R. Verma, editors, Rule Based Programming (RULE'01), volume 59/4 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, September 2001.

15. E. Visser. *Meta-programming with concrete object syntax.* In D. Batory, C. Consel, and W. Taha, editors, Generative Programming and Component Engineering (GPCE'02), volume 2487 of Lecture Notes in Computer Science, pages 299-315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
16. E. Visser. Personal communication, Feb. 18, 2004.
17. V.L. Winter and M. Subramaniam. *The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem.* Science of Computer Programming (accepted).
18. V.L. Winter, S. Roach, and F. Fraij. *Higher-Order Strategic Programming: A Road to Dependable Software.* Submitted to IEEE Transactions on Dependable and Secure Computing.
19. V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software.* Advances in Computers vol 58.
20. V. Winter. HATS. *http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html*

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

95-11 *	M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases
95-12 *	G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
95-13 *	M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views
95-14 *	P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work
95-15 *	S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems
95-16 *	W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming
96-1 *	Jahresbericht 1995
96-2	M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees
96-3 *	W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
96-4	K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability
96-5	K. Pohl: Requirements Engineering: An Overview
96-6 *	M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
96-7	O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs
96-8 *	S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
96-9	M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming
96-10	R. Conradi / B. Westfechtel: Version Models for Software Configuration Management
96-11 *	C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement
96-12 *	R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
96-13 *	K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools
96-14 *	R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
96-15 *	H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases

96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet

96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation

96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations

96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto

96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design

97-01 Jahresbericht 1996

97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization

97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting

97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen
98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases
98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML
98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
99-01 * Jahresbericht 1998
99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager
99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL
99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge / Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-04 Andreas Becks / Stefan Sklorz / Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 Mareike Schoop: Cooperative Document Management
2000-06 Mareike Schoop / Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 * Jahresbericht 2000
2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03 Thierry Cachat: The power of one-letter rational languages
2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC Languages
2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem
2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09 Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11 Klaus Indermark / Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 * Jahresbericht 2001
2002-02 Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03 Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages
2002-04 Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05 Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09 Markus Mohnen: Interfaces with Default Implementations in Java
2002-10 Martin Leucker: Logics for Mazurkiewicz traces
2002-11 Jürgen Giesl / Hans Zantema: Liveness in Rewriting
2003-01 * Jahresbericht 2002
2003-02 Jürgen Giesl / René Thiemann: Size-Change Termination for Term Rewriting
2003-03 Jürgen Giesl / Deepak Kapur: Deciding Inductive Validity of Equations
2003-04 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Improving Dependency Pairs
2003-05 Christof Löding / Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07 Horst Lichter / Thomas von der Maßen / Alexander Nyßen / Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Mechanizing Dependency Pairs
2004-02 Benedikt Bollig / Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03 Delia Kesner / Femke van Raamsdonk / Joe Wells (eds.): Proceedings of the Second International Workshop on Higher-Order Rewriting (HOR 2004)

2004-04 Slim Abdennadher / Christophe Ringeissen (eds.): Proceedings of the Fifth International Workshop on Rule-Based Programming (RULE 2004)

2004-05 Herbert Kuchen (ed.): Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)

2004-06 Sergio Antoy / Yoshihito Toyama (eds.): Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004)

2004-07 Michael Codish / Aart Middeldorp (eds.): Proceedings of the 7th International Workshop on Termination (WST 2004)

* These reports are only available as a printed version.

Please contact biblio@informatik.rwth-aachen.de to obtain copies.