View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NORA - Norwegian Open Research Archives

A Meta-Level Framework for Recording and Utilizing Communication Histories in Maude

Eyvind Wærsted Axelsen

Cand. Scient. Thesis
4th August 2004

University of Oslo
Department of Informatics

Preface

This thesis was written between January 2003 and August 2004, and is part of a Cand. Scient. degree at the Department of Informatics, University of Oslo.

I would like to extend my gratitude to my supervisors, Einar Broch Johnsen and Olaf Owe, for patience, valuable feedback, inspiration and generally for a good time.

Oslo, 4th August 2004,

Eyvind Wærsted Axelsen

Contents

1 Introduction **1**
- 1.1 Problem domain . . . 1
  - 1.1.1 Maude . . . 1
  - 1.1.2 Creol . . . 2
- 1.2 Motivation . . . 2
- 1.3 Goals . . . 3
- 1.4 Contents of this thesis . . . 3
- 1.5 Results . . . 4

2 Background **5**
- 2.1 Maude . . . 5
  - 2.1.1 Functional Maude . . . 5
  - 2.1.2 Rewriting logic in Maude . . . 9
  - 2.1.3 Maude's Meta-level . . . 11
- 2.2 Creol . . . 22
  - 2.2.1 Classes and objects . . . 22
  - 2.2.2 Interfaces . . . 22
  - 2.2.3 Method calls . . . 23

3 Communication Histories **25**
- 3.1 Extending Maude specifications with histories . . . 27
  - 3.1.1 Where are the histories stored? . . . 27
  - 3.1.2 When do we record message events in the history? . . . 28
  - 3.1.3 How do we build such a history during execution? . . . 29

4 A Meta-Level Rewrite Strategy for Recording the Communication History **33**
- 4.1 The *Engine* object . . . 34
- 4.2 The *History* object . . . 37
- 4.3 Performance . . . 42
  - 4.3.1 Test results . . . 43

5 **Predicates on Finite Communication Histories** **51**
5.1 Data structures for predicate construction 52
5.2 Projections on the history 54
5.3 Parsing and checking the predicates 56
5.4 Integrating predicates with the rewrite strategy 59

6 **Regular Expressions in Predicates** **61**
6.1 Data structures for regular expressions 61
6.2 Regular patterns 62
6.3 Non-deterministic finite automata 62
6.4 Deterministic finite automata 68
6.5 Putting the regular expressions to work in predicates 74

7 **Examples** **77**
7.1 Simple producer-consumer specification 77
7.2 The dining philosophers 78

8 **Extensions to the Predicate Framework** **85**
8.1 Quantifiers 85
8.2 Adapting the framework for use with the Creol interpreter . . 90
8.2.1 Creol's message format 91
8.2.2 Completion messages 92
8.2.3 An example: The dining philosophers in Creol 93
8.3 Component testing and abstract environments 96
8.3.1 An example: Abstract dining philosophers 99
8.4 Parameters and variables in predicates 105
8.4.1 Parameters 105
8.4.2 Variables 106
8.4.3 An example: The alternating bit protocol 114

9 **Non-deterministic Execution** **121**
9.1 A Maude module for generating pseudo-random numbers . . . 122
9.2 Pseudo-random selection of rewrite rules 124
9.2.1 An example: The dining philosophers with randomized rule selection 128
9.3 Randomized rule application 132
9.3.1 An example: The dining philosophers with only one rewrite rule 132
9.3.2 Randomization between and within objects 134

10 **Communication over Sockets** **139**

11 **Conclusion** **145**
11.1 Main results from this thesis 145
11.2 Future work 147

Bibliography **149**

A Source Code **153**
A.1 Rewrite strategy . . . 153
A.2 Predicates . . . 160
A.3 Regular expressions . . . 165
A.4 Pseudo-random number generator . . . 183
A.5 Auxiliary modules . . . 184
A.6 Python code for socket communication . . . 190

B Papers **195**

List of Figures

2.1 A module in which the rewrite rule *age* models the aging of John Doe, one year at the time 10

3.1 A simple server interface in Java 26
3.2 A rewrite rule that sends a message *'M* from *'S* to *'R* 30
3.3 A new version of the rewrite rule from Figure 3.2 that makes use of an object *'H* that maintains a global history of messages 30

4.1 Outline of the rewrite strategy 34
4.2 A rewrite strategy that applies rewrite rules in a round robin fashion 36
4.3 A rewrite strategy which records a communication history as the execution proceeds 41
4.4 Counter performance test 43
4.5 Counter performance test with only one applicable rule out of ten 44
4.6 Performance test specification containing a configuration of objects that send messages in a ring 45

5.1 A rewrite strategy that checks whether rewrites will lead to a state that violates the predicate 60

6.1 An NFA for the regular expression a 63
6.2 An NFA for the concatenation of the regular expressions r and s 63
6.3 A Maude representation of an NFA for the regular expression $(a \mid b)^* :: c$. 67
6.4 A Maude representation of a DFA for the regular expression $(a \mid b)^* :: c$. 73

8.1 A rewrite strategy that can check predicates in two different modes, *force* and *fail-stop* 100
8.2 A non-deterministic finite automaton for the regular expression $int(n)^*$. 109
8.3 A deterministic finite automaton for the regular expression $int(n)^*$. 109

8.4 A non-deterministic finite automaton for the regular expression *scope*(*int*(x)) *int*(x) *endscope* *. 110
8.5 A pseudo code outline of the base *Match* equation for NFA matching, for which the message list parameter is the empty list *nil* 113
8.6 A pseudo code outline of the general recursive *Match* equation for NFA matching 115

9.1 A Maude module for generating pseudo-random numbers . . . 123
9.2 A rewrite strategy with randomized rule selection 129
9.3 A rewrite strategy with randomized rule and solution number selection 137

10.1 A schematic view of socket communication in Maude 140
10.2 An example XML file with locations on the network for two objects 141
10.3 An example setup with two Maude processes communicating across the network through sockets 143

Chapter 1

Introduction

This thesis is part of the ongoing *Creol* [26, 24] research project at the Precise Modeling and Analysis (PMA) group at the Department of Informatics, University of Oslo. Creol is an acronym for Concurrent Reflective Object-oriented Language, and is focused towards programming constructs and reasoning control with regards to the development of open distributed systems. An interpreter for (a subset of) the Creol language [3] has been developed in the rewriting logic tool Maude [7].

1.1 Problem domain

In this thesis, we will consider objects that communicate *asynchronously* through message passing. Objects and messages are represented by a global state called a *configuration*. The configuration is a multiset that models a highly non-deterministic system with concurrent objects. This model is well suited for both standard object based Maude specifications and Creol programs executed in Maude by the Creol interpreter.

The messages that the objects send, can be recorded by an *external observer* in a *communication history*. By specifying predicates on this history, we can define invariants for an object's behavior, as well as an object's assumptions with regards to the behavior of its surrounding environment.

1.1.1 Maude

All the specifications that we will be considering in this thesis are written in Maude, either as standard Maude code, or as Maude representations of Creol code. Maude is a powerful high-level programming and specification language based on *rewriting logic* [32]. Rewriting logic is a logic in which concurrent change and non-deterministic problems can be specified in a natural

way, and this fits well with our focus on concurrent distributed objects.[1]

Furthermore, since rewriting logic is *reflective* [9], Maude specifications can be used to examine, modify, execute and reason about other Maude specifications. This is known as *meta-programming*, or programming at the *meta-level*. In this thesis, we will make extensive use of Maude's meta-level capabilities.

1.1.2 Creol

The Creol language focuses on open distributed systems, and employs an object model in which every object is seen as having its own processor, and its own thread (or threads) of execution.

Creol objects communicate synchronously or asynchronously through method calls. The language supports *black box* specification of objects in terms of their *observable behavior*, as defined in their respective interfaces.

The operational semantics of Creol is formally defined in rewriting logic.

1.2 Motivation

The Creol interpreter, running on top of Maude, can execute Creol programs represented in Maude as *Creol Machine Code* [3]. An important part of Creol interfaces, is *assumption-guarantee* specifications [28]. Such specifications are expressed as predicates on a communication history. These predicate specifications, however, are ignored by the current interpreter. Hence, much of the motivation for this thesis stems from a desire to experiment with these features. For this purpose, we will devise a mechanism for recording a communication history from an executing specification, and constructs for specifying predicates on the history. We will also consider how the predicates can be used to control the execution of specifications.

Another motivating factor was the desire to develop a strategy that allows for a fairer execution of the Creol programs executed with the interpreter. In [3], we see that using Maude's built-in *rew* and *frew* rewrite strategies for executing non-deterministic problems results in skewed and unfair results.

Since both the interpreter and the Creol language itself are under continuous development, it would be advantageous if the mechanisms discussed above were implemented separately, instead of changing the actual interpreter code.

Due to the fact that the Creol interpreter executes on top of Maude, we have chosen to not limit the concepts introduced in this thesis to Creol only, but instead provide a general framework for executing object based Maude

[1] However, rewriting logic specifications of non-deterministic problems cannot as easily be *executed* in a satisfying manner in Maude, which is a deterministic tool. We will get back to this in the following chapters of this thesis.

specifications, and to make the necessary adaptations to account for Creol as well.

1.3 Goals

The main goal for this thesis is to develop a framework for executing specifications modeling distributed systems, that can record and utilize a communication history. With this in mind, we can rephrase the goal as several more specific questions:

- How can we execute Maude specifications and *transparently*, in the sense that the original specification remains unchanged, build a communication trace as the execution proceeds?
- How can we define predicates on this trace, and use such predicates to control and test the behavior of objects?
- How can these techniques be applied to the Creol language, and more specifically, to the Creol interpreter developed in Maude?
- How can we execute models of highly non-deterministic concurrent systems, such as Creol programs, in the deterministic rewrite tool Maude?

In the following chapters of this thesis, solutions to the problems presented above will be developed, and we will get back to the specific questions in the conclusion in Chapter 11.

1.4 Contents of this thesis

A brief summary of the contents of the individual chapters of this thesis is given below:

- In Chapter 2 we take a look at the languages Maude and Creol, to provide necessary background information for the concepts that will be introduced in the following chapters.
- In Chapter 3, we consider communication histories at a generalized level. Furthermore, we discuss how the recording of such histories should be done.
- The implementation of the concepts from Chapter 3 is discussed in Chapter 4. A rewrite strategy that records a communication history during run-time is developed in Maude.
- Predicates on communication histories are introduced in Chapter 5. The predicates are used by the rewrite strategy from the previous chapter to control the execution.

- In Chapter 6 we look at how regular expressions can be used in predicates on communication histories.
- Some examples of how the mechanisms developed in the preceding chapters can be used, are considered in Chapter 7.
- In Chapter 8, some additional predicate constructs are considered. Furthermore, we discuss how the Creol interpreter can be integrated with the rewrite strategy that we have developed up till this point.
- A rewrite strategy that uses a pseudo-random number generator to provide fair rewriting suitable for highly non-deterministic problem domains is introduced in Chapter 9.
- In Chapter 10, we discuss how Maude can be extended with socket support, and how this can be used together with the framework we have developed.
- Finally, in Chapter 11, we summarize the most important concepts and results from this thesis.

1.5 Results

A framework containing a non-deterministic meta-level rewrite strategy, a communication history logger and a predicate parsing and checking engine has been developed and is executable in the Maude rewriting logic tool. The framework supports both standard Maude specifications and Creol specifications executed by the Creol interpreter. The source code for the framework is included in Appendix A.

Results from this thesis have contributed to two scientific papers, *Toward Reflective Application Testing in Open Environments* [4] and *A Run-Time Environment for Concurrent Objects with Asynchronous Method Calls* [24]. Both papers are included in Appendix B.

Chapter 2

Background

In this chapter, we will take a look at some properties of the languages Maude and Creol. We will focus on the aspects that will be of importance to us in the rest of this thesis; this is in other words not a general overview of any of the languages.

2.1 Maude

Maude [7] is a high level programming and specification language, based on *rewriting logic* [32]. It is highly expressive, due to a syntax that is entirely user definable, and can be used as a tool for modeling both deterministic and concurrent non-deterministic problem domains. Maude contains a *functional sublanguage* based on the OBJ3 language [17] for equational specifications, and extends OBJ3 by providing *rewrite rules* that capture *concurrent change*.

2.1.1 Functional Maude

The *functional modules* in Maude are theories in *membership equational logic* [7, 6] that satisfy some additional properties. Such a module consists of sort declarations and equations. Terms are reduced by using the equations in the module on a given term until a canonical normal form is found and no equations can be applied. Hence, the extra requirement for equations in functional modules is that they should be Church-Rosser (there is a unique normal form for every term), terminating and sort-decreasing [7, 6] (even if a given specification does not satisfy these properties, Maude will not complain, but the results from computations may be undesirable). All reductions and rewrites can be performed modulo associativity, commutativity and identity, if the attributes *assoc*, *comm* and *id* are specified for the equations, respectively.

Below we will take a closer look at the components of a functional module. The definitions are from [38, 32, 7]:

A *sort* is declared in a Maude module by using the *sort* keyword, e.g. *sort Nat*.

Definition 1 (Sorts). *The sorts in a specification are defined by a set S of sorts (or sort names).*

Note that the sorts are *just* names, it is the function symbols that define the values of each sort. So, in the example above, the sort *Nat* is just an arbitrary name, it has no associated values yet.

A *signature* consists of sorts and function (or operator) symbols.

Definition 2 (Signature). *A many-sorted signature (S, Σ) consists of a set S of sorts and an $S^* \times S$-sorted family $\{\Sigma_{w,s} | w \in S^*, s \in S\}$ of function symbols.*

It is conventional to write $f : w \rightarrow s \in \Sigma$ for $f \in \Sigma_{w,s}$. A function symbol for which w is the empty list is called a *constant*.

The values in a specification are made from *ground terms*, which are constructed from function symbols.

Definition 3 (Ground terms). *Given a many-sorted signature (S, Σ), the ground terms of any sort $s \in S$ is denoted $\mathcal{T}_{\Sigma,s}$, and is defined inductively as follows:*

1. *$\Sigma_{\epsilon,s} \subseteq \mathcal{T}_{\Sigma,s}$; that is, every constant of sort s is a ground term of sort s.*
2. *If $f \in \Sigma_{s_1...s_n,s}$, and $t_1 \in \mathcal{T}_{\Sigma,s_1}, ..., t_n \in \mathcal{T}_{\Sigma,s_n}$, and $n \geq 1$ then $f(t_1, ..., t_n) \in \mathcal{T}_{\Sigma,s}$. That is, a function symbol applied to ground terms of the correct sort gives another ground term.*
3. *In addition, each set $\mathcal{T}_{\Sigma,s}$ is the smallest set satisfying the above conditions. That is, only "things" which can be built from constants and the application of function symbols to ground terms of the right sorts are ground terms.*
4. *Finally, the ground terms $\mathcal{T}_\Sigma$ of the many-sorted signature (S, Σ) are defined by $\mathcal{T}_\Sigma = \{\mathcal{T}_{\Sigma,s} | s \in S\}$.*

A *term* is is constructed in a sort-correct manner from variables and ground terms.

Definition 4 (Variables). *Given a many-sorted signature (S, Σ), a variable set X is an S-sorted family $X = \{X_s \mid s \in S\}$ of pairwise disjoint sets (meaning that no variable has two different sorts: $s \neq s' \Rightarrow X_s \cap X_{s'} = \emptyset$), also disjoint from Σ (that is, nothing can be both a variable and a function symbol).*

Definition 5 (Terms). *Given a many-sorted signature* (S, Σ) *and a variable set* $X = \{X_s \mid s \in S\}$, *the S-sorted set of terms* $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ *is defined inductively by the following conditions:*

1. $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ *for* $s \in S$; *that is, a variable of sort* s *is also a term of sort* s.

2. $\Sigma_{\epsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ *for* $s \in S$; *that is, a constant of sort* s *is also a term of sort* s.

3. $f(t_1, ..., t_n) \subseteq \mathcal{T}_{\Sigma,s}(X)$ *if* $f \in \Sigma_{s_1 \ldots s_n,s}$ *and* $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ *for each* $1 \leq i \leq n$.

4. $\mathcal{T}_{\Sigma,s}(X)$ *is the smallest S-sorted set satisfying the above conditions.*

Functions declared in a signature are defined recursively by equations.

Definition 6 (Equations). *Given a many-sorted signature* (S, Σ) *(without subsorts), a* (Σ)*-equation is a triple* (X, t, t'), *written* $(\forall X)t = t'$, *where* X *is an S-sorted variable set disjoint from* Σ, *and* t *and t' are terms of the same sort.*

Finally, we are ready to *define many-sorted equational specifications*, which correspond to Maude modules.

Definition 7 (Many-sorted equational specifications). *A many-sorted equational specification is a tuple* (S, Σ, E) *where* (S, Σ) *is a many-sorted signature and* E *is a set of* Σ*-equations and conditional* Σ*-equations.*

As an example of a functional module, we take a closer look at the module *BOOLEAN*:[1]

```
fmod BOOLEAN is
    sort Boolean .

    op true  : -> Boolean .
    op false : -> Boolean .

    op _and_ : Boolean Boolean -> Boolean
       [assoc comm prec 55 id: true] .
    op _or_ : Boolean Boolean -> Boolean
       [assoc comm prec 59 id: false] .
    op _xor_ : Boolean Boolean -> Boolean
```

[1]Note that this is not the same module as the built-in BOOL module, which will be used later on. The BOOLEAN module is only meant to exemplify the concepts in this section.

```
      [assoc comm prec 57] .
   op not_ : Boolean -> Boolean [prec 53] .
   op _implies_ : Boolean Boolean -> Boolean
      [prec 61] .

   vars A B C : Boolean .

   eq true and A = A .
   eq false and A = false .
   eq A and A = A .
   eq false xor A = A .
   eq A xor A = false .
   eq A and (B xor C) = A and B xor A and C .
   eq not A = A xor true .
   eq A or B = A and B xor A xor B .
   eq A implies B = not (A xor A and B) .
endfm
```

In this module, only one sort is defined, the sort *Boolean.* There are two constants defined of this sort, *true* and *false.* Furthermore, there are four binary operators (*and, or, xor* and *implies*) and one unary operator (*not*) defined by equations. We also have the three *Boolean* variables that are used in the equations, *A, B* and *C.*

For the operators, various attributes are specified, such as associativity (*assoc*), commutativity (*comm*), identity (*id: id-element*) and precedence (*prec n*). For the latter, the operator with the lowest n takes precedence for terms with several possible parse trees.

Maude supports a so-called *mixfix* syntax, meaning that arguments for an operator can be placed at user-defined positions. Argument placement is indicated by an underscore (_), as in the *_and_* operator above, or the built-in *if_then_else_fi* operator. If no underscores are supplied in an operator declaration, arguments are placed in parenthesis behind the operator in a traditional manner.

Several useful built-in modules are supplied with Maude, below are a few examples:

- The *BOOL* module closely resembles our *BOOLEAN* module from above, and provides the constants *true* and *false* and all the standard boolean operators. This module is imported into all other modules unless you explicitly state that it should not be.
- For representing integers, the *INT* module is used.
- *QID* is another built-in module, and contains a sort *Qid*, which is the sort of the so-called *quoted identifiers.* A quoted identifier is a string

(that cannot contain whitespace) with a quote as its first character, e.g. *'hello-world*. This sort, although not very common in other languages, is an integral part of Maude, and *Qid*s are useful in many situations, as we shall see several examples of later on.

2.1.2 Rewriting logic in Maude

Rewriting logic [32] is a logic of concurrent change, that in a natural way can model highly non-deterministic problem domains. In Maude, specifications in rewriting logic are contained within so-called *system modules*.

In functional modules, all the equations should, as mentioned, be terminating and confluent. This is not, however, the case when it comes to general rewrite rules. In rewriting logic, it is allowed for specifications to be both non-terminating and non-confluent. That is, infinite rewrite paths and divergent rewriting paths that may never again meet are fully acceptable.

In an equational specification, the equations represent *equality*. In a rewriting logic specification, the rewrite rules are interpreted as *local state transitions* [7]. This means that a rule of the form $t \to t'$ is not symmetric, it can only be applied in one direction, as opposed to the equation $t = t'$. Furthermore, a given transition can take place independently of and concurrently with any other non-overlapping local state transition in the system.

Definition 8 (Rewrite Theory). *A rewrite theory $\mathcal{R}$ is a 4-tuple (Σ, E, L, R), in which Σ is an equational signature, E is a set of Σ-equations, L is a set of labels and R is a set of labeled rewrite rules.*

System modules are the most general form of Maude modules. A system module specifies a rewrite theory. As an example, consider the module in Figure 2.1 on the following page, which models the process of a person aging one year at the time. It should be clear that this process cannot be captured by a symmetric equation.

In this figure, the signature Σ consists of all the sorts, subsort relations and operator declarations in the modules *INT* and *QID* as well as the sort *Person* and the declarations for the operators *person* and *init*. E is the equations in the modules *INT* and *QID* together with the equation for the *init* operator in the figure. Since both *INT* and *QID* are functional modules, they contain no rule labels or rules, hence the set L consists only of the label *age*, and the set of rules, R, consists of only the rule that is labeled *age*.

Given a set E of Σ-equations, $T_{\Sigma,E}$ represents the equivalence class of terms $\{[t]_E \mid t \in T_\Sigma\}$. The E-equivalence class of t is conventionally denoted $[t]_E$, or just $[t]$.

Given a rewrite theory $\mathcal{R}$, the sequent $\mathcal{R} \vdash [t] \to [e]$ holds if and only if $[t] \to [e]$ can be deducted by application of the following rules:[2]

[2] In the same way as in [32], the *unsorted* case is treated. Many-sorted and order-sorted

```
mod AGING is
  protecting INT .
  protecting QID .
  sort Person .

  op person : Qid Int -> Person [ctor] .
  op init : -> Person .

  *** John Doe is 30 years old:
  eq init = person('JohnDoe, 30) .

  var NAME : Qid .
  var AGE : Int .
  rl [age] :
     person(NAME, AGE) => person(NAME, AGE + 1) .
endm
```

Figure 2.1: A module in which the rewrite rule *age* models the aging of John Doe, one year at the time

Definition 9 (Deduction rules of rewriting logic).

1. *Reflexivity: For every* $[t] \in T_{\Sigma,E}(X)$

$$\frac{}{[t] \rightarrow [t]}$$

2. *Congruence: For every function symbol* $f \in \Sigma_n$

$$\frac{[t_1] \rightarrow [t'_1] ... [t_n] \rightarrow [t'_n]}{[f(t_1, ..., t_n)] \rightarrow [f(t'_1, ..., t'_n)]}$$

3. *Replacement: For each rewrite rule* $r : [t(x_1, ..., x_n)] \rightarrow [t'(x_1, ..., x_n)]$ *in* R

$$\frac{[w_1] \rightarrow [w'_1] ... [w_n] \rightarrow [w'_n]}{[t(w/x)] \rightarrow [t'(w'/x')]}$$

where x *represents variables* $x_1, ..., x_n$*,* w *represents terms* $w_1, ..., w_n$*, and* w/x *is the substitution of* w *for* x*.*

4. *Replacement for conditional rewrite rules: For each rewrite rule*

cases can be treated similarly.

$$r : [t(x)] \rightarrow [t'(x)] \ \textit{if} \ [u_1(x)] \rightarrow [v_1(x)] \wedge ... \wedge [u_k(x)] \rightarrow [v_k(x)] \ \textit{in} \ R$$

$$\frac{[w_1] \rightarrow [w'_1]...[w_n] \rightarrow [w'_n] \quad [u_1(w/x)] \rightarrow [v_1(w/x)]...[u_k(w/x)] \rightarrow [v_k(w/x)]}{[t(w/x)] \rightarrow [t'(w'/x')]}$$

where x represents variables $x_1, ..., x_n$, w represents terms $w_1, ..., w_n$, and and w/x is the substitution of w for x.

5. *Transitivity:*

$$\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Note that equational logic (modulo a set of axioms) can be obtained from rewriting logic by adding the symmetry rule:

$$\frac{[t_1] \rightarrow [t_2]}{[t_2] \rightarrow [t_1]}$$

Maude comes with two built-in deterministic rewrite strategies, implemented by the commands *rew* and *frew*. Both strategies rewrite terms in a specification according to the specification's rewrite rules modulo its equations.

The rewrite command *rew* rewrites terms using a leftmost outermost approach to applying the rewrite rules. This will in many cases provide a very skewed result, and will often leave some rewrite rules unused.

The fair rewrite command *frew* attempts to be fairer than *rew* by using a depth-first position-fair strategy for applying rewrite rules. Even though the strategy implemented by this command is arguably fairer than *frew*, it is not fair enough for specifications that model concurrent, highly non-deterministic problem domains, as we shall see examples of later on. In Chapter 9 a strategy that uses a pseudo-random number generator for applying rules is proposed.

In addition to the rewrite strategies, Maude also provides a command *red* for reducing terms using equations only.

2.1.3 Maude's Meta-level

A meta-program can, informally, be seen as a program that takes a representation of another program as input, or returns a representation of a program as its output, or both. Hence, a meta-program can be used to manipulate, reason about, examine and execute other programs, provided a sensible representation of such programs can be given.

In this area, Maude excels. Any valid Maude program can quite easily be represented at the meta-level using standard Maude syntax, and as such it can also be manipulated using standard Maude mechanisms.

To provide it's meta-programming capabilities, Maude makes use of the fact that rewriting logic is reflective [9]. This implies, quoting [10], that

> there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) (...) in $\mathcal{U}$.

The meta-representation of a term t is conventionally denoted $\overline{t}$, and the meta-representation of a module $\mathcal{R}$ is likewise denoted $\overline{\mathcal{R}}$. Using this notation, we have the following equivalence [10]:

$$(\dagger)\ \mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

In other words, if a term t can be rewritten to a term t' in a rewrite theory $\mathcal{R}$, then the meta-representation of t in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ can be rewritten to the meta-representation of t' in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$, in the universal rewrite theory $\mathcal{U}$, and vice versa.

Furthermore, since $\mathcal{U}$ is universal, it can represent itself. This means that a term t in $\mathcal{R}$, meta-represented in $\mathcal{U}$ as $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, can be meta-meta-represented in $\mathcal{U}$ as $\langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle$, and meta-meta-meta-represented as $\langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle} \rangle$ and so on. Hence, any term can be meta-represented an arbitrary number of times.

Meta-representing terms in Maude

Terms are represented at the meta-level by a data type *Term*. This data type is made up of several operators, each corresponding to different types of terms. The presentation below is based on [7, 8, 38].

Variables Since variables can only have one sort in any given module, they can be represented simply by a quoted identifier. Hence, it is sufficient to declare the sort *Qid* a subsort of *Term* to be able to meta-represent variables. Thus, a variable X is meta-represented as *'X*.

For clarity and readability, though, it may still be convenient to have the variable's sort immediately available with the representation of the variable. Maude makes this possible by allowing the sort of a variable to follow its name, separated by a colon, and a variable X of sort *Bool* may be meta-represented as *'X:Bool*.

Constants As opposed to variables, a constant declaration may be overloaded in a given module. For example, the constant *none* may very well be declared as being of both sort *Configuration* and *GenericMultiSet*; in other words, the declarations

```
op none : -> Configuration .
```

and

```
op none : -> GenericMultiSet .
```

are perfectly legal in the same module. For this reason, constants are meta-represented by their quoted version followed by their sort.[3] For example, the constant *none* of sort *Configuration* is meta-represented as

```
'none.Configuration
```

Compound Terms The meta-representation of a compound term

$$f(t_1,\ \dots,\ t_n)$$

(in which each of the subterms t_1 to t_n may also be a compound term) is defined by an operator

```
op _[_] : Qid TermList -> Term .
```

The sort of the second argument of this operator, a *TermList*, is defined as follows:

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

Mixfix symbols such as the built-in + and *if then else* are meta-represented in their prefix form as `'_+_` and `'if_then_else_`, respectively.

To illustrate this concept, we use an example from [10]; the term $s(s(0))+s(0)$ in the built-in module *NAT* is meta-represented in the following way:

```
'_+_['s['s['0.Nat]], 's['0.Nat]].
```

[3] In versions of Maude prior to 2.0, a slightly more involved notation for constants was used, utilizing the following operator signature: `op {_}_ :  Qid Qid -> Term`.

Objects Maude provides support for object oriented specification. In this thesis, we will to some extent be using Maude's standard notion of objects communicating by means asynchronous message passing

An object in Maude is a compound term, and as such it will be meta-represented according to the rules stated above. However, since an object may be quite complex, with quite a few attributes, its meta-representation may look rather "ugly" to the human eye. As we will be seeing quite a lot of these later in this thesis, we will look at an example here to familiarize ourselves with their appearance.

Consider a simplified object representing a philosopher, from the well-known *dining philosophers* problem (for now, just consider the object, we will get back to the problem in detail in Section 7.2):

```
< 'Socrates : Philosopher | hungry : true, leftstick: false,
  rightstick: false >
```

As we can see, the philosopher object has an identifier of the built-in sort *Qid* (for *quoted identifier*), *'Socrates*, and three boolean attributes indicating whether he is hungry or not and whether he is currently in possession of his left and right chopstick, respectively. Below is the meta-representation of this simple object:

```
'<_:`Philosopher`|`hungry:_`,leftstick:_`,rightstick:_>
  [''Socrates.Qid,'true.Bool,'false.Bool,'false.Bool]
```

Note the use of the *back quote* (`) as an escape code for characters with special meaning in Maude, such as whitespace and comma. This should not be confused with the standard *forward quote* (') which is used for quoted identifiers (*Qid*s).

Messages In the same way as objects, Maude's conventional representation of messages are as compound terms. Hence, given a standard object-level message with contents M from *'A* to *'B*,

```
msg M from 'A to 'B
```

we have the following meta-level representation:

```
'msg_from_to_['M.Msg, ''A.Qid, ''B.Qid].
```

Meta-representing modules in Maude

A module specification consists of several distinct parts. Below, we will look at the two (from our point of view) most interesting of them; equations and rewrite rules, before we look at how to meta-represent an entire module. The interested reader may always consult the Maude manual [11] for a detailed description of all the parts.

Equations An equation is meta-represented as a term of sort *Equation*, and may be constructed from one of two operators; there is one operator for unconditional equations and one for conditional equations:

```
op eq_=_[_]. : Term Term AttrSet -> Equation [ctor] .
op ceq_=_if_[_]. : Term Term EqCondition AttrSet ->
  Equation [ctor] .
```

As we can see, the operators use terms as defined above to construct the equations. As an example, consider one of the simple equations for the exclusive or operator, *xor*, from the built-in module *BOOL*:

```
eq false xor A = A .
```

Below is the meta-representation of this equation:

```
eq '_xor_['false.Bool, 'A:Bool] = 'A:Bool [none] .
```

Rewrite rules A rewrite rule is represented in much the same way as an equation. There are two constructors, one for the unconditional case and one for the conditional case, respectively:

```
op rl=_[_]. : Term Term AttrSet -> Rule [ctor] .
op crl=_if_[_]. : Term Term Condition AttrSet -> Rule [ctor] .
```

Modules The representation of modules at the meta-level closely follows the standard Maude syntax for modules; the perhaps most important difference (apart from the fact that all rules, equations etc. must be meta-represented as well) is that the different declarations must be given in a specific order, as opposed to the quite liberal approach taken by the standard Maude syntax with regards to this.

The constructor for a meta-level system module is defined as follows:

```
op mod_is_sorts_._____endm :
  Qid ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet RuleSet -> Module .
```

Although the syntactic details are not of highest importance to us at this point, we note that equations and rules as defined above are gathered in sets of sort *EquationSet* and *RuleSet* respectively. These sets, and the rest of the sets used in the declaration, are defined as is standard in Maude: with an infix constructor that is associative and commutative and with an identity element *none*.

As an example, we will look once more at the module aging from Figure 2.1 on page 10, which models the process of a person aging one year at the time. Below is the meta-representation of this module:

```
mod 'AGING is
  protecting 'BOOL .
  protecting 'QID .
  protecting 'INT .
  sorts 'Person .
  none
  op 'init : nil -> 'Person [none] .
  op 'person : 'Qid 'Int -> 'Person [ctor] .
  none
  eq 'init.Person = 'person[''JohnDoe.Qid,
     's_^30['0.Zero]] [none] .
  rl 'person['NAME:Qid, 'AGE:Int] =>
     'person['NAME:Qid, '_+_['AGE:Int,
     's_['0.Zero]]] [label('age)] .
endm
```

As mentioned, due to the fact that rewriting logic is reflective, the meta-representation of terms can be iterated an arbitrary number of times. Hence, even the meta-representation of sort *Module* above can be meta-meta-represented as a compound term of sort *Term*, as shown below:

```
'mod_is_sorts_._____endm[
   ''AGING.Qid,
   '__['protecting_[''BOOL.Qid], 'protecting_[''QID.Qid],
      'protecting_[''INT.Qid]],
   ''Person.Qid,
   'none.SubsortDeclSet
   '__['op_:_->_`[_`].[''init.Qid, 'nil.TypeList,
                       ''Person.Type,'none.AttrSet],
       'op_:_->_`[_`].[''person.Qid, '__[''Qid.Type,
                       ''Int.Type],  ''Person.Type,
                       'ctor.Attr]],
   'none.MembAxSet,
   '__['eq_=_`[_`].[''init.Person.Term,
                    ''person`[''JohnDoe.Qid`,'s_^30`[
                      '0.Zero`]`].Term, 'none.AttrSet]]
   '__['rl_=>_`[_`].[''person`['NAME:Qid`,'AGE:Int`].Term,
                     ''person`['NAME:Qid`,'_+_`['AGE:Int`,
                     's_`['0.Zero`]`]`]`[label('age)`].Term]]
]
```

Descent-functions

Maude allows not only for programs to be represented at the meta-level; meta-modules can also be executed as were they standard object-level modules, through the use of the so-called *descent functions*.

A descent function is, in general, a function that given a meta-level rewrite theory $\overline{\mathcal{R}}$ and a term $\overline{t}$ in $\overline{\mathcal{R}}$, rewrites the term in accordance with a given strategy, and returns the meta-representation of the resulting term [7].

Descent functions in Maude exploit the equivalence (†) presented earlier in this section in order to perform meta-level computations at the object level in a systematic way. This allows for efficient rewrites even for terms that are meta-represented several times [7]. In other words, when a meta-level computation is to take place, the equivalence (†) is used to lower the reflective level as far as possible, preferably all the way down to the object level, before the actual rewrites are performed.

A general descent function d can be expressed in terms of a general *sequential interpreter function* I for rewriting logic [7]. I is a partial function, and takes three arguments: a finitely presented rewrite theory $\mathcal{R}$, a term t and a deterministic strategy S. I satisfies the correctness requirement

$$(\ddagger)\ I(\mathcal{R},\mathrm{t},S)\ = \mathrm{t'} \Rightarrow\ \mathcal{R} \vdash \mathrm{t} \rightarrow \mathrm{t'}\ [7].$$

The function d : *Module* × *Term* × *Parameters* → *Term* can then be defined by an equation

$$d(\overline{\mathcal{R}},\overline{t},p) = \overline{I(\mathcal{R},t,\mathcal{S}_d(p))}$$

where S_d is a deterministic strategy with a single free variable p of sort *Parameters* (which may in fact be a list of parameters).

Maude comes with several built-in descent functions that are provided in the module *META-LEVEL*. Below we will look at a selection of these.

metaReduce The function *metaReduce* : *Module* × *Term* → *ResultPair* is a descent function that provides the same functionality at the meta-level as the *red* command provides at the object level. In other words, it allows for terms at the meta-level to be reduced to their meta-level normal form. Hence, the interpreter function for *metaReduce* is Maude's internal interpreter function:

$$\mathit{metaReduce}(\overline{\mathcal{R}},\overline{t}) = \overline{I_{\mathrm{Maude}}(\mathcal{R},t,\mathrm{red})}\ [10].$$

The result returned by *metaReduce* is a pair containing the meta-representation of the reduced term, and the meta-representation of its sort.

metaRewrite Corresponding to *meta-reduce*, the descent function *metaRewrite : Module × Term × Int → ResultPair* provides the functionality of the Maude's *rew* command at the meta-level, allowing for a specification to be rewritten according to its rewrite rules modulo its equations:

$$metaRewrite(\overline{\mathcal{R}}, \overline{t}, n) = \overline{I_{\text{Maude}}(\mathcal{R}, t, \text{rewrite } [n])} \text{ [10]}$$

where n is a positive integer specifying the maximum allowed number of applications of the rewrite rules in $\mathcal{R}$. If the value 0 is given for n, the rewriting will continue until the execution terminates according to the rules in the specification (meaning that no rule or equation is applicable), or if the specification is non-terminating, infinitely (at least in theory).

metaXapply The (partial) operation *metaXapply* allows for more fine-grained control over the rewriting at the meta-level, compared to the aforementioned functions. Taking as arguments the meta-representation of a module, the meta-representation of a term, the meta-representation of a rule label, the meta-representation of a possibly empty set of assignments (a partial substitution), a natural number and a value of sort *Bound* representing a lower and upper bound for the position in the term where the rewrite may be applied, respectively, and finally another natural number representing a solution number, *metaXapply*($\overline{\mathcal{R}}$, $\overline{t}$, $\overline{l}$, σ, n, b, m) makes it possible to control which rewrite rule is applied to a given term at a given position within a module [11].

As an example, consider the following simple Maude rewrite specification for a population of three people, John Doe, Jane Doe and little Jonathan Doe, that are all aging through the application of the *age* rewrite rule, and die when the *die* rewrite rule is applied:

```
mod AGING-2 is
  protecting INT .
  protecting QID .

  sorts Person Configuration .
  subsort Person < Configuration .

  op __ : Configuration Configuration -> Configuration
    [ctor assoc comm id: none] .
  op person : Qid Int -> Person [ctor] .
  op none : -> Configuration .
  op init : -> Configuration .

  eq init = person('JohnDoe, 30) person('JaneDoe, 28)
    person('JonathanDoe, 4) .
```

```
  var NAME : Qid .
  var AGE : Int .
  rl [age] :
     person(NAME, AGE) => person(NAME, AGE + 1) .
  rl [die] :
     person(NAME, AGE) => none .
endm
```

From the initial state *init*, the rewrite rule *age* may be applied once at three different positions within the term; John can age one year, Jane can age one year, or Jonathan can age one year. Also, either one of the persons may die. Which of these rewrites that will actually take place, is in rewriting logic non-deterministic. Maude, however, is a deterministic tool running on a deterministic machine, and will always choose the same solution. Using *metaXapply*, we are able to control this by specifying the rule label l ('age) and the solution number m (2) ourselves, as shown below:

```
red metaXapply(['AGING-2], 'init.Configuration,
  'age, none, 0, unbounded, 2)
```

The result returned from a successful application of *metaXapply* is a four-tuple consisting of the rewritten term, the type of the term, a substitution and a context that shows where in the term the rewriting took place. Each of the components of the tuple is available through the use of the following functions [11]:

```
op getTerm : Result4Tuple -> Term .
op getType : Result4Tuple -> Type .
op getSubstitution : Result4Tuple -> Substitution .
op getContext : Result4Tuple -> Context .
```

So, to conclude our example, the resulting term from the application of the *age* rule at position 2 using *metaXapply* shown above, can be found using *getTerm* as follows:

```
red getTerm(metaXapply(['AGING-2], 'init.Configuration,
  'age, none, 0, unbounded, 2))
```

and the result is shown in standard meta-level syntax:

```
'__['person[''JaneDoe.Qid,'s_^28['0.Zero]],
    'person[''JohnDoe.Qid,'s_^30['0.Zero]],
    'person[''JonathanDoe.Qid,'s_^5['0.Zero]]]
```

Strategies

A specification in rewriting logic may be both non-terminating and non-confluent, in addition to being non-deterministic. This clearly makes the question of how to execute such a specification on a deterministic machine non-trivial.

Maude provides two basic strategies, namely those implemented by the *rew* and *frew* commands. Apparently, and as we shall see later in this thesis, these are not satisfactory for a number of problems.

The point of defining a strategy is to complement or replace Maude's internal strategy. Using meta-level descent-functions such as *metaXapply*, a strategy for a rewriting logic specification can be defined *in rewriting logic*, because rewriting logic is reflective [9].

For example, a specification of an unreliable network would probably contain a rule for modeling that a message is lost during transmission due to a network failure of some kind. However, using one of Maude's internal strategies, we risk that this rule will be applied too often, perhaps every time it is applicable, meaning that every message will be lost(!). This is clearly not the desired behavior. The solution is to define our own strategy. Below we will look at a small yet illustrative example.

We assume that the rule for loosing a message is labeled *loose-msg*, and that we want this rule to fire at most once for every hundred rule applications. To achieve this, we define the following strategy:

```
fmod LOOSE-STRAT is

  protecting META-LEVEL .
  protecting INT .

  op loose-strat : Module Term Int -> Term .
  op loose-strat : Module Term Int Int Int -> Term .

  op remove-rule : Module RuleLabel -> Module .

  vars I J K : Int .
  var M : Module .
  var T : Term .
  var L : Qid .
  var MODNAME : QID .
  var IL : ImportList .
  var SS : SortSet .
  var SDS : SubSortDeclSet .
  var ODS : OpDeclSet .
  var MAS : MembAxSet .
```

```
  var ES : EquationSet .
  var RS : RuleSet .

  eq loose-strat(M, T, K) = loose-strat(M, T, 1, 0, K) .
  eq loose-strat(M, T, I, J, K) =
     if J > K then
       T
     else
       if I >= 100 then
         if metaXapply(M, T, 'loose-msg, none, 0,
            unbounded, 0) =/= failure then
              loose-strat(M, getTerm(metaXapply(
                M, T, 'loose-msg, none, 0,
                unbounded, 0)), 0, J + 1, K)
         else
           loose-strat(M, getTerm(metaRewrite(remove-rule(
             M, 'loose-msg), T, 1)), I + 1, J + 1, K)
         fi
       else
         loose-strat(M, getTerm(metaRewrite(remove-rule(
           M, 'loose-msg), T, 1)), I + 1, J + 1, K)
       fi
    fi .


  eq remove-rule(mod MODNAME is IL . SS SDS ODS
     MAS ES (L RS) endm, L) =
     (mod MODNAME is IL . SS SDS ODS
       MAS ES RS endm) .
endfm
```

The strategy defined above allows the user to specify a module M, a term T and an integer K, the latter corresponding to the number of rewrites to be performed.

Making use of the auxiliary function *remove-rule : Module* $\times$ *RuleLabel* $\rightarrow$ *Module*, this strategy rewrites the term T in module M with the *'loose-msg* rule removed, using Maude's standard rewrite strategy, until the variable I is equal to or greater than 100. When this is the case, the *'loose-msg* rule is tried, and if applied successfully, the counter I is reset to 0 and the rewrites will be performed according to Maude's internal strategy for at least 100 more iterations.

2.2 Creol

The *Creol* project is an ongoing research project at the University of Oslo, Norway. Creol is an acronym for Concurrent Reflective Object-oriented Language. The goal of the project is, quoting [12], to

> develop a formal framework and tool for reasoning about dynamic and reflective modifications in object-oriented open distributed systems, ensuring reliability and correctness of the overall system.

The syntax of the language is designed to appear familiar to programmers with little or no formal background, and is inspired by such languages and technologies as Simula [13], Java [18] and Corba [5]. It includes standard object-oriented mechanisms such as inheritance, interfaces, method calls etc.

The operational semantics of Creol is defined in rewriting logic, and based on this an interpreter has been defined in Maude, that can execute Creol specifications. In the following, we will look at some important aspects of the language that are central to our work. For a more detailed description, the reader is referred to the references that can be found on the Creol web page [12]. The information below is mainly based on [24, 26].

2.2.1 Classes and objects

From a programming point of view, attributes (object variables) and method declarations are organized in classes in a standard way. Objects are dynamically created instances of classes, and a new object is created with the *new* keyword. Creol supports multiple inheritance of both classes and interfaces. The attributes of an object are encapsulated and can only be accessed from the outside via the object's methods.

2.2.2 Interfaces

Creol objects are typed by interfaces (as opposed to being typed by classes, which is common in many other languages), and the methods an object offers to its environment may be specified through a number of interfaces. All interaction with an object happens through the methods of its interfaces. That is, any method that is not declared in an interface is considered internal to the object.

A method can be restricted to be callable only by objects implementing a certain interface using the *with <interface>* construct. If the interface following *with* is *any*, which is the super-interface of all interfaces, the method(s) in question can be called by any object.

An interface may also include both an invariant and an assumption. The assumption is a predicate on the communication history describing the expected behavior of an object's surrounding environment. The invariant is also

a predicate on the communication history, that describes (and enforces) the required behavior of any object that implements the interface. The invariant is guaranteed to always hold as long as the assumption on the environment holds.

2.2.3 Method calls

Creol supports both synchronous and asynchronous method calls, regardless of whether the calls are local or remote. The caller decides whether an invocation is synchronous or asynchronous. This provides a very flexible communication model.

In the synchronous setting, the caller is blocked until the call has completed at the callee, and the result is ready.

In the asynchronous setting, method calls can always be emitted, because the receiving object cannot block communication. The caller can proceed with its activity until the return value of the call is needed, and must then wait in the event that it has not yet arrived. Method overtaking is allowed in the sense that if methods offered by an object are invoked in one order, the object may react to the invocations in another order.

Since an object may have several pending calls, a unique *label* is used to identify each asynchronous call for which a reply is wanted. If the caller does not specify a label for an asynchronous call, it is not interested in the return value, even though the method may have out parameters.

Let m be a method name, *in* and *out* be (possibly empty) lists of in and out parameters, respectively, l a label and o an object reference. The different mechanisms for method calls can then be summarized as follows:

- m(in;out):
  local synchronous call
- l!m(in):
  local asynchronous call with label
- !m(in):
  local asynchronous call without label
- o.m(in;out):
  synchronous call to remote method
- l!o.m(in):
  asynchronous call to remote method with label
- !o.m(in):
  asynchronous call to remote method without label
- ?l(out):
  request for return values of an asynchronous call

Method calls are implemented by a message pair of an *invocation* message and a *completion* message. This makes for a very natural concept of asynchronicity.

Chapter 3

Communication Histories

In the object oriented programming paradigm, a common way to view an object is as a *black box*. This means that the programmer using the object in principle needs to know nothing of its implementation. As opposed to the so-called *glass box* view, the object's internal state is hidden and hence not directly available neither for inspection nor modification by its surrounding environment during execution, and the values of the object's attributes can only be changed through method calls. This is known as *encapsulation*, and allows an object to maintain full control of its internal variables and data structures.

Instead of resorting to the actual implementation, in the black box view an object can be described in terms of its *observable behavior*. The observable behavior of an object is, informally, its interaction patterns with other objects in the environment that can be observed by an external observer without any prior knowledge of the object's implementation.

In many object oriented languages that are popular in the computing industry, such as Java [18] and C♯ [20], observable interaction takes the form of method calls and/or events. A common way to specify which methods and events a class exposes is through an interface definition.

An interface definition for a Java or C♯ object is, however, quite limited, in the sense that it only specifies *which* operations are available (along with their respective type information), but nothing regarding the possible relationship between method invocations and ordering of them.

For example, consider a server object that provides services for reading from and writing to a shared resource, and that requires a connection to be opened in either read mode or write mode and the client to be authenticated before further requests can be made. In Java, a public interface for this server might look like the code in Figure 3.1 on the following page.

If we have access to the interface of the object, we can tell which methods the server offers, and may perhaps be able to make a qualified guess regarding the order in which they are supposed to be invoked, but apart from this the

```
public interface SimpleServer
{
    ConnectionHandle connect(int clientID, char mode);
    void close(ConnectionHandle conn);
    Data read(ConnectionHandle conn);
    void write(ConnectionHandle conn, Data data);
}
```

Figure 3.1: A simple server interface in Java

interface tells us little about how interaction with the object should proceed. For example, a common requirement is that when a connection is opened for writing, requests to open other connections must be either delayed or declined. Unfortunately, there is no way to express this in a Java interface (without resorting to comments), to know this we must have access to the actual implementation of the server.

Furthermore, the interface does not specify what *active* behavior an object may have. In other words, a Java/C♯ interface for a given object specifies what observable behavior other objects may perform on the object to which the interface belongs, and not the other way around.

Observable behavior can, however, be expressed in many ways that complement a standard interface definition consisting only of signatures (as shown in the example in Figure 3.1). One way is through a history [14] (or finite trace [21]) that is recorded as the object (and its environment) performs observable behavior. If all the manipulation of a given object happens through observable behavior, such a history can be considered an abstract representation of the object's state, and the object's behavior can be specified by a function on its history [27].

Using a history of observable behavior to specify the interactions with the server, we can for example express that in order to read, a client must first establish a connection, and then close it again after it has finished reading. This kind of specification is often referred to as a *safety specification*, stating that "nothing wrong will happen" [2]. An example is the following predicate:

$$P(H, x) = H\ /\ from(x)\ \mathrm{prs}\ (connect\ read^*\ close)^*$$

The predicate above uses a part of the global history (referred to as H), by utilizing a *projection* (/) that spans over only the messages originating from a given object x. Furthermore, the *prs* operator states that this projection of the history must be a prefix of the sequence specified by the regular pattern *(connect read* close)**. Hence, we have effectively specified that an object must open a connection before it can perform zero or more read operations, and that it must close the connection again before a new sequence of operations is started (for brevity, we have skipped the fact that an object

may also perform write operations etc).

The mechanisms used above will be discussed in further detail in Chapter 5, this little example is just meant as a motivation for what follows.

In the object-based Maude (and Creol) specifications that we will consider in this thesis, observable interaction takes place in the form of an exchange of messages between the objects. Hence, the history of observable behavior can be narrowed down to a history of messages; a *communication history.*

We shall now move on to consider some questions regarding important design choices for how communication histories can be integrated in the executable object environment provided by Maude.

3.1 Extending Maude specifications with histories

Messages and objects will be defined in a form resemblant of the style used in Full Maude [16]:

Object: $< O : C \mid att_1, ..., att_k >$

Message: $msg\ M\ from\ P\ to\ Q$

where O, P and Q are object identifiers, C is a class, att_1 through att_k are attributes and M is the actual contents of the message (possibly containing some additional data such as parameters).

Hence, the communication history H that will be built during execution, will be a list of such messages as defined above.

The implementation-level details concerning how to record the actual communication history will be delayed to the next chapter, where we will look closer at an implementation using Maude's meta-level. However, before we get that far, there are some important questions we need to address:

3.1.1 Where are the histories stored?

When it comes to storing the history during runtime, we have two choices:

1. each object can store its local history
2. a separate object can be used to store a global history

At first glance, it may seem that option number one is the better solution here, since it seemingly fits nicely into the object oriented programming mantra that each object should know of and be responsible for its own behavior and state. There are some problems with this approach, however:

First of all, if we want each object to store its history, each object needs some attribute for storing such a list of messages. This might not seem like a major problem, but as we shall see in section 3.1.3, we do not really want to modify the original specification, and since we cannot assume that the

objects in the original specification contain such attributes, we would have to add them.

Secondly, we can recreate each local history by using a projection on the global history (see Chapter 5 for more on this), but the same is not true of the opposite — it is not always possible to correctly recreate a global history from several local histories in a non-deterministic system, as the order in which the messages are to be interleaved is not known.

A final argument against encapsulating the communication history in the individual objects is that is does not really make sense to encapsulate information that to begin with is meant to be observable by an *external* observer.

Hence, we choose to create a global history of messages sent from the objects in a specification, and will provide the means to extract the local histories from the global history using projections.

3.1.2 When do we record message events in the history?

We consider an asynchronous communication model in which objects emit messages into a global state, and the messages may (or may not) eventually arrive at their destination. In this setting, there are basically three options when it comes to the time of recording of messages in the history. The logging can occur

1. when the message is sent
2. when the message is received
3. both of the above

In this thesis, we will use option number one; we will record messages in the history when they are sent. There are several reasons for this:

- Our communication history is a history of *observable* behavior, and for an external observer, it is clear that the action of an object sending a message can easily be observed. However, when it comes to the reception of a message, it is not obvious that this is indeed, from a conceptual point of view, observable by the environment at the exact time that it happens.
- We strive for simplicity rather than complexity. Using communication histories with messages recorded at the time of sending, we are, as we shall see later on, able to express quite sophisticated predicates in a quite easy and straightforward way. If we were to use option three, and record messages both at the time of sending and reception, the complexity would obviously increase significantly, arguably without adding much in terms of power of expression.

- By using option number two or three instead of one, the amount of possible states that can be reached from a given initial state will most certainly increase significantly because of the fact that a message can be received by an object at any point in time (due to the fact that an asynchronous network model may introduce an arbitrary delay between the time of sending and reception of a given message). A significant increase of reachable states makes searching (using Maude's *search* command) more demanding in terms of computational power, if not for all practical purposes impossible.

- In the Creol language, the point in code at which a message is emitted is easily located, however, no such point exists for the reception of a message, again due to the asynchronous communication model. Hence, when specifying communication properties for a given Creol program, the point at which a message is sent is a lot more usable than the point at which it is received.

Based on these arguments, we choose to record messages in the history at the time they are sent.

3.1.3 How do we build such a history during execution?

There are many ways in which we can make a Maude specification record a communication history of the messages that are sent between objects. Amongst our choices are:

1. the specification can be changed/reprogrammed so that each rewrite rule that sends a message also records that message in the history.

2. we can develop a compiler that will transform a standard specification into a specification that records messages in the history.

3. we can make use of Maude's meta-level to develop a strategy that records the messages in the history as it executes the specification.

What we will do is to use option number three, to make use of Maude's meta-programming capabilities. However, let us first look a little bit closer at the other two options.

Option number one, while clearly being a possible solution, is obviously the least desirable of the three, as it involves manual changes to every specification (an example of such a change is shown in figures 3.2 and 3.3 on the following page). As we know, manual change introduces many potential problems to a system:

- there is always the possibility that the programmer changing the specification makes an error that in some way breaks an already well-functioning application.

```
rl [send]:
  < 'S : Sender | Reciever : 'R >
=>
  < 'S : Sender | Reciever : 'R >
  msg 'M from 'S to 'R .
```

Figure 3.2: A rewrite rule that sends a message *'M* from *'S* to *'R*

```
rl [send]
  < 'S : Sender | Reciever : 'R >
  < 'H : History | messages : MLIST >
=>
  < 'S : Sender | Reciever : 'R >
  < 'H : History | messages : MLIST ++
     (msg 'M from 'S to 'R) >
  msg 'M from 'S to 'R .
```

Figure 3.3: A new version of the rewrite rule from Figure 3.2 that makes use of an object *'H* that maintains a global history of messages

- it is easy to forget to change certain parts of a specification, and such errors lead to an incomplete history, which in turn may lead to an incorrect result when checking a predicate/invariant.

- it is obviously time-consuming to make manual changes, and with a large code-base it would require a substantial effort from the development team, hence it is also expensive.

On the positive side, one might claim that manual changes may allow for a fine-grained approach to the problem that suites each case perfectly and that may outperform a generic solution. However, it seems clear that the negative sides to a manual change outweigh the positive.

So, let us move on to option number two; to develop a compiler that transforms a standard Maude specification into a Maude specification that builds a communication history while executing.

This option is more or less an automated version of option one (so the transformation of the code in Figure 3.2 to the code in Figure 3.3 would happen automatically in the compilation process), though without the possibility of a "tailor-made" solution for each specific problem. However, the manual work is also gone, which invalidates that part of the criticism of option one.

The biggest problem with this approach is probably the complexity involved in creating such a compiler, for several reasons:

- the syntax in Maude is entirely user-definable [7], which means that writing such a compiler would approach the complexity of writing a full parser for the Maude language.

- we want to be able to support a wide range of specifications, ranging from standard Maude specifications to Creol programs that are executed by the interpreter available from [12] (more on this later).

While creating a compiler is a viable solution, we have, as mentioned, chosen option three; to make a meta-level program that is able to execute an object-level specification, and recognize when an object in this specification sends a message, and record this message in its global history.

The main criticism of this approach is probably the performance overhead, which might be substantial, and we will look into this in closer detail in Chapter 4. However, since our main focus is on prototyping and testing, performance is perhaps not the most critical of aspects.

On the positive side, using a meta-level approach to this problem makes the solution completely *transparent*, in the sense that the original specification remains unchanged, and the meta-level may be plugged in only in the situations needed.

Furthermore, using a meta-level strategy opens up for several other possibilities in addition to just recording a communication history; as we shall see in later chapters the strategy will be extended to handle predicates on the history, as well as implement a pseudo-random rule selection scheme, and the latter would be very hard if not impossible to do using just a compiler from one object-level specification to another.

Chapter 4

A Meta-Level Rewrite Strategy for Recording the Communication History

In this chapter, we will introduce a meta-level Maude strategy for executing object-level Maude specifications. In addition to being able to execute a specification, the strategy should also be able to recognize, respond to and record an object's observable behavior.

The general idea is to keep track of the current state, and between each rewrite check the state to see whether any object initiated some observable behavior, as illustrated in Figure 4.1 on the next page.

Following the conclusions we made in Chapter 3, the strategy should be completely transparent with respect to the module that is executed, in the sense that no change should be needed in this module. Furthermore, the kind of observable behavior we are looking at is communication in the form of an exchange of messages.

The current state is represented as a configuration, which is a multi-set containing messages and objects. The configuration itself is represented as a *compound term* (see Section 2.1.3) at the meta-level. Our goal is to be able to recognize when an object sends a message to another object, and to record this message in a global communication history (the history will be used in Chapter 5, in which we will define predicates on a finite communication history).

In order to achieve this goal, we need some mechanism for keeping track of the current state, the communication trace, and a concrete strategy that is able to decide which rewrite rule to apply next. This mechanism will be developed in a Maude module called *META-ENGINE*, utilizing two objects, an *Engine* object and a *History* object.

The rationale behind using two separate objects for this is that they provide for quite different functionality, as we shall see in the following sec-

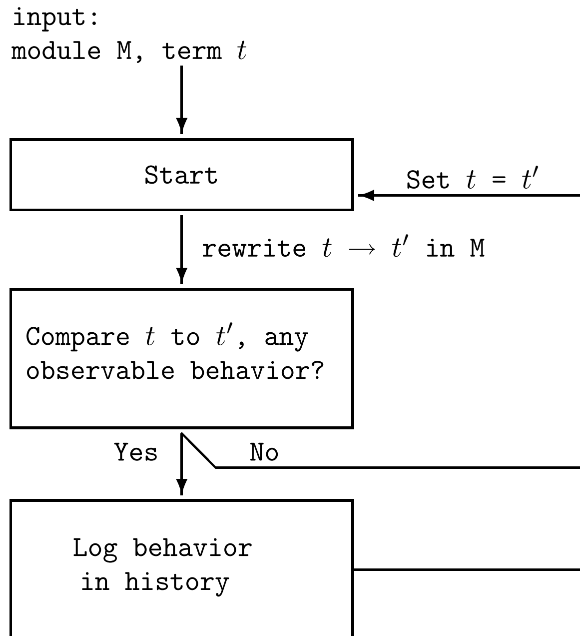

Figure 4.1: Outline of the rewrite strategy

tions. Furthermore, we want to be able to "plug" in and out the *History* object without changing the *Engine* object, should the need arise.

We start out by introducing a sort *EngineObject*, of which both our objects will be. *EngineObject* is a subsort of *EngineConfig*, which is a multiset of engine objects. Such an object will keep track of data that is needed for our strategy (or that we want to store for other reasons) between consecutive rewrites.

In this thesis, we will only look at situations where there is exactly one *Engine* object, and at most one *History* object. However, extending this to allow for more than one engine could perhaps be an interesting study.

4.1 The *Engine* object

To keep track of the current state, we will make use of an object called *Engine* of sort *EngineObject*. This object keeps track of data that is needed by our strategy in order to perform the rewrites. In a sense this object corresponds to the data maintained by Maude's default internal strategy. It is defined as follows:

```
op Engine[curTerm:_, curModule:_, labels:_,
  failedRules:_, numRules:_] :
```

```
Term Qid QidList QidList Int -> EngineObject [ctor] .
```

As we see, the *Engine* object contains several attributes:

- The *curTerm* attribute contains the entire state of the system (to which we are applying our rewrite strategy), which typically would be a multiset of objects and messages of sort *Configuration*, meta-represented as a term of sort *Term*.
- The name of the current module (in which the equations and rewrite rules that define the operations and transitions allowed on the term stored in *curTerm* are contained) is stored as a quoted identifier (*Qid*) in the attribute *curModule*.
- A module in rewriting logic has a (possibly empty) set of rewrite rules. In Maude, each of the rules has a label that is used as an identifier. These labels are meta-represented as a term of sort *Qid*, and as such they can easily be gathered in a list of sort *QidList* and stored in the *labels* attribute.
- Whenever the application of a given rule is attempted on a given meta-representation of a configuration, the outcome may be one of two: either the rule can be applied (it is *enabled* in the configuration) , and the term resulting from the application is returned. Otherwise, if the rule is not enabled, the application will fail, and the *descent* function will return *failure* (see Section 2.1.3 for more information on descent functions). The *failedRules* attribute keeps track of the rules that have failed up till now for the current term.
- The *numRules* attribute contains the number of rules in the module whose name is contained in *curModule*, for efficiency reasons (we could easily have calculated this each time we needed it from the *labels* list).

We now introduce a first strategy to illustrate meta-level programming (we will look at different ways to extend and improve this strategy in the following chapters). The rules will be applied in a round robin fashion, using the *labels* list attribute. We try to apply the rule that is at the head of this list to the term in the *curTerm* attribute. If the rule application succeeds, the rule is moved to the back of the list, *curTerm* is replaced with the resulting term from the application, the list of failed rules is reset to the empty list (*nil*) and the next rule is tried. If, on the other hand, the rule application fails, the rule is moved to the list of failed rules, and the current term remains unchanged. This is handled by the conditional rewrite rule *exec*, as shown in Figure 4.2 on the following page.

Since the rule in Figure 4.2 is conditional, the execution will terminate when the length of the list of failed rules is equal to the number of rules in the module, since no rule is applicable.

```
crl [exec]:
  Engine[curTerm: T, curModule: MOD, labels: LABEL LABELS,
    failedRules: FAILEDRULES, numRules: NUMRULES]
=>
  if metaXapply([MOD], T, LABEL, none, 0, unbounded, 0) =/=
    failure
  then
    Engine[curTerm: getTerm(metaXapply([MOD], T, LABEL, none,
      0, unbounded, 0)), curModule: MOD, labels: LABELS LABEL,
      failedRules: nil, numRules: NUMRULES]
  else
    Engine[curTerm: T,
      curModule: MOD, labels: LABELS LABEL, failedRules:
      FAILEDRULES LABEL, numRules: NUMRULES]
  fi
if length(FAILEDRULES) < NUMRULES .
```

Figure 4.2: A rewrite strategy that applies rewrite rules in a round robin fashion

As mentioned in Section 2.1, Maude provides two built-in basic rewriting strategies, implemented by the *rew* and *frew* commands. The *rew* command will perform rewrites using a leftmost and outermost strategy for applying rules and reduce the whole term by equations after each successful rule rewrite [11]. This means that on subsequent rewrites, the same rule(s) will always be tried first, possibly yielding a very skewed result.

The *frew* command, on the other hand, tries to be position fair by making a number of depth-first traversals of the term. Each position is rewritten only once per traversal. Furthermore, only the subterm that was rewritten is reduced using the equations in the module, allowing other subterms to rewritten further down the traversal.

The strategy presented in Figure 4.2 lies somewhere in-between those implemented by *rew* and *frew*. Since it tries rules in a round robin fashion, it will in many cases avoid the typically skewed results that *rew* produces. Our strategy is fair when it comes to the rule applications. On the other hand, as opposed to *frew* it is not position fair, the first position applicable for a given rule will always be tried first.

In Chapter 9 we will look at how we can make the rewrite strategy fairer by using a non-deterministic approach based on random numbers.

4.2 The *History* object

In order to build a communication history, we need to know when a new message is sent from one object to another (meaning that it is put into the configuration via some rule application), and we need a means of storing the communication history between rewrite steps.

To store the history, we introduce a new object, *History* of sort *EngineObject*:

```
op History[h:_] : MsgList -> EngineObject [ctor] .
```

This object has only one attribute: *h*. This attribute will contain the actual communication trace during the execution of a given specification, represented by a message list of sort *MsgList*, which is defined as follows:

```
fmod MSG-LIST is
    sort MsgList .
    subsort Msg < MsgList .

    op nil : -> MsgList [ctor] .
    op _@_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .
endfm
```

We note that the concatenation operator for message lists is the @ operator.

To build the communication trace, we need to check the current term for new messages between each successful rule application, hence we need to make some changes to the *exec* rewrite rule:

- The *History* object needs to be included in the left hand side of the rule, this is a trivial change.
- If the rule application fails, the history object remains unchanged.
- If the rule application succeeds, the history objects needs to be updated with any new messages created by the last rule application, if any.

Clearly, the main issue is the last bullet, how to update the history with new messages, and to achieve this there are two specific problems that need to be addressed:

1. How do we recognize a message in a meta-level representation of an entire configuration?
2. How do we separate the messages that were created by the current rule application (which are the ones we want to add to the history) from those created by previous rule applications that may still be present in the configuration?

Addressing problem number one first, we know that a configuration by definition is a multiset of messages and objects. In Maude, it is conventional to construct such multisets using the "invisible" constructor operator `__`, which is defined as being both associative and commutative with *none* as the identity element. Knowing this, we also know that the meta-representation of a given configuration would be a compound term, as described in Section 2.1.3. For instance, a meta-representation of a configuration containing two simple objects with no attributes, *'A* and *'B*, and one message from *'A* to *'B* could look something like this:

```
'__['<_:´ Object>[''A.Qid], '<_:´ Object>[''B.Qid],
  'msg_from_to_['M.Msg, ''A.Qid, ''B.Qid]]
```

To find out which of the subterms within the compound term are messages, we need to iterate through them and check them one by one. This functionality could be implemented in several ways, we choose to make use of Maude's built-in function $wellFormed : Module \times Substitution \rightarrow Bool$, that returns *true* if a given substitution is valid within a given module, and false otherwise. To simplify things a bit, we define an auxiliary function *isMetaMessage* that takes a term and the name of a module, and returns *true* if the term is a valid meta-level representation of a message:

```
op isMetaMessage : Term Qid -> Bool .
eq isMetaMessage(T1, MOD) = wellFormed([MOD], 'M:Msg <- T1) .
```

Using this function we can check each subterm of the configuration recursively to decide whether it is a message or not.

Having found a way to decide which parts of a given meta-representation of a configuration are actually messages, we can move on to our second problem from above — how do we tell new messages from old ones?

Given that a configuration might contain several identical messages, and that each rewrite rule might produce or consume an arbitrary number of messages, we need to use a counting scheme to figure out whether a meta-level message in a given meta-level configuration is new or not. For this purpose, we introduce a function *countGroundTerms*, which counts the number of ground terms in a ground term list that are equal to a given ground term:

```
op countGroundTerms : GroundTerm GroundTermList -> Int .
op countGroundTerms : GroundTerm GroundTermList Int -> Int .
vars GT1 GT2 : GroundTerm .
var GTL2 : GroundTermList .
var I : Int .

eq countGroundTerms(GT1, GTL2) =
   countGroundTerms(GT1, GTL2, 0) .
```

```
eq countGroundTerms(GT1, GT2, I) =
   if GT1 == GT2 then 1 else 0 fi .
eq countGroundTerms(GT1, (GT2, GTL2), I) =
   if GT1 == GT2 then
      1 + countGroundTerms(GT1, GTL2, I + 1)
   else
      if I > 0 then
         0
      else
         countGroundTerms(GT1, GTL2, 0)
      fi
   fi .
```

Since we know that Maude sorts the subterms in the resulting term between each rewrite, we can stop counting when the two terms differ and we have already counted to one or more, as shown in the code above.

Closing in on the solution to this problem, we are now ready to define a function *getNewMessages*, that returns a list of messages that are new in one meta-level configuration compared to another. The method that we will use is, as described above, to iterate through all the terms in the new configuration, and for each term check whether it is a message or not, and whether it is new in this configuration or not, using the auxiliary functions defined above.

The definition of *getNewMessages* is as follows:[1]

```
op getNewMessages : Term Term Qid -> MsgList .

var GT2 : GroundTerm . vars GTL1 GTL2 : GroundTermList .
var MOD : Qid .

eq getNewMessages('__[GTL1], '__[GT2, GTL2], MOD) =
   if isMetaMessage(GT2, MOD) and countGroundTerms(GT2, GTL1) <
      countGroundTerms(GT2, (GT2, GTL2))
   then
      groundTermToMessage(GT2) @
        getNewMessages('__[GTL1], '__[GTL2], MOD)
   else
      getNewMessages('__[GTL1], '__[GTL2], MOD)
   fi .
```

[1]This code shows the general idea, but actual implementation is a bit more involved, seeing as we have to take into consideration that either of the two configurations may be empty or consist of only one term etc. The interested reader may consult the source code in Appendix A for the full details.

As shown above, *getNewMessages* makes use of yet another auxiliary function; *groundTermToMessage : GroundTerm → Message*. This function returns an object-level message from its meta-level representation, since messages will be stored in their object-level form in the communication history.

The code for *getNewMessages* and the auxiliary *countGroundTems* functions may seem inefficient. Since Maude sorts the subterms within a term, we should be able to perform the check for new messages with one traversal through both ground term lists at the same time, as shown in the non-executable code below:

```
eq getNewMessages('__[GT1, GTL1], '__[GT2, GTL2], MOD) =
   if GT1 == GT2
      getNewMessages('__[GTL1], '__[GTL2], MOD)
   else
      if GT1 MaudeSort< GT2 then
        getNewMessages('__[GTL1], '__[GT2, GTL2], MOD)
      else
        groundTermToMessage(GT2) @
          getNewMessages('__[GT1, GTL1], '__[GTL2], MOD)
      fi
   fi .
```

The problem with this code, is that we do not have access to Maude's internal sorting algorithm, and hence we cannot define the *MaudeSort<* relational operator that returns *true* if its left argument is smaller than its right according to this algorithm.[2]

Another possible objection to our strategy can be illustrated by the following example: suppose we have a configuration in which there is an object A, and a message M in transit from A to A:

$$< \ A \mid ... \ > \ \textit{msg } M \textit{ from } A \textit{ to } A$$

If, in the next rewrite, A consumes the message and, in the same rewrite, sends out an identical message, our strategy would not be able to capture the fact that a new message is sent, since the states before and after the rewrite are identical.

However, we focus on *observable* behavior, and since there is no observable change in the state before and after the rewrite, it can be argued that the two states are essentially the same. Hence, any such rewrite steps are not

[2]When looking at the results produced by Maude, the sort order appears to be alphabetical. However, there is no easy way in Maude today to alphabetically compare two terms. Writing an algorithm that turns two terms into strings and compares them with the built-in $<$ operator might be worth considering, but we will not pursue this idea any further in this thesis.

visible to an external observer, and sending of messages that do not result in a state change should not be included in the history.

As we shall see in Chapter 8, none of this is a problem when dealing with Creol messages, as every such message has a unique identifier attached to it.

We are now ready to define our entire rewrite strategy; in Figure 4.3 we have the *exec-history* conditional rewrite rule, which is a modified version of the *exec* rule presented in Figure 4.2 on page 36. This strategy builds a communication trace as the execution proceeds.

```
crl [exec-history]:
  Engine[curTerm: T, curModule: MOD, labels: LABEL LABELS,
    failedRules: FAILEDRULES, numRules: NUMRULES]
  History[h: ML]
=>
  if metaXapply([MOD], T, LABEL, none, 0, unbounded, 0) =/=
    failure
  then
    Engine[curTerm: getTerm(metaXapply([MOD], T, LABEL, none,
      0, unbounded, 0)), curModule: MOD, labels: LABELS LABEL,
      failedRules: nil, numRules: NUMRULES]

      History[h: ML @ getNewMessages(T, getTerm(metaXapply(
        [MOD], T, LABEL, none, 0, unbounded, 0)), MOD, ML)]

  else
    Engine[curTerm: T,
      curModule: MOD, labels: LABELS LABEL, failedRules:
      FAILEDRULES LABEL, numRules: NUMRULES]
    History[h: ML]
  fi
if length(FAILEDRULES) < NUMRULES .
```

Figure 4.3: A rewrite strategy which records a communication history as the execution proceeds

The first thing that happens in the *exec-history* rule, is as before that a rule application is tried. If it succeeds, the *Engine* object is updated so that the current term is now the resulting term from the rule application performed by *metaXapply*, and the rule label is placed at the back of the labels list. Furthermore, the list of failed rules is reset to *nil*, seeing as every rule may be applicable after a successful rewrite.

The *History* object needs to be updated as well, and this is done by concatenating the message list in h (using the @ concatenation operator)

with the new messages found by *getNewMessages*. If there are several new messages, the order in which they are added to the history is arbitrary; we use the order provided by Maude's internal sorting algorithm.

If, on the other hand, the rule application fails, the *History* object remains unchanged, while the rule label of the failed rule is placed in the *failedRules* list of the *Engine* object.

In order to make the strategy easier to use, we define a function *start* : $Qid \times Term \rightarrow EngineConfig$ that takes the name of a module and a term, and returns an *Engine* object and a *History* object with the correct values in their respective properties:

```
eq start(MOD, T) =
   Engine[curTerm: T, curModule: MOD, labels:
     getRuleLabels(MOD), failedRules: nil,
     numRules: length(getRuleLabels(MOD))]
   History[h: nil] .
```

The *getRuleLabels* auxiliary function returns, as its name implies, a list of rule labels for a given module.

4.3 Performance

When we introduce meta-level computation, there is obviously some overhead involved compared to standard object-level computation. First, there is the fact that in order to perform meta-level rewrites, Maude lowers the term that is to be rewritten all the way down to the object level before any rewrites are performed, and then raises the resulting term to the appropriate meta-level representation again when the computation is completed. Maude does this in order to increase the performance of meta-level computing.

In addition to the actual lowering and raising of terms, there is also overhead involved with our strategy in general; rewrites must be performed to decide which meta-level rewrite rules that are to be applied, etc.

Furthermore, perhaps the heaviest operation (in terms of computational power required) is to check whether any new messages have been sent, and if so to add them to the history.

To test the performance overhead, we will establish a test suite of small Maude specifications, and perform some simple tests:

- The test in Figure 4.4 on the facing page has only one object, a counter, that will add one to the counter value per rewrite, until the counter has reached a value of one hundred thousand. We will use this test to measure the performance overhead for "raw" term rewriting - there are no messages being sent, and there is only one rule which will always be applicable until the execution terminates.

- When a specification has a high number of rules that are not applicable at any given point, it is natural to suspect that our rewrite strategy will perform worse than in situations where most of the rules are applicable, since we actually have to try a given rule to see if it fails or not, and this will clearly be expensive with a lot of failures. The code in Figure 4.5 on page 44 is a variation of the one in 4.4, only this time there are ten rules that increase the counter, but only one is applicable at any given time.

- For the final test in this suite we will consider a specification in which there will be a number of objects that send messages to each other. The sending is organized in such a way that the messages will be sent in a ring between the objects. The main purpose of this test is to see how the history enabled strategy performs when there are many messages in transit. Figure 4.6 on page 45 shows the code for the specification.

```
mod COUNTER-TEST is

    protecting OBJ .
    protecting QID .
    protecting INT .

    op <_: Counter | i:_ > : Qid Int -> Obj .

    op init : -> Configuration .
    eq init = < 'C : Counter | i: 0 > .

    var I : Int .

    crl [add-one] :
      < 'C : Counter | i: I >
    =>
      < 'C : Counter | i: I + 1 >
    if I < 100000 .
endm
```

Figure 4.4: Counter performance test

4.3.1 Test results

Below we will look at some results from running the tests suite defined above. When executing a specification, Maude returns three numbers of interest:

```
mod NON-APPLICABLE-TEST is

    protecting OBJ .
    protecting QID .
    protecting INT .

    op <_: Counter | i:_ > : Qid Int -> Obj .

    op init : -> Configuration .
    eq init = < 'C : Counter | i: 0 > .
    var I : Int .

    crl [add-one-0] :
      < 'C : Counter | i: I > =>
      < 'C : Counter | i: I + 1 >
    if I rem 10 == 3 and I < 100000 .

    crl [add-one-1] :
      < 'C : Counter | i: I > =>
      < 'C : Counter | i: I + 1 >
    if I rem 10 == 6 and I < 100000 .

    crl [add-one-2] :
      < 'C : Counter | i: I > =>
      < 'C : Counter | i: I + 1 >
    if I rem 10 == 2 and I < 100000 .

    *** rules 3 - 8 are similar

    crl [add-one-9] :
      < 'C : Counter | i: I > =>
      < 'C : Counter | i: I + 1 >
    if I rem 10 == 7 and I < 100000 .
endm
```

Figure 4.5: Counter performance test with only one applicable rule out of ten

```
mod MESSAGE-TEST is

    protecting OBJ .
    protecting QID .
    protecting INT .

    op <_: SenderAndReciever | sendTo: _ > : Qid Qid -> Obj .
    op <_: Counter | i:_ > : Qid Int -> Obj .

    op init : -> Configuration .
    eq init =
      < 'A : SenderAndReciever | sendTo: 'B >
      < 'B : SenderAndReciever | sendTo: 'C >
      < 'C : SenderAndReciever | sendTo: 'D >
      < 'D : SenderAndReciever | sendTo: 'E >
      < 'E : SenderAndReciever | sendTo: 'A >
      < 'Counter : Counter | i: 0 > .

    vars O1 O2 O3 C : Qid . var I : Int .

    crl [send] :
      < O1 : SenderAndReciever | sendTo: O2 >
      < 'Counter : Counter | i: I >
    =>
      < O1 : SenderAndReciever | sendTo: O2 >
      (msg 'Test from O1 to O2)
      < 'Counter : Counter | i: I + 1 >
    if I < 100000 .

    crl [recv] :
      < O1 : SenderAndReciever | sendTo: O2 >
      (msg 'Test from O3 to O1)
      < 'Counter : Counter | i: I >
    =>
      < O1 : SenderAndReciever | sendTo: O2 >
      < 'Counter : Counter | i: I + 1 >
    if I < 100000 .
endm
```

Figure 4.6: Performance test specification containing a configuration of objects that send messages in a ring

- the number of rewrites that were performed,
- the number of milliseconds of CPU-time that the execution took,
- and finally, the number of milliseconds of real elapsed time the execution took.

Of these three numbers, the first one, the number of rewrites performed, is probably the most interesting, since CPU speed varies from machine to machine, and depends on many factors beyond our control, hence it is not easy to get accurate results, especially when it comes to actual time elapsed.

However, the number of milliseconds of CPU-time may in some cases be interesting in order to see how much work Maude does internally compared to the number of rewrites performed.

Counter test We test the specification presented in Figure 4.4 on page 43:

1. Maude's *rew* command:

```
Maude> rew in COUNTER-TEST : init .
rewrite in COUNTER-TEST : init .
rewrites: 300002 in 1230ms cpu (1330ms real)
  (243904 rewrites/second)
result Obj: < 'C : Counter | i: 100000 >
```

2. Our strategy without the history object:

```
Maude> rew in META-ENGINE : start('COUNTER-TEST,
  'init.Configuration) .
rewrite in META-ENGINE : start('COUNTER-TEST,
  'init.Configuration) .
rewrites: 1100022 in 9820ms cpu (10630ms real)
  (112018 rewrites/second)
result EngineObject: Engine[curTerm: '<_:`Counter`|`i:_>[
    ''C.Qid,'s_^100000['0.Zero]],curModule: 'COUNTER-TEST,
    labels: 'add-one,failedRules: 'add-one, numRules: 1]
```

3. Our strategy with the history object:

```
Maude> rew in META-ENGINE : start('COUNTER-TEST,
  'init.Configuration) .
rewrite in META-ENGINE : start('COUNTER-TEST,
  'init.Configuration) .
rewrites: 2200022 in 16290ms cpu (17760ms real)
  (135053 rewrites/second)
result EngineConfig: History[h: nil] Engine[curTerm:
    '<_:`Counter`|`i:_>[''C.Qid,'s_^100000['0.Zero]],
    curModule: 'COUNTER-TEST,labels: 'add-one,
    failedRules: 'add-one,numRules: 1]
```

From these tests, we see that when using our strategy without the history, Maude must perform approximately 3.66 times more rewrites than when using the plain built-in *rew* command. This is not really surprising, seeing as for each rewrite that is to be performed at the object level, *at least* one rewrite must be performed at the meta-level only to execute the *metaXapply* function.

When plugging in the history object as well, the strategy requires approximately 7.33 times the number of rewrites that Maude's internal command does, or about twice as much as our strategy without the history. However, it is worth noting at this point that there is only one object in this test, and it does not send any messages.

Counter test with only one applicable rule out of ten We test the specification presented in Figure 4.5 on page 44:

1. Maude's *rew* command:

```
Maude> rew init .
rewrite in NON-APPLICABLE-TEST : init .
rewrites: 3199817 in 4280ms cpu (4280ms real)
  (747620 rewrites/second)
result Obj: < 'C : Counter | i: 100000 >
```

2. Our strategy without the history object:

```
Maude> rew in META-ENGINE : start('NON-APPLICABLE-TEST,
  'init.Configuration) .
rewrite in META-ENGINE : start('NON-APPLICABLE-TEST,
  'init.Configuration) .
rewrites: 9839531 in 26400ms cpu (26480ms real)
  (372709 rewrites/second)
result EngineObject: Engine[curTerm: '<_:`Counter`|`i:_>[
    ''C.Qid,'s_^100000['0.Zero]],curModule:
    'NON-APPLICABLE-TEST,labels: 'add-one-2 'add-one-3
    'add-one-4 'add-one-5 'add-one-6 'add-one-7
    'add-one-8 'add-one-9 'add-one-0 'add-one-1,
    failedRules: 'add-one-2 'add-one-3 'add-one-4
    'add-one-5 'add-one-6 'add-one-7 'add-one-8
    'add-one-9 'add-one-0 'add-one-1,numRules: 10]
```

3. Our strategy with the history object:

```
Maude> rew in META-ENGINE : start('NON-APPLICABLE-TEST,
  'init.Configuration) .
rewrite in META-ENGINE : start('NON-APPLICABLE-TEST,
  'init.Configuration) .
```

```
rewrites: 10939443 in 33210ms cpu (33220ms real)
  (329402 rewrites/second)
result EngineConfig: History[h: nil] Engine[curTerm:
    '<_:`Counter`|`i:_>[''C.Qid,'s_^100000['0.Zero]],
    curModule: 'NON-APPLICABLE-TEST,labels:'add-one-2
    'add-one-3 'add-one-4 'add-one-5 'add-one-6
    'add-one-7 'add-one-8 'add-one-9 'add-one-0
    'add-one-1,failedRules: 'add-one-2 'add-one-3
    'add-one-4 'add-one-5 'add-one-6 'add-one-7
    'add-one-8 'add-one-9 'add-one-0 'add-one-1,
    numRules: 10]
```

Looking at the results from this test, they are actually quite surprising. The suspicion that our strategy would be very inefficient compared to Maude when a number of the rewrite rules are not applicable seems to be wrong, judging from these results. A mere 3.42 times more rewrites than Maude's internal engine performed is what resulted from the test run without the history object. Note that this is actually *better* than in the previous test. When considering CPU usage, however, we see that our solution uses about six times the CPU time that Maude does.

When the history object is plugged in, we see that Maude is approximately 3.42 times more efficient than our strategy, which is about the same as for the case without the history, so in this situation, the checking for new messages is clearly not a dominating factor.

Test with objects sending messages to each other in a ring We test the specification presented in Figure 4.6 on page 45.

1. Maude's *rew* command:

```
Maude> rew in MESSAGE-TEST : init .
rewrite in MESSAGE-TEST : init .
rewrites: 300006 in 1730ms cpu (1740ms real)
  (173413 rewrites/second)
result Configuration:
  < 'A : SenderAndReciever | sendTo: 'B >
  < 'B : SenderAndReciever | sendTo: 'C >
  < 'C : SenderAndReciever | sendTo: 'D >
  < 'D : SenderAndReciever | sendTo: 'E >
  < 'E : SenderAndReciever | sendTo: 'A >
  < 'Counter : Counter | i: 100000 >
```

2. Our strategy without the history object:

```
Maude> rew in META-ENGINE : start('MESSAGE-TEST,
  'init.Configuration) .
```

```
rewrite in META-ENGINE : start('MESSAGE-TEST,
  'init.Configuration) .
rewrites: 1100040 in 15850ms cpu (16080ms real)
  (69403 rewrites/second)
result EngineObject: Engine[curTerm: '__[
    '<_:`SenderAndReciever`|`sendTo:_>[''A.Qid,''B.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''B.Qid,''C.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''C.Qid,''D.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''D.Qid,''E.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''E.Qid,''A.Qid],
    '<_:`Counter`|`i:_>[''Counter.Qid,'s_^100000['0.Zero]]],
    curModule: 'MESSAGE-TEST,labels: 'send 'recv,
    failedRules: 'send 'recv,numRules: 2]
```

3. Our strategy with the history object:

```
Maude> rew in META-ENGINE : start('MESSAGE-TEST,
  'init.Configuration) .
rewrite in META-ENGINE : start('MESSAGE-TEST,
  'init.Configuration) .
rewrites: 28199907 in 67760ms cpu (68310ms real)
  (416173 rewrites/second)
result EngineConfig: History[h: (msg 'Test from 'A to 'B) @
    (msg 'Test from 'A to 'B) @ (msg 'Test from 'A to 'B) @
    (msg 'Test from 'A to 'B) @ (msg 'Test from 'A to 'B) @
    (msg 'Test from 'A to 'B) @ (msg 'Test from 'A to 'B) @
      . . .
    (msg 'Test from 'A to 'B)] Engine[curTerm: '__[
    '<_:`SenderAndReciever`|`sendTo:_>[''A.Qid,''B.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''B.Qid,''C.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''C.Qid,''D.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''D.Qid,''E.Qid],
    '<_:`SenderAndReciever`|`sendTo:_>[''E.Qid,''A.Qid],
    '<_:`Counter`|`i:_>[''Counter.Qid,'s_^100000['0.Zero]]],
    curModule: 'MESSAGE-TEST,labels: 'send 'recv,
    failedRules: 'send 'recv,numRules: 2]
```

Our strategy without the history compared to Maude's *rew* command is slower in this test than it has been before, the number of rewrites are approximately 6.34 times higher. This indicates that a lot of objects and messages leads to more meta-level computation, and hence will have a slight negative impact on the performance of our strategy.

Moving on to the strategy with the history, the first thing to note is that the result shown in the listing number three above is severely shortened compared to the actual output from Maude, due to the fact that the history after this execution contained in the order of several tens of thousands of messages. This is also something to keep in mind when we consider the performance results.

The strategy with the history recording needs approximately 25 times as many rewrites as the strategy without the history, however, it does actually do quite a lot more as well.

Chapter 5

Predicates on Finite Communication Histories

In Chapter 4, we looked at a meta-level rewrite strategy that enabled us to build a communication history when executing a Maude specification. We did not, however, make any use of this history; that is the scope of this chapter. We will develop mechanisms to specify predicates, and mechanisms to check whether a given communication history is in concordance with a given predicate in-between rewrites. These mechanisms will be used actively by our rewrite strategy to select which rules to apply to the current configuration, in order to ensure that a given specification is executed according to a given predicate.

A predicate is, in general, a statement that evaluates to *true* or *false* for some input of the correct type. We define a *history predicate* to be a statement that evaluates to either *true* or *false* for any given finite sequence of messages of sort *Msg*. The communication histories recorded by our strategy from Chapter 4 are such sequences.

The predicates that we will define in Maude in this Chapter are based on the concepts introduced in [25, 27].

In order to express predicates on the communication history, it would be convenient if we could use basic list functions, like for example *length(H)* and *isEmpty(H)*, where H is a communication history. Furthermore, we would like to use standard boolean operators like *and* and *or* etc.

At first glance, it would be tempting to define these aforementioned functions directly in Maude, for a message list H and a single message M, in the standard way, e.g.

```
op length : MsgList -> Int .
eq length(nil) = 0 .
eq length(M @ H) = 1 + length(H) .
```

and

```
op isEmpty : MsgList -> Bool .
eq isEmpty(nil) = true .
eq isEmpty(H) = false [otherwise] .
```

and then use them in an appropriate expression as a parameter to our meta-rewriting engine.

However, if one were to try this, the problems with this approach would soon become apparent. Assume that we try to modify the *start* function defined at the end of Section 4.2, so that we could specify a predicate in addition to a term and a module. We could then, for example, try to put restrictions on the length of the history in the following way, for a given natural number N:

```
start(Term, Module, length(H) <= N) .
```

What would happen here? First of all, H needs to be a message list that is available at the time of the call to *start* (otherwise, Maude will not know what to do with it). Second, assuming H is available, the predicate would immediately be reduced to either *true* or *false*, depending on the contents of *H at the time of the call*, because Maude performs all rewriting *modulo equations*, as described in Section 2.1.

Since we want to check the predicate in-between each rewrite during runtime, it becomes clear that we need some way of constructing the predicates such that

1. we do not need to have access to the history H at the time of the call, and

2. the predicates are not immediately reduced by Maude's rewrite engine.

5.1 Data structures for predicate construction

In order to make the predicate specification as easy as possible for the user, it is important that we maintain a "natural" syntax. At the same time, we need something that can be easily parsed, as we will have to parse it ourselves (at least partly), in order to avoid instant reduction by Maude to either *true* or *false*.

We start out by introducing a new module in Maude, called *PRED*. In this module we define a sort *Pred*, of which our predicate specifications will be.

The first requirement that needs to be addressed, is that we need some way of referring to the communication history without actually having access to it at the time of the call. For this we introduce the sort *History*, and the constant H that will be used as a *placeholder* for the actual communication history:

```
sort History .
op H : -> History [ctor] .
```

We may now refer to *H* in place of the real history, and are now able to define constructors for e.g. the length of the history in the following way:

```
op length : History -> Pred [ctor] .      (1)
```

Notice how this definition differs from the one that we introduced at the start of this chapter, shown again below:

```
op length : MsgList -> Int .              (2)
```

What we are doing here in our topmost definition (1), is that we are creating a length operator that, contrary to our previous definition (2), does not actually return the length of a history, but instead constructs an expression of sort *Pred*, that is not further reducible in its current form. We will, in other words, have to parse expressions of this kind later on, and hence we can control when and how the reductions are performed.

To structure our functions in a clearer manner, we introduce two new sorts, *BoolExp* and *IntExp*, and let them denote expressions (or parts of predicates) that when parsed and further reduced are supposed to return integer and boolean values, respectively. We make both these sorts subsorts of the general sort *Pred*, and refine our *length* signature (1) to be of sort *IntExp* instead of just *Pred*, since the length of a history is assumed to be an integer.

Now we are able to define other desirable predicate operators quite easily, e.g. the boolean *and* and *or* operators:

```
op _and_ : BoolExp BoolExp -> BoolExp [ctor] .
op _or_  : BoolExp BoolExp -> BoolExp [ctor] .
```

and binary relation operators *equal*, *less than or equal* and *greater than or equal* for integer expressions:[1]

```
op _eq_ : IntExp IntExp -> BoolExp [ctor] .
op _lte_ : IntExp IntExp -> BoolExp [ctor] .
op _gte_ : IntExp IntExp -> BoolExp [ctor] .
```

Furthermore, we state that an integer is also an *IntExp* by using Maude's subsort feature:

```
subsort Int < IntPred .
```

[1] Due to conflicts with predefined modules in Maude, we are unable to use $=$ for equality, $<=$ for less than or equal, etc.

5.2 Projections on the history

To make our predicates more fine-grained, it would be nice to have the ability to define predicate expressions ranging over only a certain part of the history, as opposed to the entire history at once. To achieve this, we define *projections*.

For communication histories H and H', a projection is a function P such that $P(H) = H'$. For every message m_i in H', m_i must also be in H. Furthermore, for every pair of messages m_j and m_k in H', if m_j comes before m_k in H, m_j must also come before m_k in H'. A projection P on a history H is conventionally denoted H/P.

To define projections in Maude, we start out by introducing a new sort, *Projection*, and a projection constructor operator $/$:

```
sort Projection .
op _/_ : History Projection -> History [ctor] .
```

Since the objects that we are considering (both in Maude and Creol) have *explicit and unique identities*, and these identities are contained within the messages they send, we can define projections that span messages originating from or destined for a single object:

```
op from : Qid -> Projection [ctor] .
op to : Qid -> Projection [ctor] .
```

As in the previous section, these constructor operators have no real functionality yet, but they enable us to *express* predicates on the history (we will get to the implementation of the actual functionality shortly).

The projections enable us to specify properties concerning only one object, as shown below:

```
H / from('myObject) .
```

Since the projection operator $/$ is of sort *History*, projections can be combined with other operators that we have already introduced, e.g. in an imaginary scenario with only two objects, *'myObject* sending messages exclusively to *'myObject2*, the following predicate should be a system invariant:

```
length(H / from('myObject)) eq length(H / to('myObject2)) .
```

Furthermore, we can also combine two or more projections. For example if we wish to look only at messages from a given object *'A* to an object *'B*, we can use the following construct:

```
H / from('A) / to('B)
```

Intuitively, the projection shown above is logically equivalent to the following:

```
H / ( from('A) and to('B) )
```

However, what if we in a similar way wish to define an operator *or : Projection* $\times$ *Projection* $\rightarrow$ *Projection*? This actually forces us to take into consideration something we have gently skipped up till now; should the projections happen at the element level, or at the list level? In other words, should a projection such as *from('A)* be applied to a list of messages, or should it be applied to each element/message in the list?

At first glance, those two options might seem to be the same, however if we look at our example with the *or* operator, we see that this is not the case. Consider the following example:

Say we have a message list *ML* looking like this:

```
(msg 'M from 'A to 'B) @ (msg 'M from 'B to 'C) @
(msg 'M from 'C to 'A) @ (msg 'M from 'A to 'C)
```

Suppose now we want to check some property with regards to messages that are sent from *'A* or destined for *'A*. If we do the projection at the list level, we would do the following (assuming functions *from* and *to* are defined for message lists):

```
ML / from('A) or ML / to('A)
```

This would result in the following list:

```
(msg 'M from 'A to 'B) @ (msg 'M from 'A to 'C) @
(msg 'M from 'C to 'A)
```

On the other hand, using a projection on the element level, we would have this construct

```
ML / ( from('A) or from('B) )
```

and the resulting list would be

```
(msg 'M from 'A to 'B) @ (msg 'M from 'C to 'A) @
(msg 'M from 'A to 'C)
```

As we see, in the second list, the original internal order of the list is preserved, and for this reason, it is clear that projections should be applied at the element level instead of at the list level.

In addition to the projections defined above, other projections can be defined in a similar manner, for example

- *msgType* - spanning only messages of a given type
- *not From* - spanning only messages that are not from a given object

and so on.

5.3 Parsing and checking the predicates

In the previous section we built predicates from constructor operators. To make use of the predicates, we need to be able to parse them and check them against our real communication history at any given time. For this purpose we introduce a new operator:

```
op CheckPredicate : Pred MsgList -> Bool .
```

The operator *CheckPredicate* takes a predicate as defined in the previous sections, and a message list, which is the sort our communication history will be of, and returns a boolean value *true* if the history is in compliance with the predicate, and *false* otherwise.

The *CheckPredicate* operator cannot, however, just perform a simple check and return *true* or *false*. In order to compute its desired boolean return value, it will have to parse the predicate and call appropriate auxiliary functions to check the different parts it consists of.

CheckPredicate will be recursively defined, with one equation for each outermost predicate operator. As an example, consider the equation for the case in which the outermost operator is the binary constructor *and* as defined in the previous section

```
vars P1 P2 : Pred . var ML : MsgList .
eq CheckPredicate(P1 and P2, ML) = CheckPredicate(P1, ML)
   and CheckPredicate(P2, ML) .
```

The equation above uses the built-in boolean *and* operator together with recursive calls to CheckPredicate to compute the result of the equation. Note how the built-in *and* operator differs from the predicate constructor *and*; the former takes two boolean arguments, and returns a boolean value, while the latter takes two arguments of the sort *BoolExpr* and returns an expression of the same sort.

In a similar manner, equations for the operators *eq*, *lte* and so on, can be defined:

```
vars IP1 IP2 : IntExp . var ML : MsgList .
eq CheckPredicate(IP1 eq IP2, ML) =
   ReduceIntExp(IP1, ML) == ReduceIntExp(IP2, ML) .
```

In the equation above, the auxiliary *ReduceIntExp* is used to reduce an integer expression to an integer that can be compared with the built in equality operator $==$. The signature for *ReduceIntExp* is defined as follows:

```
op ReduceIntExp : IntExpr MsgList -> Int .
```

ReduceIntExp is recursively defined. The base case in which the *IntExp* argument is just a an integer is straightforward:

```
eq ReduceIntExp(I, ML) = I .
```

However, in the general situation, a term of sort *IntExp* will consist of a function *f* (e.g. *length*), applied to a term of sort *History* (for example the placeholder *H* or a projection). Regarding the function *f*, it would be desirable to avoid having to write separate equations for each such defined function, and rather have a generic equation for arbitrary functions of the correct sort and arity. For this we would need higher order functions. This is unfortunately not available in Maude at the time being, so we will have to settle for the less desirable approach: write one *ReduceIntExp* equation for each function.

So, as an example, consider the equation for the *length* function:

```
var ML : MsgList . var HIST : History .
eq ReduceIntExp(length(HIST), ML) =
   length(ParseProjection(HIST, ML)) .
```

The outermost *length* function to the right of the equals operator (=) is the one defined in the beginning of this chapter; it takes an actual message list and returns an integer equal to the number of elements in the message list. The *length* function on the left hand side of the equation, on the other hand, is a predicate constructor function. Also note that we have introduced a new operator *ParseProjection*, that takes a history and a message list, and returns a message list. This function applies the projection to the message list representing our communication history. It is defined as follows (the equations are numbered from one to three):

```
op ParseProjection : History MsgList  -> MsgList .

var ML : MsgList . var HIST : History .
var M : Msg . var PROJ : Projection .

eq ParseProjection(H, ML) = ML .                  (1)
eq ParseProjection(HIST / PROJ, nil) = nil .      (2)

eq ParseProjection(HIST / PROJ, M @ ML) =         (3)
   Project(ParseProjection(HIST, M), PROJ)
   @ ParseProjection(HIST / PROJ, ML) .
```

The first equation substitutes the history placeholder *H* for the real recorded communication history in the message list *ML*.

The second equation states that any projection applied to an empty message list (communication history) is the empty list.

The final equation is a bit more complex, and the first thing we note is that the second argument to *ParseProjection*, the message list, is expanded in the equation's left hand side, so that we now have a single message *M*, and the rest of the message list *ML*. Hence, the third line in equation three is a tail recursive call for processing the rest of the list (ML). In the second line, the message *M* is processed, and since projections happen at the element level, a call to the *Project* function (which we have not yet defined) is made. The reason for the recursive call to *ParseProjection* within the call to *Project* is that the projection itself may be projected (e.g. as in `H / PROJ1 / PROJ2`, for a history *H* and projections *PROJ1* and *PROJ2*).

Moving on to the final function of this section, the *Project* function has the following signature:

```
op Project : Msg Projection -> MsgList .
```

The task of this function is to check whether a given message is included in a given projection, and if so return this message (we remember that a single message is also a message list consisting of one element), otherwise return the empty list (*nil*).

Depending on the actual projection that is used, the *Project* function calls auxiliary functions. For example, if the *from* projection is used (spanning only messages from a given object), we have the following:

```
 eq Project(M, from(Q1)) = from(Q1, M) .
```

The *from* function checks whether a given message was sent by a given object:

```
op from : Qid Msg -> MsgList .
eq from(Q1, (msg M from Q2 to Q3) @ ML) =
   if Q1 == Q2 then
      (msg M from Q2 to Q3)
   else
      nil
   fi .
```

Other projection functions, e.g. *to* that returns messages sent to a given object, can now be defined in the same way as we did with *from*, by providing another equation for *Project* and implementing the *to* function correspondingly.

5.4 Integrating predicates with the rewrite strategy

To be able to use the mechanisms that we have defined in this chapter, we must integrate them with the rewrite strategy that we initially defined in Chapter 4.

First of all, since our predicates are functions on the communication history, it seems natural to store the predicate in the *History* object (from Section 4.2). We add an extra attribute *pred* to its definition in the following way:

```
op History[h:_ , pred:_] : MsgList Pred -> EngineObject .
```

Furthermore, in order to make use of a given predicate, we must be able to check whether it is violated by the current communication history or not. The way we will approach this, is to have the rewrite strategy check whether a given rewrite will result in a state that is in violation of the predicate, and if so, choose not to execute that rewrite rule at that point. The code for this strategy is shown in Figure 5.1 on the following page.

The main change from our previous strategy from Figure 4.3 on page 41 is that we in lines 12–14 now perform a check against the predicate in the history object using the *CheckPredicate* function we defined earlier in this chapter. If this check fails, the rule is not applied and the rule label is added to the list of failed rules.

In other words, what we are doing with this code is to force the specification to behave in accordance with our predicate. Later on, in Section 8.3, we will look at another approach; instead of forcing the specification to behave, we will halt the execution in the event that it attempts to do something illegal.

```
1.  crl [exec-pred] :
2.    Engine[curTerm: T, curModule: MOD, labels: LABEL LABELS,
3.      failedRules: FAILEDRULES]
4.    History[h: ML, pred: PRED]
5.  =>
6.    if metaXapply([MOD], T, LABEL, none, 0, 1, 0) == failure
7.    then
8.      Engine[curTerm: T, curModule: MOD, labels:
9.        LABELS LABEL, failedRules: FAILEDRULES LABEL]
10.     History[h: ML, pred: PRED]
11.   else
12.     if CheckPredicate(PRED, ML @
13.       getNewMessages(T, getTerm(metaXapply([MOD],
14.       T, LABEL, none, 0, 1, 0))))
15.     then
16.       Engine[curTerm: getTerm(metaXapply([MOD], T, LABEL,
17.         none, 0, 1, 0)), curModule: MOD, labels:
18.         LABELS LABEL, failedRules: nil]
19.       History[h: ML @
20.         getNewMessages(T, getTerm(metaXapply([MOD], T,
21.         LABEL, none, 0, 1, 0))), pred: PRED]
22.     else
23.       Engine[curTerm: T, curModule: MOD, labels:
24.         LABELS LABEL, failedRules: FAILEDRULES LABEL]
25.       History[h: ML, pred: PRED]
26.     fi
27.
28.   fi
29. if length(FAILEDRULES) < length(LABEL LABELS) .
```

Figure 5.1: A rewrite strategy that checks whether rewrites will lead to a state that violates the predicate

Chapter 6

Regular Expressions in Predicates

Regular expressions are well known and widely used in computer science to describe patterns of various kinds. In this chapter we will develop mechanisms that allow us to use regular expressions in predicates on communication histories [25, 27], and integrate these mechanisms with the rewrite strategy we defined in the preceding chapter. This will enable us to take full advantage of the expressiveness that regular expressions inherently provide when specifying predicates.

6.1 Data structures for regular expressions

In order to implement regular expressions in Maude, we will make extensive use of so-called finite automata.

Our strategy will be as follows:

1. Define a data structure for specifying the regular expressions themselves.
2. Define a module for a non-deterministic finite automaton (NFA), and transform the regular expression into an NFA using Thompson's ϵ-algorithm [35, 29].
3. Define a module for a deterministic finite automaton (DFA), and transform our NFA to a DFA using the well-known subset construction algorithm [29].

It may seem odd that we go through the trouble of first defining an NFA and then transforming this automaton to a DFA. The reason for this is two-fold: first, the algorithm for transforming a regular expression directly to a DFA is substantially more complex than the corresponding algorithm for transforming a regular expression to an NFA, while the algorithm for

transforming an NFA to a DFA is quite tractable. Second, as we shall see in Section 8.4, the NFA itself will be useful later on.

6.2 Regular patterns

We will now define constructors for regular patterns in Maude. We start by introducing a new module *PATTERN* with a sort *Pattern*. Furthermore, since our regular expressions will be dealing with a communication history consisting of messages, we define the sort *Msg* to be a subsort of *Pattern*. We can now in a quite natural syntax define the pattern constructors:

```
op _::_ : Pattern Pattern -> Pattern [ctor assoc prec 54] .
op _*   : Pattern -> Pattern [ctor prec 53] .
op _+   : Pattern -> Pattern [ctor prec 53] .
op _?   : Pattern -> Pattern [ctor prec 53] .
op _|_  : Pattern Pattern -> Pattern [ctor prec 55] .
```

Note that the `::` operator is used for concatenating patterns (just a blank cannot be used in our Maude specification due to conflicts with previously defined modules). The other operators should be self-explanatory. One final thing to take note of is the precedence that is used; choice (`|`) binds weaker than concatenation (`::`), which in turn binds weaker than the rest of the operators (`*`, `+` and `?`). This is in compliance with standard rules for regular expressions.

6.3 Non-deterministic finite automata

In this section we will show how to transform a regular expression into a non-deterministic finite automaton.

Definition 10 (Non-deterministic finite automaton). *An NFA M consists of an alphabet Σ, a set of states S, a transition function $T : S \times (\Sigma \cup \{\epsilon\}) \rightarrow \wp(S)$, a start state s_0 from S, and a set of accepting states A from S. $\wp(S)$ is the power set of S. The language accepted by M is the set of strings $c_1 :: c_2 :: ... :: c_n$ where $c_i \in (\Sigma \cup \{\epsilon\})$ such that there exists states s_1 in $T(s_o, c_1)$, s_2 in $T(s_1, c_2)$, ..., s_n in $T(s_{n-1}, c_n)$ with $s_n \in A$.*

Thompson's ϵ-algorithm For transforming a regular expression to an NFA, we will be using Thompson's ϵ-algorithm. This algorithm uses so-called ϵ-transitions to "glue together" sub-automata constructed from the individual parts of a regular expression. An ϵ-transition is a transition by which no input is consumed.

A basic regular expression is of the form a, ϵ or ϕ, where a represents a match of a single token from the alphabet, ϵ a match of the empty string, and ϕ a match of no strings.

An NFA for the the regular expression a is shown in Figure 6.1. Note that an accepting state is indicated by a double border.

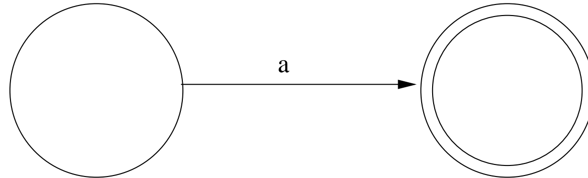

Figure 6.1: An NFA for the regular expression a

For a regular expression $r :: s$, we assume that the sub-automata for r and s have already been constructed, and we can then construct the automaton for $r :: s$ by joining them with an ϵ-transition, as shown in Figure 6.2.

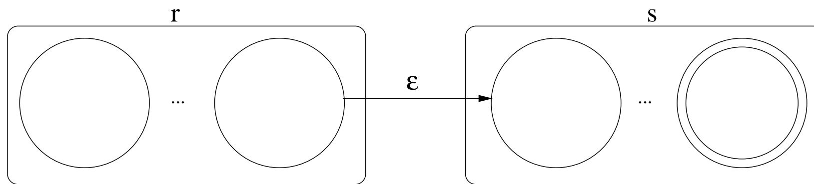

Figure 6.2: An NFA for the concatenation of the regular expressions r and s

Note how the accepting state of r has now been connected to the start state of s, and that the accepting state of s is now the accepting state for the entire automaton. This shows the key point in the ϵ-algorithm; sub-automata are constructed for the individual pieces of a regular expression (without any knowledge of the internals of any other sub-automata), and joined together by ϵ-transitions. For a more thorough presentation of the algorithm, the reader is referred to [35, 29].

Implementation in Maude We start out by defining a module *NFA*. In this module, we introduce the sort *NFA*. A general NFA consists of one or more states, as well as zero or more transitions between the states. Terms of the sort *NFA* will be constructed from a subsort *NFA-State*. For each state in our NFA, we will need to store at least two pieces of information:

- an identifier of some kind, so that we can refer to any given state in an easy manner, and
- a flag stating whether the state is an accepting state or not.

In addition, we choose to store the transitions from a given state A to another given state B as a property of state A, in the same manner as the

identifier and accepting flag are stored. This, as we will see, will ease our computation later on. We can now define a state object in the following manner:

```
op {State:_, Accepting:_, Transitions:_} :
   String Bool TransitionSet -> NFA-State .
```

The state identifier is, as we can see from the definition above, of sort *String*. One might think it more natural to use e.g. an integer as identifier instead, however, as we will see later on, this would not be as easy to accomplish as one might expect.

With our definition of an NFA-State well in hand, we need to define the second important part of our NFA; the transitions. A transition can be seen as a tuple *(O, T, D)*, where O is the state from which the transition originates, D is the destination state, and T is the token that must match the next item in the input if a transition is to be allowed from O to D. The sort *Msg* is a subsort of the sort *Token*.

Since, as mentioned above, we keep the outgoing transitions from a given state inside the state itself, the value of O is implicit, and we do not need to keep this piece of information in the transition itself. Hence, we will define a transition in the following manner:

```
op _->_ : Token String -> Transition .
```

Assuming $S1$ and $S2$ are state names in an NFA, and $T1$ is a token, a transition from state $S1$ to state $S2$ over $T1$, would be represented as follows

```
T1 -> S2
```

and be contained within state $S1$. State $S1$ would therefore be represented in the following manner:

```
{State: S1, Accepting: B, Transitions: (T1 -> S2) TS}
```

where B is a boolean value indicating whether $S1$ is accepting or not, and TS is a (possibly empty) set containing other transitions from $S1$.

Now that we have defined the basic structure for an NFA, we can move on to our main goal in this section; to transform regular patterns defined with the constructs introduced in Section 6.2 to an NFA. To achieve this, we will make use of Thompson's ϵ-algorithm, as described above.

To begin with, we introduce an operator *MakeNFA : Pattern* $\rightarrow$ *NFA*, that will serve as our starting point for building the NFA from a specified pattern. Second, we introduce an auxiliary operator that overloads the one we just defined: *MakeNFA : Pattern* $\times$ *String* $\times$ *String* $\rightarrow$ *NFA*. This function is the one that actually builds the NFA, and we will take a closer look at its implementation below.

Our second signature for *MakeNFA* takes two strings in addition to the pattern as its parameters. The first string parameter is the name of the state to be constructed, and will be passed in from the caller (which will be another instance of the same function, since it is recursive, as we will see). The reason the name is passed in from the calling function, is that the calling function needs to know the name of the new state in order to create a transition leading to it.

Any given regular pattern will either be a single token, or have an outermost operator, like for example repetition (*) or concatenation (::). Hence, we can define one equation for each case:

- 0 or more repetitions:

  ```
  eq MakeNFA(P1 *, NAME, NEXTSTATE) = ...
  ```

- Concatenation:

  ```
  eq MakeNFA(P1 :: P2, NAME, NEXTSTATE) = ...
  ```

- Choice:

  ```
  eq MakeNFA(P1 ?, NAME, NEXTSTATE) = ...
  ```

and so on.

Let us now take a closer look at the equation in which the operator for 0 or more repetitions, the Kleene star *, is the outermost operator:

```
1.   eq MakeNFA(P1 *, NAME, NEXTSTATE) =
2.     {State: NAME + "1", Accepting: false, Transitions:
3.       (epsilon -> NAME + "3")
4.       (epsilon -> NAME + "21")
5.     }
6.     MakeNFA(P1, NAME + "2", NAME + "3")
7.     {State: NAME + "3", Accepting: NEXTSTATE == "",
8.       Transitions: (epsilon -> NAME + "1")
9.       if NEXTSTATE =/= "" then
10.         (epsilon -> NEXTSTATE)
11.      else
12.         emptyTransitionSet
13.      fi
14.    } .
```

P1 is of sort *Pattern*, as is *P1 **, and *NAME* and *NEXTSTATE* are of sort *String*.

First, in lines 2–4 an NFA state named *NAME + "1"* is constructed, meaning that this is the first state constructed by this particular function call ("1"). This number is concatenated with the string already present in the parameter *NAME*. In other words, if this state is the first state in the entire NFA, it will be named "1" (the *NAME* parameter would be empty). If, on the other hand, it is for example the first state in the third outermost operator, it might be given a name like "221".

In lines 3 and 4, we create two new transitions to other states. The transition in line 3 is an ϵ (epsilon) transition to the last state in the sub-NFA created by this call to *MakeNFA* (this state is created in lines 7–13). This transition represents a choice of zero repetitions of the pattern *P1*.

The transition in line 4 is an ϵ-transition to the "internals" of the pattern *P1 **, namely *P1*, which will be constructed by a recursive call to *MakeNFA* in line 6. It is worth noting in line 6 the third parameter in the call, *NAME + "3"*. This is the name of the state that comes after the last state of the as of yet not created sub-NFA from pattern *P1*.

In lines 7–13 we create the last state of this NFA. We note that the accepting flag is true only if this state has no successors, and in the opposite situation, an ϵ-transition is created to the succeeding state as specified in the parameter *NEXTSTATE*.

The observant reader may have wondered why the identifiers/names of the states are represented as strings and not for example integers, and why they are concatenated in such a way, e.g. "2321" etc in stead of just a standard incremental number. In other words, why are the states not just named *1, 2, 3, ..., n*? There are two main reasons for this:

- First, when the NFA of a pattern like *P1 ** is constructed, we need to create both the state leading to the sub-NFA constructed from *P1*, and the state succeeding it, as well as the transitions between these states, and we need to know their names to be able to do this. If we were to use incremental integers, we would have to know how many states that would be created from *P1*, and this is clearly not a trivial problem.

- Second, there is no simple way to construct a function that returns successive integers in Maude, without having to pass the previous integer as a parameter, and that would defy the entire purpose of using such a function.

An example In this paragraph, we will look at an example of a regular expression, and how the NFA that can be created from this expression will look. The regular expression that we will consider, is the following:

$$(a \mid b)^* \;::\; c$$

This expression corresponds to strings consisting of the zero or more a's and b's followed by a c. For example, the following would be valid strings:

$$aaaaaac, abaabac, bc, c$$

while the following strings are not valid:

$$a, b, ca, aaaba$$

By using the *MakeNFA* equation, we can turn the regular expression into an NFA by using Thompson's ϵ-algorithm. A Maude representation of the resulting automaton is show in Figure 6.3.

```
{State: "1",    Accepting: false, Transitions: (epsilon ->"21")}
{State: "21",   Accepting: false, Transitions:
  (epsilon ->"221") (epsilon -> "23")}
{State: "221",  Accepting: false, Transitions:
  (epsilon -> "2221") (epsilon -> "2231")}
{State: "2221", Accepting: false, Transitions: (a -> "2222")}
{State: "2222", Accepting: false, Transitions: (epsilon -> "224")}
{State: "2231", Accepting: false, Transitions: (b -> "2232")}
{State: "2232", Accepting: false, Transitions: (epsilon -> "224")}
{State: "224",  Accepting: false, Transitions: (epsilon -> "23")}
{State: "23",   Accepting: false, Transitions:
  (epsilon -> "21") (epsilon -> "3")}
{State: "3",    Accepting: false, Transitions: (epsilon -> "41")}
{State: "41",   Accepting: false, Transitions: (c -> "42")}
{State: "42",   Accepting: false, Transitions: (epsilon -> "5")}
{State: "5",    Accepting: true,  Transitions: emptyTransitionSet}
```

Figure 6.3: A Maude representation of an NFA for the regular expression $(a \mid b)^* \ :: \ c$.

We note how the first state, state 1, has an ϵ-transition to the second state, 21. The second state marks the beginning of the sub-pattern $(a \mid b)^*$. It has one transition to the internals of this pattern, $(a \mid b)$, and one to the next state *after* the entire pattern, which is state 23. Following the transition directly to state 23 allows for zero repetitions of $(a \mid b)$.

State 221 is the first state of the sub-pattern $(a \mid b)$. It has one ϵ-transition to each of its sub-patterns, a and b, which starts with states 2221 and 2231, respectively.

In state 23, we note how one ϵ-transition leads back to the start of the sub-pattern $(a \mid b)^*$, allowing for more than one repetition, and one ϵ-transition leads to the beginning of the pattern c, in state 3.

Finally, we note how state 5 is the only state with the *Accepting* flag set to *true*, and that there are no transitions originating from it.

6.4 Deterministic finite automata

Now that we have the data structures and equations needed to create an NFA from a regular pattern, we can move on to create the corresponding *deterministic* finite automaton.

Definition 11 (Deterministic finite automaton). *A DFA M consists of an alphabet Σ, a set of states S, a transition function $T : S \times \Sigma \to S$, a start state $s_0 \in S$, and a set of accepting states A from S. The language accepted by M is the set of strings $c_1 :: c_2 :: ... :: c_n$ where $c_i \in \Sigma$ such that there exists states $s_1 = T(s_o, c_1)$, $s_2 = T(s_1, c_2)$, ..., $s_n = T(s_{n-1}, c_n)$ with $s_n \in A$.*

Subset construction algorithm To construct a DFA from an NFA, we will be using the *subset construction* algorithm. What we are trying to achieve in this algorithm, is to remove all ϵ-transitions from the automaton, and eliminate multiple transitions over a single token from a given state.

The algorithm makes use of *ϵ-closures*. An ϵ-closure for a given state s is the set of states reachable by following zero or more ϵ-transitions from s. This set is conventionally denoted $\overline{s}$ (even though the notation is identical, this should not be confused with the meta-representation of a term t, denoted $\overline{t}$ in Chapter 2).

For a set of states S, the ϵ-closure of this set, $\overline{S}$, is defined as the union of the ϵ-closures of each state s in S.

The subset construction algorithm is defined as follows (the definition is from [29]): Given a set S of NFA states and a token a, compute the set $S'_a = \{t|$ for some s in S there is a transition from s to t on a $\}$. Then, compute $\overline{S'_a}$, the ϵ-closure of S'_a. This defines a new state in the subset construction, together with a new transition $S \xrightarrow{a} S'_a$. Continue with this process for every state s in S, until no new states or transitions are created. Mark as accepting those states constructed in this manner that contain an accepting state from the NFA. This will construct a DFA from an NFA.

Implementation in Maude We start out by defining an appropriate data structure for DFA states and transitions. The states are represented as follows:

```
op {State:_, Accepting:_, Transitions:_} :
     StateSet Bool TransitionSet -> DFA-State .
```

We note that the state name/identifier is of sort *StateSet*. This is because a DFA state may represent several states from the NFA. (The state set does not, however, contain the actual NFA states themselves, but rather their names.)

The representation of a DFA transition is pretty straightforward and resembles that of an NFA transition.

```
op _->_ : Token StateSet -> DFA-Transition .
```

In the same way as for NFA states, the DFA states will contain the transitions leading from them, and there is therefore no need to maintain information concerning the originating state's name in the DFA transition.

To create a DFA from a given NFA, we will use the subset construction algorithm, as described above.

Our algorithm for creating an NFA made extensive use of ϵ-transitions. Such transitions are not allowed in a deterministic automaton. We also need to get rid of any two transitions leading from one state over the same token to different destination states, as there can be only one transition over any token from a given DFA state (otherwise, the automaton would not be deterministic). This involves creating a closure with regards to the token.

However, before we delve any deeper into the intricacies of implementing the subset construction, we start out by defining an operator *MakeDFA : NFA* $\rightarrow$ *DFA*. This will be the interface to the "outside", or in other words, this will be the function that is called from modules using our DFA module.

We then move on to look at the perhaps most important part of the algorithm, the part that implements the ϵ-closures. For this, we define a function *eClosure : String NFA* $\rightarrow$ *StateSet*. The string parameter is the name of the state for which we want to create the closure, and the parameter of sort *NFA* is an NFA as created with the aid of the constructs introduced in section 6.3. The return value is of sort *StateSet*, and contains the names of the states reachable by following one or more transitions over ϵ from the state given in the first parameter (i.e., it returns the ϵ-closure).

The implementation of this algorithm in Maude is (unfortunately) not all that straightforward. What we need to do, basically, is to check ϵ-transitions recursively, and at the same time keep track of the states included in the closure up till the current call, in order to avoid infinite recursion. To be able to keep track of our progress, we overload our *eClosure* function, adding a third parameter for the result so far:

```
op eClosure : String NFA StateSet -> StateSet .
```

Furthermore, we let our first *eClosure* signature call the latter, with the third parameter *emptyStateSet* (since this is the first time we call the function, and our result so far is thus empty):

```
eq eClosure(S1, {State: S1, Accepting: B,
   Transitions: TS} NFA) =
   eClosure(S1, {State: S1, Accepting: B,
   Transitions: TS} NFA, emptyStateSet) .
```

Now we are ready to look at the implementation of the second equation for the *eClosure* function. We start with the non-recursive case first:

```
eq eClosure(S1, {State: S1, Accepting: B, Transitions:
  emptyTransitionSet} NFA, SS) = S1 .
```

The equation above states that the ϵ-closure of a state with no transitions, is the state itself.

The recursive case, on the other hand, is somewhat more involved:

```
1.      eq eClosure(S1, {State: S1, Accepting: B,
2.       Transitions: (T -> S2) TS} NFA, SS) =
3.       S1,
4.       if T == epsilon then
5.         if not S2 in SS, S1 then
6.          S2, eClosure(S2, ({State: S1, Accepting: B,
7.            Transitions: (T -> S2) TS} NFA), (SS, S1, S2))
8.         else
9.           emptyStateSet
10.         fi
11.       else
12.         emptyStateSet
13.       fi,
14.       eClosure(S1, {State: S1, Accepting: B,
            Transitions: TS} NFA, SS,
15.         S1,
16.         if T == epsilon then
17.           if not S2 in SS, S1 then
18.             S2, eClosure(S2, ({State: S1, Accepting: B,
19.               Transitions: (T -> S2) TS} NFA), (SS, S1, S2))
20.           else
21.             emptyStateSet
22.           fi
23.         else
24.           emptyStateSet
25.         fi
26.       ) .
```

In line 3, which is the first line of the right hand side of the equation, we state that any given state is part of its own ϵ-closure. In lines 4–13, we check to see if the first transition *(T -> S2)* in the state's transition set is an ϵ-transition. If so, we check in line 5 whether we have already included state $S2$ in our closure. If either of these checks fail, we add nothing to our closure (or, in other words, we add an empty state set). Otherwise, we add $S2$ to our closure (in line 6), and call recursively, this time for state $S2$ (as there might be ϵ-transitions going from $S2$ as well). Note how the third parameter, the result so far, now has become (*SS, S1, S2*).

Moving on to the second half of the function, lines 14–26 contains a recursive call for state $S1$, having removed the transition *(T -> S2)* that we have already processed. As the observant reader may have noticed, lines 3–13 are actually equal to lines 15–25. This is due to the fact that the first lines contain the result from the current call to the function, and hence this needs to be included in the result so far in the recursive call.

As we stated above, removing ϵ-transitions is not our only task. We also need to deal with states that contain more than one transition over a given token T. Realizing that T might be ϵ as well as any other token, we may define a function that returns the ϵ-closure for all states that can be reached by following transitions over T from a given state S. Hence we define a function *eClosures : Token* $\times$ *TransitionSet* $\times$ *NFA* $\rightarrow$ *StateSet*, that will make use of our already defined function *eClosure* in the following manner:

```
eq eClosures(T, emptyTransitionSet, NFA) = emptyStateSet .
eq eClosures(T, (T2 -> S1) TS, NFA) =
   if T == T2 then eClosure(S1, NFA) else emptyStateSet fi,
   eClosures(T, TS, NFA) .
```

As we have now defined the necessary functions for creating the closures, we move on to looking at our main function, namely *SubsetConstruction*. Its task will be to create DFA states with the aid of our closure functions. It is defined as follows:

```
op SubsetConstruction : NFA -> DFA .
eq SubsetConstruction(NFA) = SubsetStart(NFA)
   Subset2(SubsetStart(NFA), NFA, SubsetStart(NFA)) .
```

As we can see, this function is a wrapper for the auxiliary functions *SubsetStart* and *Subset2*. The *SubsetStart : NFA* $\rightarrow$ *DFA* function creates the first state in our DFA. The interested reader may look up its full definition in the source code in Appendix A, as its details are not vital to the rest of this discussion.

The *Subset2* function, on the other hand, is a bit more involved. It has the following signature: *Subset2 : DFA-State* $\times$ *NFA* $\times$ *DFA* $\rightarrow$ *DFA*. Its first parameter is the current DFA state (the very first state is constructed using the *SubsetStart* function mentioned above). The second parameter is the NFA from which we are constructing our DFA, and the third is the result so far.

The *Subset2* function is defined in the following manner:

```
1.  eq Subset2({State: SS, Accepting: B, Transitions:
2.       (T -> S1) TS}, NFA, RESULT-SO-FAR) =
3.      if MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA)
```

```
       in RESULT-SO-FAR
4.    then
5.      emptyDFA
6.    else
7.      MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA)
8.    fi
9.    Subset2({State: SS, Accepting: B, Transitions: TS},
10          NFA, RESULT-SO-FAR
11          MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA))
12    if MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA)
13      in RESULT-SO-FAR
14    then
15      emptyDFA
16    else
17      Subset2(MakeDFAState(eClosures(T, (T -> S1) TS,
          NFA), NFA),
18       NFA, RESULT-SO-FAR
19       MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA))
20    fi .
```

The first thing we note about the above algorithm, is its use of the function *MakeDFAState*. All this function does, as its name implies, is to create a valid DFA state from its parameters, a state set (containing as we remember the names of NFA states), and an NFA. Again, the interested reader might look up its full definition in the source code.

Other than that, we see that the *Subset2* function follows a pattern similar to some of our previous function definitions; the first part (lines 3–8) checks to see whether the state we are about to create already exists, and if so does nothing, the second part (lines 9–11) does a recursive call excluding the transition we have already processed *(T -> S1)*, and finally the third part (lines 12–20) does essentially the same as the first part, except that it makes a recursive call with the newly created state as the current one.

Having come this far, we have almost completed our goal of this section, to transform an NFA to a DFA. Only some "glue" remains, in order to make the separate pieces work well together, as well as some necessary auxiliary functions. We will settle for taking a final look at the *MakeDFA* function, our interface to the outside, as defined in the beginning of this section. We are now ready to understand its equational specification:

```
eq MakeDFA(NFA) = RenameTransitions(
   {State: "START", Accepting: false, Transitions:
    start -> "1"} SubsetConstruction(NFA), NFA) .
```

The first thing to note is the *RenameTransitions* function that wraps around the rest of the right hand side of the equation. This function takes

care of a problem that we have overlooked this far; the transition within a DFA state needs to lead to the ϵ-closure of the NFA states they were originally leading to. The full definition of this function can be found in Appendix A.

Second, we create a *START* state to ease our checking later, allowing us easy access to the start of the automaton. Finally, we make a call to the *SubsetConstruction* function.

An example We will now consider a DFA created from the NFA that we looked at in the example in the previous section. This NFA was made from the regular expression $(a \mid b)^* \; :: \; c$, and can be found in Figure 6.3 on page 67.

By utilizing the subset construction algorithm explained above, a DFA can be made from this NFA in the following manner:

```
red MakeDFA(MakeNFA((a | b) * :: c)) .
```

The resulting DFA is shown in Figure 6.4.

```
{State: "START",
  Accepting: false,
  Transitions: start -> "1","21","221","2221","2231","23","3","41"}
{State: "1","21","221","2221","2231","23","3","41",
  Accepting: false,
  Transitions: (a -> "21","221","2221","2222","2231","224","23","3","41")
               (b -> "21","221","2221","2231","2232","224","23","3","41")
               (c -> "42","5")}

{State: "21","221","2221","2222","2231","224","23","3","41",
  Accepting: false,
  Transitions: (a -> "21","221","2221","2222","2231","224","23","3","41")
               (b -> "21","221","2221","2231","2232","224","23","3","41")
               (c -> "42","5")}
{State: "21","221","2221","2231","2232","224","23","3","41",
  Accepting:false,
  Transitions: (a -> "21","221","2221","2222","2231","224","23","3","41")
               (b -> "21","221","2221","2231","2232","224","23","3","41")
               (c -> "42","5")}
{State: "42","5",
  Accepting:true,
  Transitions: emptyTransitionSet}
```

Figure 6.4: A Maude representation of a DFA for the regular expression $(a \mid b)^* \; :: \; c$.

We note that this automaton has only five states (including the start state), as opposed to the NFA which had thirteen states. There are no ϵ-transitions, and there is at most one transition over any given token in any

given state. Furthermore, we see that the states have names that are sets of NFA state names, as mentioned above. This is also the reason for why the special *START* state is useful — we have no easy way of knowing what the name of the actual start state (in this case "1", "21", "221", "2221", "2231", "23", "3", "41") will be.

This concludes our discussion with regards to creating a DFA from an NFA. As mentioned before; the most important issues are explained, but there are still some details that are skipped for the sake of conciseness, which can be looked up in Appendix A.

6.5 Putting the regular expressions to work in predicates

Up to this point in this chapter, we have looked at how to create automata that can be used to check regular expressions, but we have not integrated this with the rest of the predicates — that is what we will look at below.

In order for a given deterministic finite automaton to be usable, we clearly need some way of checking whether a given message list is a sequence that the automaton accepts. For this purpose, we introduce the function *Match*:

```
op Match : TokenList DFA -> Bool .
```

Match will return *true* if the list of tokens (which in our case will be a list of messages) matches the regular pattern from which the DFA is made, and *false* otherwise. In order to achieve that, the function must traverse the DFA recursively using the defined transitions between the automaton's states.

As mentioned in section 6.4, the DFA has a special initial state named *START*. We make use of this in the match function, to find the first state, and make a recursive call to start the actual checking of the communication history in the parameter *TL* (of sort *TokenList*):

```
eq Match(TL, {State: "START", Accepting: B,
   Transitions: (start -> SS)} DFA)  =
   Match(TL, DFA, SS) .
```

We note that the recursive call to *Match* makes use of a third parameter, *SS*. This parameter is of sort *StateSet* (which is used as names for DFA states), and is used to identify the current state. The *Match* function is, in other words, overloaded, and has the following signature in addition to the one presented above:

```
op Match : TokenList DFA StateSet -> Bool .
```

Moving on to the implementation of this function, we have the following equation:

```
var T1 : Token . var TL : TokenList . var SS : StateSet .
var B : Bool . var TS : TransitionSet . var DFA : DFA .

eq Match(T1 @ TL, {State: SS, Accepting: B, Transitions: TS}
     DFA, SS) =
   if FindTransition(T1, TS) =/= emptyStateSet then
      Match(TL, {State: SS, Accepting: B, Transitions: TS}
        DFA, FindTransition(T1, TS))
   else
      false
   fi .
```

As we see, this equation checks, with the help of the auxiliary *FindTransition*, whether the transition set in the current state (*TS*) has a transition over token *T1*. If this is the case, a recursive call is made with the state to which this transition leads now being the current state. Otherwise, false is returned, meaning that the check failed.

If the communication list is the empty list *nil* (all tokens have been checked), the *Match* function should return *true* if the current state is an accepting state, and *false* otherwise. This can be handled by an equation as shown below (where the *IsAccepting* function returns *true* if the current state *SS* is indeed an accepting state, and *false* otherwise):

```
eq Match(nil, DFA, SS) = IsAccepting(SS, DFA) .
```

However, this is in many scenarios not what we want. Rather, we would want the check to succeed if the recorded history *this far* is in compliance with the predicate. In other words, the matching function should return *true* if the communication history is a prefix of the sequence defined by the regular expression. For this, we introduce the *prs* operator:

```
op _prs_ : TokenList DFA -> Bool .
```

Now, we could implement this function in more or less the same way as we have done with the *Match* function, but seeing as their functionality is essentially the same, we instead add a fourth boolean parameter to the definition of *Match*, to indicate whether prefix matching is to be used, and let the *prs* operator call *Match* with this parameter set to true:

```
op Match : TokenList DFA StateSet Bool -> Bool .

eq TL prs DFA = Match(TL, DFA, emptyStateSet, true) .
```

All we need to change in the match function now, is the equation for which the communication history is the empty list. It will now look like this:

```
eq Match(nil, DFA, SS, PRS) = PRS or IsAccepting(SS, DFA) .
```

In this way, the *Match* function will return *true* if all the input has been checked and prefix matching is used.

This concludes our discussion on regular expressions in predicates for now. We have shown how to transform a regular expression to a deterministic finite automaton via a non-deterministic one, and how to check if a given message list is accepted by a given automaton. In Section 8.4, we will take a second look at these mechanisms, as variables are introduced in predicates.

Chapter 7

Examples

In this chapter, we will take a look at some scenarios in which the mechanisms that we have developed so far can be put some actual use.

7.1 Simple producer-consumer specification

The example we will be looking at below is a variant of the well-known producer-consumer problem. Our example specification consists of three objects; a producer *'P*, a buffer *'B*, and a consumer *'C*. The producer knows the identity of the buffer, and is hence capable of sending messages to it. Likewise, the consumer is also aware of the buffer's identity, whereas the buffer knows basically nothing about its surroundings.

Our initial state will be as follows:

```
< 'P : Producer | Buffer: 'B >
< 'C : Consumer | Buffer: 'B >
< 'B : Buffer > .
```

The producer will send *Put*-messages to the buffer, in order to put data into it. The buffer will reply with a *PutOk*-message to signal to the producer that it has written the data to its internal data store.

The consumer, on the other hand, will send *Get*-messages to the buffer, in order to read the (imaginary) data stored inside of it. The buffer will then respond with a *GetOk*-message to the consumer, in order to send the requested data back. However, since the actual data is irrelevant to the predicate that we will define, the messages in our example will not contain any data.

Clearly, the consumer cannot read what the producer has not yet written to the buffer, and this will be our invariant; the number of *PutOk* messages from the buffer to the producer must be equal to or greater than the number of *GetOk*-messages from the buffer to the consumer. In order to express this, we could have written specific conditional rules in the producer and

consumer to make sure the invariant is never violated, but with the use of our communication history, we may take another approach. We can define the rewrite rules as simple as possible, making no assumptions with regards to the contents of the buffer. For example, the rule where the buffer responds to the consumer with a *GetOk*-message, looks like this:

```
rl [get-ok] :
  < B : Buffer >
  msg Get from C to B
=>
  < B : Buffer >
  msg GetOk from B to C .
```

Now, this would obviously violate our invariant if the buffer was empty, so we need to prevent this from ever happening. And it is here our predicates on the communication history comes into play. We may define our invariant in the following way using the constructs introduced in Chapter 5:

```
length(H / msgType('PutOk)) >= length(H / msgType('GetOk))
```

Because our meta-rewriting engine always checks whether the predicate will be violated before applying any given rewrite rule, we have successfully established our invariant.

We may also use regular expressions in our predicates, as defined in Chapter 5. To achieve a similar invariant (though with some further restrictions on the history as well), we could define our predicate in the following way:

```
H prs ((msg 'Put from 'P to 'B) :: (msg 'PutOk from 'B to 'P)
:: (msg 'Get from 'C to 'B) :: (msg 'GetOk from 'B to 'C)) *
```

This predicate ensures that the messages in the history always are in the following order: *Put*, *PutOk*, *Get*, *GetOk*.

7.2 The dining philosophers

In this section we will take a look at a classic synchronization problem in computer science; the dining philosophers [15].

The example goes as follows: n philosophers, where $n > 1$, are seated around a circular table, and in the middle of the table, there is a bowl of food.

Now, since the guests at our dinner party are philosophers, they spend most of their time around the table thinking, however, even bright minds need to eat. So, at some point in time a given philosopher will get hungry and will hence want eat from the bowl of food on the table. Unfortunately,

there are only n chopsticks available, and each philosopher will need two sticks to be able to get hold of any food. Therefore, they cannot all eat at the same time.

To make the situation worse, once a philosopher has gotten hold of a chopstick, he will not let go of it until he has gotten hold of the other one, and finished his eating. So, if all the philosophers pick up their left stick at the same time (or at least before any of the other philosophers have picked up their right stick), they will all starve to death(!) waiting for their right stick. The philosophers' behavior may in other words give rise to a deadlock situation.

In Maude, we can represent a philosopher as an object in the following manner:

```
op <_: Philosopher | state:_, leftStick:_, rightStick:_ ,
   butler:_, seat:_> : Qid State StickState StickState
   Qid Nat -> Phil .
```

Each philosopher sits at a specified seat at the table, indicated by the *seat* attribute. The *state* attribute indicates the activity of the philosopher at present, and the *leftStick* and *rightStick* attributes are boolean flags indicating whether the philosopher is currently in possession of his left and right chopstick, respectively. Furthermore, since the philosophers are lazy, they have a butler, James, to hand them the sticks at their request. The butler is represented as follows:

```
op <_: Butler | sticks:_ > : Qid IntSet -> Butler .
```

The butler has a fixed number (n) of chopsticks that are available to the philosophers, each stick is represented by an integer corresponding to its position on the table. Chopstick number one is placed to the left of seat one, chopstick number two to the left of seat two (and to the right of seat one), and so on.

Below is an initial configuration with $n = 5$, consisting of the philosophers Socrates, Plato, Aristotle, Anaximander and Pythagoras, and the butler James:

```
< 'Socrates : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 1 >

< 'Plato : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 2 >

< 'Aristotle : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 3 >
```

```
< 'Anaximander : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 4 >

< 'Pythagoras : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 5 >

< 'James  : Butler | sticks: 1 2 3 4 5 > .
```

To begin with, all the philosophers are thinking, and have acquired no chopsticks. When a philosopher becomes hungry, he will request sticks from the butler, one at a time. This can be modeled using the following rewrite rules:

```
rl [getting-hungry] :
  < P : Philosopher | state: thinking, leftStick: no,
   rightStick: no, butler: B, seat: I >
=>
  < P : Philosopher | state: hungry, leftStick: no,
    rightStick: no, butler: B, seat: I > .

rl [req-left-stick] :
  < P : Philosopher | state: hungry, leftStick: no,
    rightStick: S1, butler: B, seat: I >
=>
  < P : Philosopher | state: hungry, leftStick: requested,
    rightStick: S1, butler: B, seat: I >
  msg RequestStick(I) from P to B .

rl [req-right-stick] :
  < P : Philosopher | state: hungry, leftStick: S1,
    rightStick: no, butler: B, seat: I >
=>
  < P : Philosopher | state: hungry, leftStick: S1,
    rightStick: requested, butler: B, seat: I >
  msg RequestStick(if I == 5 then 1 else I + 1 fi)
    from P to B .
```

The butler, James, distributes sticks to the philosophers according to the following rewrite rule:

```
rl [give-stick] :
  < B : Butler | sticks: I IS >
  msg RequestStick(I) from P to B
```

```
=>
  < B : Butler | sticks: IS >
  msg Stick(I) from B to P .
```

When a given philosopher has finished eating, he returns his chopsticks to the butler. The butler, being the polite gentleman he is, replies with a *ThankYou* message to acknowledge that he has received the stick, as shown in the rules *finished-eating* and *get-returned-stick* below.

```
rl [finished-eating] :
  < P : Philosopher | state: eating, leftStick: yes,
    rightStick: yes, butler: B, seat: I, eatcount: C >
=>
  < P : Philosopher | state: thinking, leftStick: no,
    rightStick: no, butler: B, seat: I, eatcount: C >
  (msg Stick(I) from P to B)
  (msg Stick(if I == 5 then 1 else I + 1 fi) from P to B) .


rl [get-returned-stick] :
  < B : Butler | sticks: IS >
  msg Stick(I) from P to B
=>
  < B : Butler | sticks: IS I >
  msg ThankYou from B to P .
```

So far, all is well. However, if we try to run this specification in Maude using the fair rewrite command *frew*, we end up in a deadlock, that is, a configuration in which each of the philosophers hold on to exactly one chopstick, as shown below. (Note that we end up in a deadlock because of the particular execution strategy that Maude's *frew* command makes use of.)

```
Maude> frew init .

frewrite in DINING-PHILOSOPHERS : init .
rewrites: 189 in 0ms cpu (0ms real) (~ rewrites/second)
result Configuration:

< 'James : Butler | sticks: none >
(msg RequestStick(1) from 'Socrates to 'James)
(msg RequestStick(2) from 'Plato to 'James)
(msg RequestStick(3) from 'Aristotle to 'James)
(msg RequestStick(4) from 'Anaximander to 'James)
(msg RequestStick(5) from 'Pythagoras to 'James)
```

```
< 'Socrates : Philosopher | state: hungry,leftStick:
  requested,rightStick: yes,butler: 'James,seat: 1 >
< 'Plato : Philosopher | state: hungry,leftStick:
  requested,rightStick: yes,butler: 'James,seat: 2 >
< 'Aristotle : Philosopher | state: hungry,leftStick:
  requested,rightStick: yes,butler: 'James,seat: 3 >
< 'Anaximander : Philosopher | state: hungry,leftStick:
  requested,rightStick: yes,butler: 'James,seat: 4 >
< 'Pythagoras :Philosopher | state: hungry,leftStick:
  requested,rightStick: yes,butler: 'James,seat: 5 >
```

As we can see, Socrates, Plato, Aristotle, Anaximander and Pythagoras all cling to their right hand chopstick, and hope that they will soon receive the left one from James (they have all sent a *RequestStick* message). Sadly, as none of them will ever let their right chopstick go, James has no more sticks to hand out, and they will all starve to death.

So, the question arises: Can we, with the aid of our predicate enabled meta-rewriting engine, prevent this from ever happening?

The answer to this question is affirmative. We observe that if we are to avoid a deadlock, we only need to look at the local history for the butler James. If he only has one chopstick left, he cannot send this chopstick to a philosopher that does not already possess a stick. However, if this simple invariant is not broken, there will never be a deadlock in this specification.

Expressed with the predicate constructors from Chapter 5, the invariant will look as follows:[1]

```
if length(H / from('James) / msgtype('Stick)) minus
   length(H / from('James) / msgtype('ThankYou)) eq 5
then
   (length(H / from('James) / msgtype('Stick) /
      to('Pythagoras)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Pythagoras)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Anaximander)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Anaximander)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Aristotle)) minus
```

[1] At this point, we will simply ignore the parameter in the *RequestStick* and *Stick* messages at the meta-level, since they are not of any importance to our invariant. Parameters in predicates will be introduced in Chapter 8.

```
    length(H / from('James) / msgtype('ThankYou) /
      to('Aristotle)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Plato)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Plato)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Socrates)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Socrates)) eq 2)
else
   TRUE
fi
```

The first thing that we note about this predicate is that we are only looking at the local history of the butler, as mentioned above. This is accomplished through a projection that spans only messages originating from this object.

The number of sticks that the butler has left can be found from his local history by subtracting the number of sticks that has been returned to him (given by *length(H / from('James) / msgtype('ThankYou))*) from the number of messages he has sent (given by *length(H / from('James) / msgtype('Stick))*).

In the same way, we can find how many sticks a given philosopher possesses at the moment, by subtracting the number of returned sticks from the number of received sticks.

84

Chapter 8

Extensions to the Predicate Framework

In this chapter we will look at some additional mechanisms for specifying and using predicates on finite communication histories that we have not covered in the preceding chapters.

In the first section, we will consider how to implement quantifiers to increase the expressiveness of our predicates.

In Section 8.2, we take a look at our rewrite strategy from the perspective of making it work as well as possible with the Creol interpreter, allowing us to specify predicates for Creol specifications as well as standard Maude specifications.

In Section 8.3, we consider an alternative approach for executing a specification in concordance with a predicate. Instead of enforcing an execution path that is in compliance with the predicate, we will in this section look at how we can stop the execution and alert the user when the predicate is violated.

Finally, in Section 8.4, parameters in messages and variables in predicate specifications are treated.

8.1 Quantifiers

In the previous example with the dining philosophers in Section 7.2, we saw that in order to express the desired invariant (if the butler James has given away all his chopsticks, then at least one of the philosophers must have received two of them) with the constructs we have introduced this far, we had to write quite a few lines of code:

```
if length(H / from('James) / msgtype('Stick)) minus
   length(H / from('James) / msgtype('ThankYou)) eq 5
then
```

```
   (length(H / from('James) / msgtype('Stick) /
      to('Pythagoras)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Pythagoras)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Anaximander)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Anaximander)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Aristotle)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Aristotle)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Plato)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Plato)) eq 2)
   or
   (length(H / from('James) / msgtype('Stick) /
      to('Socrates)) minus
    length(H / from('James) / msgtype('ThankYou) /
      to('Socrates)) eq 2)
else
   TRUE
fi
```

From this predicate, we notice that, not considering the individual philosophers' names, there are five sub-predicates stating exactly the same thing for the different philosophers. This kind of redundancy is clearly not desirable.

In order to get around this problem, we need to increase the expressiveness of our predicates in such a way that we can refer to a set of objects; we introduce the quantifiers $\forall$ and $\exists$. Having access to these operators, we can express the sub-predicates above in a shorter and more readable form. The variable P in the predicate below is implicitly defined to hold object identifiers:

$$\exists\ P : (\mathit{length}(H\ /\ \mathit{from}(\text{'}\mathit{James})\ /\ \mathit{msgtype}(\text{'}\mathit{Stick})\ /\ \mathit{to}(P))\ - \\ \mathit{length}(H\ /\ \mathit{from}(\text{'}\mathit{James})\ /\ \mathit{msgtype}(\text{'}\mathit{ThankYou})\ /\ \mathit{to}(P))\ =\ 2)$$

Looking at this, it seems like a pretty straightforward feature to implement in our meta engine, but there are some problems that need to be addressed:

First and foremost, we need some way of knowing which objects we have in our configuration at any given time in order to be able to check if the

history of one or all of them satisfy a given condition. One way of achieving this, is to let the end user hold this responsibility. In other words, the programmer specifying a predicate that includes a quantifier, will also be responsible for supplying the set of objects over which the quantifier ranges. Looking once more at our example from above, we could specify this in the following manner:

$$\exists\ P \in \{'Socrates,' Pythagoras,' Anaximander,' Aristotle,' Plato\} :$$
$$(length(H\ /\ from('James)\ /\ msgtype('Stick)\ /\ to(P))\ -$$
$$length(H\ /\ from('James)\ /\ msgtype('ThankYou)\ /\ to(P))\ =\ 2)$$

This approach has an important advantage in that it allows the programmer to specify a subset of the objects in a given configuration. An obvious drawback is of course that in a dynamic system, most of the time one does not know which objects are instantiated at any given time. To deal with this drawback, another approach allowing us to specify the set of objects that are "alive" at any point during an execution is needed.

In order to implement this, we need some way for our meta engine to know which objects are present in a configuration. One way to solve this would be to look at the configuration itself, and use a combined counting and pattern matching scheme to find the individual object identifiers. We prefer, however, to analyze the communication history instead:

Since our predicates all revolve around the communication history of an execution, all we need to do in order to know which objects that are in existence is to look at all the object identifiers that appear in either the *from* or the *to* field of any message in the history. This will not necessarily give us *every* object in the configuration, but with respect to the predicates, objects that have not sent any messages yet are of no interest to us.[1]

Moving on to the actual Maude implementation, we start by introducing two new constructors in our *PRED* module, corresponding to the first implementation choice discussed above (the user explicitly states the set of object identifiers):

```
op forAll_elementOf_|_  : ObjectVariable QidList Pred ->
   Pred [ctor] .
op exists_elementOf_|_  : ObjectVariable QidList Pred ->
   Pred [ctor] .
```

[1]In this discussion, we have gently skipped the problem that an object might be removed from a configuration, and still have messages originating from it or having it as their destination in the global history. This problem can be resolved by assuming that objects are sent a *delete* message from their creator when they are destroyed, or that the objects themselves send a *"going down"* message to their creator or to the class from which they were created. Deleted objects can hence easily be excluded from the set of objects over which a quantifier ranges.

Using these constructions, we can now specify our predicate in pure Maude syntax:[2]

```
exists p elementOf
  ('Pythagoras 'Anaximander 'Aristotle 'Plato 'Socrates) |
  (length(H / from('James) / msgtype('Stick) / to(p)) minus
   length(H / from('James) / msgtype('ThankYou) / to(p)) eq 2)
```

As the observant reader recalls, the function *CheckPredicate : Pred* $\times$ *MsgList* $\rightarrow$ *Bool* does the actual work involved in parsing the predicate and checking whether the actual communication history (represented as a message list of sort *MsgList*) is in compliance with this predicate.

Hence, we need to introduce some new equations for this function in order to make the quantifiers work. First, we observe that checking a condition for all objects in the set over which the universal quantifier ($\forall$) ranges, is the same as checking the condition for each of the objects recursively, and joining the results with a boolean *and* operator. Similarly, for the existence quantifier ($\exists$), we check recursively and join the results with an *or* operator.

Second, we note that in order to check the predicates against an actual list of messages (that is, our communication history), we need to substitute the variable for the real object id in each recursive call to the *CheckPredicate* function. Therefore, we need to change the signature of *CheckPredicate* so that it can take a third parameter, a substitution list: *CheckPredicate : Pred* $\times$ *MsgList* $\times$ *SubstitutionList* $\rightarrow$ *Bool*. The substitution list will consist of elements of the form

```
(subst X with Q)
```

where X is of sort *ObjectVariable*, and Q is of sort *Qid* (which is the sort used for object identifiers).

Having established the new signature for *CheckPredicate*, and a data structure for substitutions, we can define the additional equations for dealing with universally quantified predicates as follows:

```
eq CheckPredicate(forAll X elementOf (Q1 QL) | P1, ML, SL) =
   CheckPredicate(P1, ML, (SL, subst X with Q1)) and
   CheckPredicate(forAll X elementOf (QL) | P1, ML, SL) .
eq CheckPredicate(forAll X elementOf (nil) | P1, ML, SL) =
    true .
```

and similarly, for existentially quantified predicates:

```
eq CheckPredicate(exists X elementOf (Q1 QL) | P1, ML, SL) =
   CheckPredicate(P1, ML, (SL, subst X with Q1)) or
   CheckPredicate(exists X elementOf (QL) | P1, ML, SL) .
eq CheckPredicate(exists X elementOf (nil) | P1, ML, SL) =
    false .
```

[2]The variable p must be specified in the module as a constant of sort *ObjectVariable*.

The actual substitutions of the variable X for a real object identifier will occur at the deepest recursive level at which the granularity of the predicate expression being processed is at its smallest, as is the case in the equations for the projections, e.g.:

```
var OV1 : ObjectVariable . var M : Msg . var Q1 : Qid .
var SUBST : Subst . var SL : SubstitutionList .

eq Project(M, to(OV1), SL) = to(Subst(OV1, SL), M) .

op Subst : ObjectVariable SubstitutionList -> Qid .
eq Subst(OV1, (subst OV1 with Q1), SL) = Q1 .
eq Subst(OV1, (SUBST, SL)) = Subst(OV1, SL) [otherwise] .
```

The reason for performing the substitutions at the deepest recursive level instead of performing them right away in the equation for *CheckPredicate* above is two-fold:

- If we were to perform them right away, we would have to write another recursive function to do this, and that would add unnecessary complexity to our solution and hence further complicate the implementation.
- Secondly, to be able to apply the substitutions, one will either way have to process the predicate recursively to apply the substitution at the required places in the predicate expression. Doing a separate recursive "traversal" will of course affect the efficiency of the solution negatively, as opposed to performing the substitutions during the same traversal in which we check the predicate and expand the projections.

As we have now developed the constructs necessary to make the quantifiers work for a given explicitly specified set of object identities, let us return to the problem of making them work for the set of all objects that are in existence in a given configuration at a given point.

We start with the easiest part first; we specify the signatures in our *PRED* module with Maude syntax:

```
op forAll_|_ : ObjectVariable Pred -> Pred [ctor] .
op exists_|_ : ObjectVariable Pred -> Pred [ctor] .
```

As we see, these are identical to the signatures used when we were dealing with an explicit set of object identifiers, except for the fact that the third parameter, the set of object identifiers, is of course not present anymore.

Implementing the functionality for these operators will actually be quite easy now, as we are able to make use of our existing functionality to make this work. All we have to do, is to calculate the set of object identifiers

present in the system at the time the check is to be made, and then do a call to the function we defined above that dealt with explicit object identities.

Hence, the equation for the $\forall$ quantifier will be as follows:

```
eq CheckPredicate(forAll X | P1, ML, SL) =
   CheckPredicate(forAll X elementOf (GetObjectIDs(ML)) |
    P1, ML, SL) .
```

Correspondingly, for the $\exists$ quantifier:

```
eq CheckPredicate(exists X | P1, ML, SL) =
   CheckPredicate(exists X elementOf (GetObjectIDs(ML)) |
    P1, ML, SL)
```

The function *GetObjectIDs : MsgList* $\rightarrow$ *QidList* will return the identifiers of the objects present in the history (represented by the variable *ML* in the equations above) at the time of the call. ("Present in the history" is to be understood as occurring either in the *to* field or the *from* field of a message in the history as recorded at runtime.)

So, using the constructs defined in this section, we can now specify the dining philosophers invariant by means of the following Maude syntax:

```
if length(H / from('James) / msgtype('Stick)) minus
   length(H / from('James) / msgtype('ThankYou)) eq 5
then
   exists p |
    (length(H / from('James) / msgtype('Stick) / to(p)) minus
     length(H / from('James) / msgtype('ThankYou) / to(p)) eq 2)
else
   TRUE
fi
```

As we can see, we have now expressed the same invariant that we had at the beginning of this section, in a syntax that is considerably easier to both read and write.

8.2 Adapting the framework for use with the Creol interpreter

Much of the motivation for writing the meta-level rewrite strategy and predicate checker introduced in this thesis, was to be able to execute the Creol interpreter with predicate checking at the meta-level. All the concepts introduced this far, applies in this scenario as well. However, there are some rather subtle points that need to be dealt with that are specific for Creol and the Creol interpreter.

Note that in the following, we show how the meta-level rewrite strategies that we have defined in the previous chapters can be used with the interpreter as implemented in [3]. However, since the interpreter is under continuous development, some changes may need to be done in order to make it work with the version that is current at the time of reading.

8.2.1 Creol's message format

Creol method calls are syntactically similar to standard object oriented method calls, and are given in the form

$$o.m(in : p_1, ..., p_n \; out : q_1, ...q_n)$$

where o is an object, m is a method and $p_1, ...p_n$ and $q_1, ...q_n$ are (possibly empty) lists of in and out parameters, respectively. The messages we have looked at this far in this thesis, have been defined in a way conventional for Maude specifications:

$$msg_ \; from_ \; to_ \; .$$

Messages emitted from the Creol interpreter, on the other hand, look at bit different. First of all, there are two main categories of messages, invocation messages and completion messages. An invocation results in a method call. A completion message is the manner in which the result of an asynchronous invocation is returned from the callee to the caller. In the interpreter from [3], the signature of an invocation message is defined as follows:

```
op invoc(_,_,_,_,_) : Nat Oid Oid Qid List -> Msg [ctor] .
```

The first parameter is a natural number which is used as a *label* for this call, and provides together with the sender's identifier, a unique identifier for any given call. The second and third parameters are the sender and receiver, respectively, the fourth is the name of the method that is to be invoked, and the fifth is a list of parameters to the method.

At the completion of a method, a *comp* message is emitted. It has the following signature:

```
op comp(_,_,_) : Nat Oid List -> Msg [ctor] .
```

As we can see, the parameter list is shorter for the completion message. We still have the label, the receiver (which in this case equals the original caller emitting the corresponding *invoc* message) and a list of return values. What we do not have, however, is the sender and the name of the method that was called. This complicates things a bit, as we shall see.

Since we want our meta-level strategy to work both for standard Maude specifications as well as Creol programs executed on the interpreter, we will store the Creol invocation and completion messages in the communication history in a form that resembles the standard Maude way.

First, we define two operators which allow us to differentiate between invocation and completion messages:

```
op invoc_ : Msg -> Msg .
op comp_ : Msg -> Msg .
```

Furthermore, we define a message that consists of a quoted identifier representing the name of the method that is called, and a list of parameters:

```
op _(_) : Qid List -> Msg .
```

Finally, we extend the standard message definition to include a label for Creol messages:

```
op msg_from_to_label_ : Msg Oid Oid GroundTerm -> Msg .
```

It may seem odd that the label is defined as a *GroundTerm*. However, since our message processing will happen at the meta-level, the labels in the original Creol messages will already be in their meta-level form as ground terms, and since the only reason for having the label from our point of view is to use it as part of a unique identifier, the meta-level representation of the term is just as suitable as the object-level one.

A Creol message in the communication history will be of the following form:

$$msg\ \{invoc|comp\}\ M(p_1, p_2, ..., p_n)\ from\ A\ to\ B\ label\ L$$

where M is a quoted identifier, $(p_1, p_2, ..., p_n)$ is a parameter list as defined in [3], A and B are object identifiers and L is a ground term.

8.2.2 Completion messages

The completion messages emitted in the Creol interpreter lack, as mentioned above, both sender and method name. In the interpreter this is not a problem, since the receiver and label uniquely identifies the message. In the communication history, however, we want all messages to be in the format specified above, for consistency and ease of specification. Hence, we need to compensate for the information that is lacking in the completion messages. Realizing that a completion message must necessarily come after the corresponding invocation message,[3] we see that all the information we need to

[3]In the interpreter as defined in [3], this is a truth with some modifications, as we shall see a bit later on.

"fill in the blanks" for these messages is already stored in the communication history, all we have to do is to retrieve it.

When a completion message is emitted from the interpreter, the meta-level strategy calls a function *GetMsgMethod* : *GroundTerm* × *Oid* × *MsgList* → *Qid*. This function recursively backtracks through the communication history, passed in as a message list, until it finds an invocation message with a matching label and sender, and returns the name of the method, which can then be used at the meta-level to fill in the method name for the completion message. The same technique is used for finding the sender of the completion message.

Another problem is that in the interpreter, completion messages are sent for locals calls (since local calls may be asynchronous) even though no invocation message is emitted into the configuration (the invocation is handled directly by the interpreter without sending any messages). This leads to a slight inconsistency in the history, since there will be more completion messages than invocation messages. Furthermore, we have no way of knowing what method was called, since completion messages contain no such information, and we have no corresponding invocation message in the history with which we can compare label and sender. Hence, we have no choice but to label the method as *'localCall*, or to disregard it in the communication history (which in a way makes sense, since this is a *local* (or internal) call, which is not directly observable from the outside).[4]

8.2.3 An example: The dining philosophers in Creol

This example is based on the example from the article *A Run-Time Environment for Concurrent Objects with Asynchronous Methods Calls* [24], which is included in its entirety in Appendix B.

A variant of the dining philosophers example is now considered in Creol, in which the butler informs a philosopher of the identity of its left neighbor. A philosopher may borrow and return its neighbor's chopstick. Interaction between the philosophers and the butler is restricted by Creol interfaces:

```
interface Phil
begin
  with Phil
    op borrowStick
    op returnStick
end
```

```
interface Butler
begin
  with Phil
    op getNeighbor(out n:Phil)
end
```

In this approach, each philosopher controls one chopstick and must borrow its neighbor's chopstick in order to eat (as opposed to the example in 7.2,

[4]In the latest release of the Creol interpreter at the time of writing, this problem has been fixed, and local synchronous calls are implemented by message passing both for the invocation and completion of a method [26].

in which the butler controlled all the chopsticks). Thus, philosophers have their internal activity as well as responding to calls from the environment.

The philosophers are active objects, so the Philosopher class will include a *run* method, which is defined in terms of several non-terminating internal methods representing different activities within a philosopher; *think*, *eat*, and *digest*. In *run*, the internal methods are invoked asynchronously. All three methods depend on the value of the internal variable *hungry*. The *think* method is a loop which suspends its own evaluation before each iteration, whereas *eat* attempts to grab the object's and the neighbor's chopsticks in order to satisfy the philosopher's hunger. The philosopher has to wait until both chopsticks are available. In order to avoid blocking the object processor, the *eat* method is therefore suspended after asking for the neighbor's chopstick; further processing of the method can happen once the guard is satisfied. The *digest* method represents the action of becoming hungry. The Philosopher class is defined as follows:

```
class Philosopher(butler: Butler) implements Phil
begin
  var hungry: bool, chopstick: bool, neighbor: Phil
  op init == chopstick := true; hungry := false; butler.getNeighbor(;neighbor) .
  op run == true ⟶ !think ||| true ⟶ !eat ||| true ⟶ !digest .
  op think == not hungry ⟶ <thinking...>; wait ⟶ !think .
  op eat == var l : label; hungry ⟶ l!neighbor.borrowStick;
                (chopstick ∧ l?()) ⟶ <eating...>; hungry := false;
                 !neighbor.returnStick; wait ⟶ !eat .
  op digest == not hungry ⟶ (hungry := true; wait ⟶ !digest) .

 with Phil
  op borrowStick == chopstick ⟶ chopstick := false .
  op returnStick == chopstick := true .
end
```

The code of the butler class is straightforward and omitted here.

Translating the Creol code to so-called *Creol Machine Code*, which is executable by the Creol interpreter in the Maude engine (and hence it can also be executed by our meta-level strategy), we have the following initial configuration (the configuration is limited to three philosophers instead of five for brevity):

```
eq init =
(new 'Butler ('Butler0.0))
< 'Butler : Cl |
Att: ('this : null), ('p1 : null), ('p2 : null), ('p3 : null),
('p4 : null), ('p5 : null),

Init: ('p1 := new 'Philosopher('this)) ;
      ('p2 := new 'Philosopher('this)) ;
      ('p3 := new 'Philosopher('this)), no,
```

```
Mtds:
< 'getNeighbor : Mtdname |
Latt: ('label : null), ('caller : null), ('n : null),
Code: (if ('caller = 'p1) th 'n := 'p2
el (if ('caller = 'p2) th 'n := 'p3
el 'n := 'p1 fi) fi) ;
(end ( 'n )) > ,
Ocnt: 0.0
>

< 'Philosopher : Cl |
   Att: ('butler : null), ('hungry : null),
        ('chopstick : null), ('neighbor : null),
        ('history : null),
   Init: ('chopstick := bool(true)) ; ('hungry := bool(false)) ;
         ('history := str("")) ;
         ('label ! 'butler . 'getNeighbor(nil)) ;
         ('label ? ('neighbor)) ; ('run(nil | nil)), ('label : null),

   Mtds: < 'think : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: not 'hungry --> ('history := ('history cat str("t"))) ;
                  wait --> ! 'think(nil) ; end(nil) > *
         < 'eat : Mtdname |
            Latt: ('label : null), ('caller : null), ('l : null),
            Code: 'hungry --> ('l ! 'neighbor . 'borrowStick(nil)) ;
                  ('chopstick & ('l G? (nil))) -->
                  ('history := ('history cat str("e"))) ;
                  ('hungry := bool(false)) ;
                  (! 'neighbor . 'returnStick(nil) ) ;
                  wait --> ! 'eat(nil) ; end(nil) > *
         < 'digest : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: wait --> ('hungry := bool(true)) ;
                  ('history := ('history cat str("c"))) ;
                  wait --> (! 'digest(nil)) ; end(nil) > *
         < 'borrowStick : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: 'chopstick --> ('chopstick := bool(false)) ;
                   end(nil) > *
         < 'returnStick : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: ('chopstick := bool(true)) ; end(nil) > *
< 'run : Mtdname | Latt: ('label : null), ('caller : null),
Code: ((bool(true) --> ! 'think(nil)) ||
         (bool(true) --> ! 'eat(nil)) ||
         (bool(true) --> ! 'digest(nil))) ; end(nil) >,
   Ocnt: 0.0
> .
```

To execute this with our strategy, we use the *start* equation (*CODE* is the name of the Maude module in which the Creol Machine Code resides):

```
rew [100] start('CODE, 'init.Configuration) .
```

Since this is a non-terminating specification, we restrict the number of rewrites to 100. Looking at the results, we see from the *History* object that several messages have been sent (the *Engine* object is ignored brevity):

```
History[h:
  (msg invoc ('getNeighbor(nil)) from 'Philosopher0.0 to
    'Butler0.0 label ('s_['0.Zero])) @
  (msg invoc ('getNeighbor(nil)) from 'Philosopher1.0 to
    'Butler0.0 label ('s_['0.Zero])) @
  (msg invoc ('getNeighbor(nil)) from 'Philosopher2.0 to
    'Butler0.0 label ('s_['0.Zero])) @
  (msg invoc ('getNeighbor(nil)) from 'Philosopher3.0 to
    'Butler0.0 label ('s_['0.Zero])) @
  (msg invoc ('getNeighbor(nil)) from 'Philosopher4.0 to
    'Butler0.0 label ('s_['0.Zero])) @
  msg comp ('getNeighbor(oid('Philosopher2.0))) from 'Butler0.0 to
    'Philosopher1.0 label ('s_['0.Zero]),
pred: TRUE]
```

We see that all the philosopher objects have requested their neighbor's identity from the butler, and that the butler has responded to one of them, Philosopher1.0, that his neighbor is Philosopher2.0. Also note that the Creol messages have been converted to a standard Maude form in the history, and that the completion message contains both method name, sender and receiver.

8.3 Component testing and abstract environments

In an environment with several objects communicating (through asynchronous message passing or some other form of communication), the behavior of a given object can be defined by an *assumption-guarantee* specification [28]. The assumption is a requirement on the behavior of the object's surrounding environment. The object's own invariant is guaranteed to hold as long as the assumption on the environment holds.

Both the assumption and the invariant can be specified in terms of observable behavior, in our case as predicates on different projections of the communication history.

In the previous chapters of this thesis, we have defined a rewrite strategy that, given that there is a selection of applicable rules, will force a specification to be executed in concordance with the predicate, and only terminate if there are no applicable rules. This approach can be used to model an environment that behaves according to a given assumption. If, in addition, the environment specification itself contains no synchronization code (meaning that the objects may send messages in an arbitrary order and at arbitrary

times), we have an *abstract* environment that is controlled only by the predicate. With an abstract environment, we may actually model several different environment behaviors just by changing the assumption predicate. Hence we may test a programmed component, that includes synchronization code as well as other code, against a variety of environments.

For testing a programmed component, however, we do not want to force the component to behave according to its invariant specification. Instead, we want to be alerted if it fails to behave in the specified way, given that the environment with which our component communicates behaves correctly.

Since the rewrite strategy implemented so far forces an execution path according to a given predicate, we need to make some changes. The idea is to allow a (sub-)predicate to be checked in one of two different modes:

- The *force* mode is the mode that all our predicates have been checked in previously, and in this mode the strategy will force an execution path that complies with the specified predicate.
- The *fail-stop* mode, on the other hand, will terminate the execution and indicate what went wrong if an object attempts to violate the predicate.

To implement this in our strategy, we start by introducing a sort *Mode*, and two constants of this sort, *force* and *fail-stop*. Furthermore, we define a constructor $Mode : Pred \times Mode \rightarrow Pred$, that we will use to specify in which mode a given sub-predicate should be checked during execution. This allows us to for example specify predicates of the following form:

```
Mode(P1, fail-stop) and Mode(P2, force)
```

where $P1$ and $P2$ represent sub-predicates. This means that in the same specification, different projections of the communication history can be checked in different modes. For example, the history of a given object X (obtained by using an appropriate projection on the global history) may be checked in *fail-stop* mode by predicate $P1$, whereas the other objects of the specification are forced to behave according to predicate $P2$ using the *force* mode.

To make this work, we need to take the different modes into account when checking the predicates. First, we once more extend the signature of the *CheckPredicate* function that we first introduced in Chapter 5 and extended with substitutions in Section 8.1. The new signature will allow for a mode to be passed to the function, that specifies in which of our two modes the message list passed as the second parameter should be checked against the predicate passed as the first parameter.

```
op CheckPredicate : Pred MsgList SubstitutionList Mode ->
   BoolAndModeTuple .
```

The return value is a tuple $(B\ ;\ M)$, where B is a boolean value indicating whether the predicate check was successful or not, and M is the mode in which the check was performed. To extract the different components from the tuple, we define two auxiliary functions:

```
op bool : BoolAndModeTuple -> Bool .
eq bool( (B ; MODE) ) = B .

op mode : BoolAndModeTuple -> Mode .
eq mode( (B ; MODE) ) = MODE .
```

Note that the *mode* function for extracting the mode part of the tuple is different from the *Mode* constructor (with a capital M) introduced above.

Let the variables $P1$ and $P2$ be predicates of sort *Pred*, *MODE* and *MODE2* constants of sort *Mode*, *ML* a communication history in the form of a message list and *SL* a (possibly empty) substitution list. The mode is then determined in the *CheckPredicate* function by adding an equation that checks for a *Mode* constructor, as shown below:

```
eq CheckPredicate(Mode(P1, MODE), ML, SL, MODE2) =
   CheckPredicate(P1, ML, SL, MODE) .
```

The actual predicates are then checked in much the same manner as defined in Chapter 5, for example the equation for the boolean *and* operator, looks as follows:

```
eq CheckPredicate(P1 and P2, ML, SL, MODE) =
   if bool(CheckPredicate(P1, ML, SL, MODE)) then
      CheckPredicate(P2, ML, SL, MODE)
   else
      (false ; mode(CheckPredicate(P1, ML, SL, MODE)))
   fi .
```

In contrast to the previous definition, a tuple is now returned from *Check-Predicate*, and the current mode is passed along in every recursive call.

The actual difference in behavior when using the *fail-stop* mode as opposed to the *force* mode that we have implicitly used up till now, is handled in the *exec-mode* conditional rewrite rule, as shown in Figure 8.1 on page 100. The code that is changed from the previous definition of the strategy is mainly in lines 23–31, in which the application of the *LABEL* rule has failed because the resulting communication history would violate the predicate *PRED*:

- In lines 23–25, we check if the mode is *force*. If this is the case, then the behavior is as before, we put the label that failed at the back of the *labels* list and add it to the *failedRules* list.

- If, on the other hand, the mode is *fail-stop*, our goal is to stop the execution and alert the user. This happens in line 30, where a new object *Fail* of sort *EngineObject* replaces the *Engine* object. The *Fail* object contains properties for the label of the rule that caused the failure, and the current configuration at the time the fail occurred. Because the *Engine* object is replaced, the execution will terminate, since the *exec-mode* rule is no longer applicable.

So, to summarize, our rewrite strategy is now able to check the communication history in one of two modes, one that will force (as far as possible) the execution of a specification to behave according to a predicate specification, and one that allows for testing whether a given specification conforms to a predicate on its communication history or not. It is also possible to check different parts of the communication history in different modes by applying appropriate projections. If no mode is specified for the predicate, the *force* mode is used.

8.3.1 An example: Abstract dining philosophers

The following example is based on the example in [4], which is included in Appendix B. The example will be specified as a standard Maude object specification.

We consider another variant of the dining philosophers problem. In this example, we will use one programmed philosopher object, that has synchronization rules and internal state, and several *abstract* philosopher objects that may send any message at arbitrary times.

There is no butler this time, so a philosopher will have to contact his right hand neighbor to borrow his stick when he is hungry. This is done by sending a *borrowStick* message. The neighbor may then reply with a *lendStick* message if he is willing to lend his chopstick to his neighbor. When a philosopher has finished eating, he returns the stick he borrowed from his neighbor by sending a *returnStick* message.

In addition to communicating with their neighbors, the philosophers may also think and eat. For a philosopher X, this is represented by sending a *think* or *eat* message from X to X, respectively.

For the philosopher objects, we define a predicate on the communication history for *acceptable behavior*. The philosophers may only eat when they are hungry, and think when they are fed. Furthermore, a philosopher may obviously only lend his stick to his neighbor when he is in possession of his own stick. These properties can be expressed for a philosopher X and a communication history ML by the following recursive equations:

```
eq hungry?(X, nil) = false .
eq hungry?(X, (msg M from X to Y) @ ML) = (M == 'borrowStick or
   M == 'eat) .
```

```
1.  crl [exec-mode] :
2.    Engine[curTerm: T, curModule: MOD, labels:
3.      LABEL LABELS, failedRules: FAILEDRULES]
4.    History[h: ML, pred: PRED]
5.  =>
6.    if metaXapply([MOD], T, LABEL, none, 0, 1, 0) == failure
7.    then
8.      Engine[curTerm: T, curModule: MOD, labels:
9.        LABELS LABEL, failedRules: FAILEDRULES LABEL]
10.     History[h: ML, pred: PRED]
11.   else
12.     if bool(CheckPredicate(PRED, ML @
13.       getNewMessages(T, getTerm(metaXapply([MOD],
14.       T, LABEL, none, 0, 1, 0))))) == true
15.     then
16.       Engine[curTerm: getTerm(metaXapply([MOD], T, LABEL,
17.         none, 0, 1, 0)), curModule: MOD, labels:
18.         LABELS LABEL, failedRules: nil]
19.       History[h: ML @
20.         getNewMessages(T, getTerm(metaXapply([MOD], T,
21.         LABEL, none, 0, 1, 0))), pred: PRED]
22.    else
23.       if mode(CheckPredicate(PRED, ML @
24.         getNewMessages(T, getTerm(metaXapply([MOD],
25.         T, LABEL, none, 0, 1, 0))))) == force
26.       then
27.         Engine[curTerm: T, curModule: MOD, labels:
28.           LABELS LABEL, failedRules: FAILEDRULES LABEL]
29.       else
30.         Fail[label: LABEL, state: T]
31.       fi
32.       History[h: ML, pred: PRED]
33.     fi
34.   fi
35. if length(FAILEDRULES) < length(LABEL LABELS) .
```

Figure 8.1: A rewrite strategy that can check predicates in two different modes, *force* and *fail-stop*

```
eq hungry?(X, (msg M from Y to Z) @ ML) =
   hungry?(X, ML) [otherwise] .

eq fed?(X, nil) = true .
eq fed?(X, (msg M from X to Y) @ ML) = (M == 'returnStick
   or M == 'think or M == 'lendStick) .
eq fed?(X, (msg M from Y to Z) @ ML) =
   fed?(X, ML) [otherwise] .

eq hasStick?(X, (msg 'lendStick from X to Y) @ ML) = false .
eq hasStick?(X, (msg 'returnStick from Y to X) @ ML) = true .
eq hasStick?(X, (msg M from Y to Z) @ ML) =
   hasStick?(X, ML) [otherwise] .
eq hasStick?(X, nil) = true .
```

A philosopher is hungry when he has requested his neighbor's chopstick, as well as when he is still eating. A philosopher is fed when he has returned a borrowed chopstick, or when he lends his stick to his neighbor. The *hasStick?* equation returns true when a philosopher's neighbor has returned the stick he borrowed, or if the philosopher has never lent his chopstick to his neighbor.

We can now specify the global $AccBeh : MsgList \rightarrow Bool$ predicate for acceptable philosopher behavior:

```
eq AccBeh(nil) = true .

eq AccBeh((msg 'think from X to Y) @ ML) =
   ML == nil or (fed?(X, ML) and AccBeh(ML)) .

eq AccBeh((msg 'eat from X to Y) @ ML) =
   (hungry?(X, ML) and AccBeh(ML)) .

eq AccBeh((msg 'borrowStick from X to Y) @ ML) =
   fed?(X, ML) and AccBeh(ML) .

eq AccBeh((msg 'returnStick from X to Y) @ ML) =
   hungry?(X, ML) and AccBeh(ML) .

eq AccBeh((msg 'lendStick from X to Y) @ ML) =
   hasStick?(X, ML) and AccBeh(ML) .
```

We see that the *AccBeh* predicate is defined by one equation for each message type. For example, the predicate specifies that if an *eat* message is to be allowed, the philosopher in question must be hungry, as defined by the *hungry?* equation.

Now that we have defined a predicate for acceptable behavior, we move on to looking at the actual philosopher objects in Maude. First, we consider a concrete (programmed) philosopher object, with internal state and synchronization rules.

```
op <_: Philosopher | state:_, myStick:_, nbrStick:_ ,
    leftNbr:_, rightNbr:_ > :
    Qid State StickState StickState Qid Qid -> Phil .
```

The state of the philosopher may be either *hungry* or *fed*, and the values for the *myStick* and *nbrStick* attributes may be either *yes*, *no* or *requested.* The identities of the left and right hand side neighbors are stored in the *leftNbr* and *rightNbr* attributes, respectively.

Let C and D be variables of sort *Qid*, which is used as identifiers for concrete philosopher objects. The behavior of concrete philosophers is specified by the following rewrite rules (attributes that are irrelevant for the respective rules are ignored in the style of Full-Maude [16]):

```
rl [think] :
  < C : Philosopher | state: fed >
=>
  < C : Philosopher | state : fed >
  msg 'think from C to C .

rl [eat] :
  < C : Philosopher | state: hungry, myStick: yes, nbrStick: yes >
=>
  < C : Philosopher | state: hungry, myStick: yes, nbrStick: yes >
  msg 'eat from C to C .

rl [requestStick] :
  < C : Philosopher | state: hungry, myStick: yes,
    nbrStick: no, rightNbr: D >
=>
  < C : Philosopher | state: hungry, myStick: yes,
    nbrStick: requested, rightNbr: D >
  msg 'borrowStick from C to D .

rl [returnStick] :
  < C : Philosopher | state: hungry, myStick: yes,
    nbrStick: yes, rightNbr: D >
=>
  < C : Philosopher | state: hungry, myStick: yes,
    nbrStick: no, rightNbr: D >
  msg 'returnStick from C to D .

rl [lendStick] :
  < C : Philosopher | state: fed, myStick: yes >
  msg 'requestStick from D to C
=>
  < C : Philosopher | state: fed, myStick: no >
```

```
  msg 'lendStick from C to D .

rl [recieveRequestedStick] :
  < C : Philosopher | nbrStick: requested, rightNbr: D >
  msg 'lendStick from D to C
=>
  < C : Philosopher | nbrStick: yes, rightNbr: D >  .

rl [recieveReturnedStick] :
  < C : Philosopher | myStick: no, rightNbr: D >
  msg 'returnStick from D to C
=>
  < C : Philosopher | myStick: yes, rightNbr: D >  .
```

We will now define an environment for the programmed philosopher in form of an *abstract* philosopher definition. The idea is to have no synchronization code, and rather let the behavior of the environment be completely defined by a predicate specification. The only attributes that we will keep is the neighbor attributes.

An abstract philosopher is defined as follows:

```
op <_: Philosopher | leftNbr:_, rightNbr:_ > :
   Aid Qid Qid -> Phil .
```

Let A and B be variables of sort *Qid*, which is used as identifiers for abstract philosopher objects. The behavior an abstract philosopher can be specified by the following rewrite rules:

```
rl [abs-think] :
  < A : Philosopher | >
=>
  < A : Philosopher | > msg 'think from A to A .

rl [abs-eat] :
  < A : Philosopher | >
=>
  < A : Philosopher | > msg 'eat from A to A .

rl [abs-requestStick] :
  < A : Philosopher | rightNbr: B >
=>
  < A : Philosopher | rightNbr: B >
  msg 'borrowStick from A to B .

rl [abs-returnStick] :
  < A : Philosopher | rightNbr: B >
=>
  < A : Philosopher | rightNbr: B >
  msg 'returnStick from A to B .

rl [abs-lendStick] :
```

```
  < A : Philosopher | >
  msg 'requestStick from B to C
=>
  < A : Philosopher | >
  msg 'lendStick from A to B .
```

These rules do not express any synchronization constraints on the interactions, only which philosophers may interact. Also note, that rules for receiving messages are no longer needed, since no internal state change takes place in the abstract philosopher objects. Instead, a simple consumption rule can be used to remove messages from the configuration:

```
rl [abs-consumeMsg] :
  < A : Philosopher | > msg M from A to B
=>
  < A : Philosopher | > .
```

Furthermore, by specifying rules for the creation and deletion of objects, we can define a *non-deterministically evolving* environment:

```
rl [create] :
  < A : Philosopher | rightNbr: B >
  < B : Philosopher | leftNbr: A >
=>
  < A : Philosopher | rightNbr: A + B >
  < A + B : Philosopher | leftNbr: A, rightNbr: B >
  < B : Philosopher | leftNbr: A + A > .

rl [destroy] :
  < A : Philosopher | rightNbr: C >
  < C : Philosopher | leftNbr: A, rightNbr: B >
  < B : Philosopher | leftNbr: C >
=>
  < A : Philosopher | rightNbr: B >
  < B : Philosopher | leftNbr: A > .
```

In the *create* rule, the new abstract philosopher object is inserted in-between two existing (abstract or concrete) philosopher objects. The new philosopher will have the concatenation of the existing objects' identifiers as its identifier. In the *destroy* rule, an abstract philosopher object in-between two other philosopher objects is deleted, and the remaining philosopher objects set their *leftNbr* and *rightNbr* properties accordingly.

The abstract specification may now be used as a *testbed* for a programmed philosopher object X. The behavior of the environment is specified using a projection on the history spanning every object but the programmed one, and checking this projection in *force* mode with the *AccBeh* predicate. The programmed object can then be *tested* by using a projection spanning only this object and checking this projection with the *AccBeh* predicate in the *fail-stop* mode:

```
P(X) =
  Mode(AccBeh(H / not from(X)), force)
    and
  Mode(AccBeh(H / (from(X) or to(X))), fail-stop)
```

In the first part of the predicate, the abstract environment is forced to behave according to the *AccBeh* predicate. In the second part (after *and*), the programmed philosopher object X is tested using the *fail-stop* mode. If X attempts any action that would violate the predicate specification even though the environment behaves correctly, the execution will be halted.

8.4 Parameters and variables in predicates

8.4.1 Parameters

The predicates that we have specified so far in this thesis, have been using simple messages without any parameters. However, as we briefly touched in on in Section 8.2.1, messages emitted from the Creol interpreter contain parameter lists. Also, as we saw in the dining philosopher's example from Section 7.2, standard Maude specifications may make use of messages with parameters. Even though the predicate specified to prevent deadlock in Section 7.2 did not need to consider the parameters in the messages (in fact, the parameters were simply ignored at the meta-level), it is not hard to imagine scenarios in which such capabilities would be vital in order to correctly specify a predicate for a given specification.

Even though a Maude message could contain a parameter of any sort, as long as we specify its constructor accordingly, we will restrict parameters in predicates to the data type *Data* (and subtypes), as defined for the Creol interpreter in [3]. This is done mainly for two reasons:

- By restricting all parameter values to one common super-type, we avoid having to define auxiliary functions and message constructors for a vast number of data types available in Maude (Maude does not natively support polymorphism in the form of a universal common super-type).
- By using the same data types as for Creol messages, we avoid having to define separate mechanisms when dealing with Maude and Creol specifications interchangeably.

Values of the *Data* type, are defined by wrapping constructors corresponding to the given type around the actual value. For example, an integer value 3 is represented as *int*(3), and a string value *"test"* is correspondingly represented as *str*(*"test"*). A message *TestMessage* from A to B with parameter values 1, *true* and *"test"*, would then be represented as

msg 'TestMessage(int(1), bool(true), str("test")) from 'A to 'B.

In Chapter 6, we introduced regular expressions for messages. Now that messages can contain parameters, we need to take this into consideration when checking the message list for compliance with the predicate. As we remember, checking whether a transition over a token T from a DFA-state S to a DFA-state S' was allowed, was done by comparing T with the next message M in the input communication history. If T and M were equal, the transition was allowed, M was removed from the input, and the current DFA-state was set to S'.

Since, at this point, the parameters are all actual values (e.g. $int(3)$), we can in fact use the same method as before when checking the regular expressions. A message with parameters from the communication history can be checked for equality with another message from the regular expression predicate specification by using Maude's built in $==$ operator. However, when the parameters in the predicate specification can contain variables or expressions, we must take a different approach, and this is the scope of the next section.

8.4.2 Variables

When specifying a predicate for a communication history that contains messages with parameters, it will often be the case that at the time of specification, we cannot know (or are not interested in knowing) the exact values of every parameter in every message.

In a safety specification, a typical example is that a given value must always be within a given bound, but as long as this bound is not exceeded, we do not want to specify the actual values in our predicate (as that would make the predicate way to restrictive and/or way to verbose). For example, one could imagine a power plant reporting temperature values from the core to an external observer through message passing, and that the parameter in these messages should never exceed a given value.

To be able to specify such predicates, we need to make some extensions to our regular expressions engine. First, we will list the requirements that we will have to meet:

1. We need a way of specifying an unknown value of a given type, so we must introduce variables.

2. The variables should be able to hold any value of the *Data* data type.

3. The variables must reside in a scope, and scopes should be nestable.

4. We need to be able to specify a boolean expression ranging over the defined variables, enabling us to for example enforce a bound value as mentioned above.

5. Our regular expressions engine must "understand" what a variable and a variable scope is, and be able to check predicate specifications that contain scoped variables.

The first requirement can be met by introducing a new sort, *DataVariable*, and constants x, y, z etc of this sort. (Here, x, y and z will be variable names. The user may define his or her own variables as constants of the sort *DataVariable*.)[5] Furthermore, we overload the definition of the *Data* constructors, so that a *Data* value can be constructed from either an actual value, such as an integer, or from a *DataVariable*. Hence, an integer variable can now be specified as $int(x)$.

For the second requirement, we need to devise a means of storing the values associated with each variable name. This will be done with a set of bindings from name to value. A binding is defined as follows:

```
op binding : Data Data -> Binding [ctor] .
```

The first *Data* parameter is the variable, and the second is the value. A binding of the value 4 to the variable name x would then be represented as $binding(int(x), int(4))$. A function $getBinding : Data \times BindingSet \rightarrow Data$ returns the value associated with a given variable from a given set of bindings.

A variable scope is defined by a *scope...endscope* construct of sort *Pattern*:

```
op scope(_)_endscope : List Pattern -> Pattern [ctor prec 53] .
```

The first parameter of sort *List* is a list of the variables that are included in the scope. The second parameter of sort *Pattern* is the regular pattern to which the scope applies, and since the sort of the *scope...endscope* constructor itself is also *Pattern*, scopes can easily be nested. A variable defined in an outer scope is accessible from any inner scopes. An example of a nested scope with regular patterns *P1* and *P2* and variables x, y and z is shown below:

```
scope(int(x), int(y)) P1 :: scope(int(z)) P2 endscope endscope
```

Note how the pattern concatenation operator :: is used in precisely the same way as in Section 6.2; it simply concatenates the two patterns *P1* and *scope(int(z)) P2 endscope*.

Requirement four states that we need a way to specify a "sub-predicate" inside the regular expressions, so that we for example can enforce an upper or lower bound for a given variable or expression. For this purpose, we introduce a *where* clause in our patterns:

[5] It may perhaps seem odd that we use *constants* for variables, but this is due to the fact that from Maude's point of view, they are just symbols with no inherent meaning, and we will keep track of the values they contain ourselves.

```
op _where_ : Pattern Expr -> Pattern [ctor prec 56] .
```

The *Expr* parameter is a boolean expression as defined in [3], and can be built from the following constructors:

```
ops not_ neg_ : Expr -> Expr .
ops _+_ _-_ _*_ _/_ _cat_ _%_ : Expr Expr -> Expr .
ops _<_ _<=_ _>_ _>=_ : Expr Expr -> Expr .
ops _and_ _or_ _/=_ _=_ : Expr Expr -> Expr .
```

As an example, we extend our previous example from above with a *where condition* on the variables x, y and z:

```
scope(int(x), int(y)) P1 :: scope(int(z)) P2
  where x + y < z endscope endscope
```

Note that the variables x and y defined in the outer scope are still in scope in the inner pattern $P2$, along with z.

Another thing about scopes that is worth noting, is the difference between the expressions

```
scope(int(x)) x * endscope       (1)
```

and

```
(scope(int(x)) x endscope) *     (2)
```

In expression (1), the value of x will be bound to the first value in the input, and for each subsequent symbol in the input, the value must be equal to x in order to be accepted by the regular expression. In other words, (1) specifies a string in which one specific integer is repeated zero or more times.

Expression (2), on the other hand, has the zero-or-more operator `*` (the Kleene star) on the outside of the scope. This means that the value of x will be rebound for every iteration. Hence, expression (2) specifies a string of arbitrary integers.

Moving on to number five in our list of requirements, we have come to the perhaps most challenging part of this section; to incorporate the other four requirements into the regular expressions engine from Chapter 6. To achieve this, there are again several issues that we need to consider:

1. How does the addition of scoping to the regular patterns affect our finite automata?

2. When and how are the variables bound to actual data values?

3. How do we check a given input message list against a regular expression specification that may contain both variables and expressions?

Considering issue number one from the enumeration above first, the fact is that we can no longer use the subset construction algorithm to create deterministic automata when we add scoping information to the regular expressions.[6] To see why, let us look at a small example. Consider the regular expression $int(n)^*$, where n is an integer. This expression specifies strings consisting of zero or more n's. In Figure 8.2 the corresponding non-deterministic finite automaton (NFA) is shown (note that marking of accepting states is ignored).

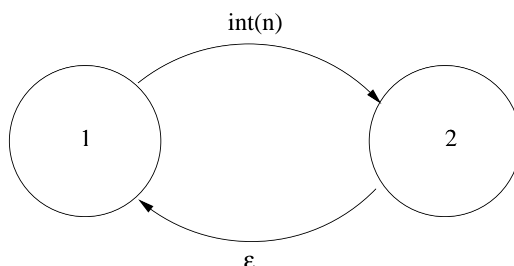

Figure 8.2: A non-deterministic finite automaton for the regular expression $int(n)^*$.

The transformation from this automaton to a deterministic one is straightforward, and the resulting DFA is shown in Figure 8.3.

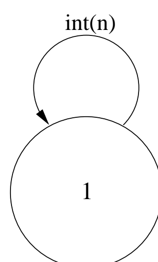

Figure 8.3: A deterministic finite automaton for the regular expression $int(n)^*$.

If we add scoping information to our example, and let the individual states contain information about which variables are in scope in an *inScope*

[6]Intuitively, it seems that a correct deterministic automaton cannot be built at all from a regular expression with variable scopes stored in the individual states, as introduced in this section. However, time prohibits a thorough investigation of this "hunch" in this thesis, so we will leave it an open issue for now.

property, it is not evident how to the transform the NFA to a corresponding DFA. Consider the example from above, with the integer number n exchanged for a scoped variable x: $scope(int(x))\ int(x)\ endscope\ ^*$. The NFA for this pattern (with scope information inside the states) is shown in Figure 8.4.

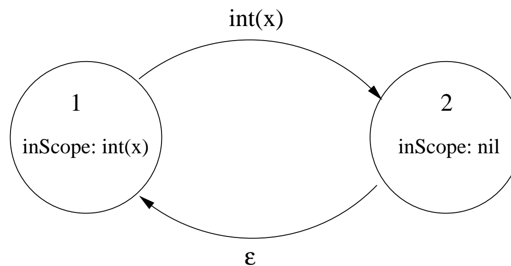

Figure 8.4: A non-deterministic finite automaton for the regular expression $scope(int(x))\ int(x)\ endscope\ ^*$.

If we try to transform this NFA to a DFA using the subset construction algorithm in the same way as we did in Figure 8.3 on the page before, what should be done with the *inScope* list of the remaining state? The answer is that we cannot correctly use the subset algorithm and at the same time preserve the correct scoping information inside the individual states when transforming an NFA to a DFA. With the small example shown in Figure 8.4, we could have manually created a correct DFA with scoping information, but for the much more complex general case, it is not at all evident how this should be done, or if it is at all possible. Hence, when it comes to regular expressions with scoped variables, we will use NFA matching instead of DFA matching to check whether a given input conforms to the specification. Note that a given NFA accepts the exact same language as the corresponding DFA.

Having decided to use NFA matching to check the regular expressions, we need to add some extra information to the NFA states originally defined in Section 6.3. We add two properties, one for the *InScope* list that will contain which variables that are in scope in a given state, and a *WhereCondition* property, that will allow a sub-predicate to be specified for the variables in the regular expressions. At first glance it may seem odd that the *WhereCondition* property is of sort *List*, but this is due to the fact that since regular expressions can contain sub-expressions, the where condition for a given state might actually be a list of where conditions. Our refined definition of an NFA state is as follows:

```
op {State:_, Accepting:_, Transitions:_, InScope:_,
  WhereCondition:_} : String Bool TransitionSet
  List List -> NFA-State .
```

Furthermore, we have to populate these properties when building an NFA from a given regular expression. This implies that we will have to make some changes to the *MakeNFA* equation originally introduced in Section 6.3 as well. The first thing we will do is to add two new parameters to its signature, as shown below:

```
op MakeNFA : Pattern String String List List -> NFA .
```

The first three parameters are as before, while the two last *List* parameters are the scope list and the list of where conditions, respectively. To begin with, both these parameters are nil, until a *scope...endscope* or *where* clause is encountered. In the case of *scope...endscope*, the parameter lists are populated by the following equation, where L, $L2$ and $W1$ are variables of sort *List*, $NAME$ and $NEXTSTATE$ are strings and $P1$ is a *Pattern*:

```
eq MakeNFA(scope( L ) P1 endscope, NAME, NEXTSTATE, L2, W1) =
  {State: NAME + "1", Accepting: false, Transitions:
    (epsilon -> NAME + "21"),
    InScope: L2, WhereCondition: W1
  }
  MakeNFA(P1, NAME + "2", NAME + "3", L L2, W1)
  {State: NAME + "3", Accepting: NEXTSTATE == "", Transitions:
    if NEXTSTATE =/= "" then
      (epsilon -> NEXTSTATE)
    else
      emptyTransitionSet
    fi,
   InScope: L2, WhereCondition: W1
  } .
```

As we can see, the new scope for the pattern $P1$ is the union of the previous scope $L2$ and the new scope L. In the state immediately after $P1$, the scope is reset to $L2$ again. In other words, in an inner scope, variables defined in the outer scope are also accessible.

Similarly, for the where clause, the *MakeNFA* equation goes as follows, in which E is a variable of sort *Expr*, and the other variables are as defined for the previous equation:

```
eq MakeNFA(P1 where E, NAME, NEXTSTATE, L, W1) =
  {State: NAME + "1", Accepting: false, Transitions:
    (epsilon -> NAME + "21"),
    InScope: L, WhereCondition: W1
  }
  MakeNFA(P1, NAME + "2", NAME + "3", L, W1 E)
  {State: NAME + "3", Accepting: NEXTSTATE == "", Transitions:
```

```
    if NEXTSTATE =/= "" then
      (epsilon -> NEXTSTATE)
    else
      emptyTransitionSet
    fi,
   InScope: L, WhereCondition: W1
 } .
```

Corresponding to the previous equation, the *where* clause for the pattern $P1$ equals the new condition E appended to the (possibly empty) list of existing conditions $W1$.

Having established the equations that build a non-deterministic automaton with scope and where condition information in the individual states, we can now move on to the problem of how we use this automaton to check the input. NFA checking is quite different from the DFA checking we did in Chapter 6. In an NFA

- ϵ-transitions are allowed in the automaton,
- a state can have several transitions originating from it over the same token, and
- we may have to back-track over several transitions and states during the checking when the path we have followed no longer matches the input.

Because of this, we will have to make some quite extensive changes to our *Match* equation. Perhaps the biggest challenge is to handle backtracking correctly. In order to achieve this, we will have to keep track of the states that we have already visited, and which transitions within these states that we have tried. Figures 8.5 on the next page and 8.6 on page 115 present a pseudo code outline of the *Match* equation, showing the base case in which the input message list is empty, and the general recursive case, respectively. (Since the full code for the *Match* equation is quite complex and would take up a couple of pages, we leave it for the interested reader to look up in the source code in Appendix A.)

In Figure 8.5 on the next page, the SS variable of sort *StateSet* is used to keep track of the states that we have already visited, and the BS variable is a *BindingSet* that contains a set of bindings from variable names to values of sort *Data*.

The *Match* function takes as its first parameter a token list, and checks if there exists a valid transition from the current state over the first token from the list. This functionality is implemented by the auxiliary *FindTransition* : *Token* $\times$ *TransitionSet* $\times$ *BindingSet* $\times$ *List* $\times$ *List* $\rightarrow$ *StateNameAndBindingSet* function. The *Token* parameter is a message from the communication

```
eq Match(nil, NFA, S, PRS, SS, BS) =
  if we use prs matching or S is an accepting state then
    return true
  else
    make a recursive call to Match with ϵ as the current token
  fi
```

Figure 8.5: A pseudo code outline of the base *Match* equation for NFA matching, for which the message list parameter is the empty list *nil*

history for which we try to find a transition to another state in the *TransitionSet*. The *BindingSet* is a list of bindings equal to the *BS* variable defined above, and the final two lists contain which variables that are in scope and any where conditions that the transition must conform to, respectively. The return value is a tuple consisting of the name of the state that the chosen transition leads to, or the empty string if no transition could be found, and a set of bindings that exists after the transition has been made.

The code for *FindTransition* goes as follows:

```
eq FindTransition(msg (Q1(PLIST1)) from FROM1 to TO1,
   ((msg (Q2 (PLIST2)) from FROM2 to TO2) -> S) TS, BS, L, W1)=
   if Q1 == Q2 and FROM1 == FROM2 and TO1 == TO2 and
      bool(ParameterCheck(PLIST1, PLIST2, BS, L)) and
      CheckCondition(W1, bindingSet(ParameterCheck(
        PLIST1, PLIST2, BS, L)))
   then
     ( S ; bindingSet(ParameterCheck(PLIST1, PLIST2, BS, L)) )
   else
     FindTransition(msg (Q1(PLIST1)) from FROM1 to TO1, TS,
       BS, L, W1)
   fi .
```

FindTransition first checks to see if the sender, receiver and message names match, and if so, makes use of the auxiliary *ParameterCheck* : *List* $\times$ *List* $\times$ *BindingSet* $\times$ *List* $\rightarrow$ *BoolAndBindingSet* function, that takes the parameter list from the message that we are checking from the communication history, the parameter list from the message in the regular expression, a set of variable bindings and a list of variables that are in scope and returns a tuple consisting of a boolean value and a set of bindings.

The parameter check is performed as follows, for each pair of parameters from the two lists:

- if the parameters are plain values, check if they are equal
- if the parameter from the regular expression is a variable, check if it is in scope:

- if it is not in scope, return false
- if it is in scope, check if it has already been bound to a value:
  * if it has been bound, check if the value from the communication history equals the value that has been bound, and return the result of the comparison
  * if it has not been bound, bind it now to the current value

If the boolean part of the returned tuple from *FindTransition* is *true*, the parameters are compatible, and the where conditions, if any, are checked by the *CheckCondition* function. If this function also returns true, the name of the new state to which the transition led is returned together with an updated set of bindings. Otherwise, the remaining transitions (if any) are checked. If no matching transition can be found, a tuple consisting of the empty string and the current binding set is returned. The *Match* equation will then have to backtrack and try to find another valid path through the NFA.

8.4.3 An example: The alternating bit protocol

The alternating bit protocol, also known as the stop-and-wait protocol, is a protocol for reliable transfer of data over an unreliable network. The sender will send one packet at the time, and wait for an acknowledgement from the receiver before sending the next packet. If no acknowledgement arrives (within a given time limit), the sender retransmits the packet. The receiver may also choose to resend an acknowledgment, if no data packet arrives. Since the sender has at most one unacknowledged packet in transit, a one bit sequence number may be used, hence the name alternating bit protocol.

The sender and reciever objects are modeled as follows:

```
op <_: Sender | Bit:_, Data:_> : Qid Int Int -> Sender .
op <_: Receiver | Bit:_> : Qid Int -> Receiver .
```

The sender has a *Bit* attribute for the sequence number, and a *Data* attribute containing the data that is to be sent. In our example, this is just an integer to which we will add 1 for every packet we send. The receiver has only one attribute, the sequence number *Bit*.

Our initial state consists of one sender, $'S$, and one receiver, $'R$, as shown below:

```
op init : -> Configuration .
eq init = < 'S : Sender | Bit: 0, Data: 0 >
          < 'R : Receiver | Bit: 0 > .
```

The sender sends *Data* messages to the receiver, containing two integers; a one bit sequence number B (represented in Maude by an integer with values 0 and 1), and an integer data value D:

```
eq Match(T1 @ TL, {State: S, Accepting: B, Transitions: TS,
   InScope: L, WhereCondition: W1} NFA, S, PRS, SS, BS) =
 if we have already visited the current state S, or
    S has no more transitions that we have not already
    tried (TS is empty) then
   return false
 else
   if the token T1 is not ϵ then
     if there is a valid transition over token T1 then
       make a recursive call to Match with TL and the
       state that T1 points to as the current state
       if the call succeeds then
         return true
       else
         make a recursive call with S as the current state,
         and with the transition we just tried removed
         from S (there might be several transitions over
         the same token in any given state)
       fi
     else
       there are no valid transitions over T1, we check for
       ϵ-transitions:
       if there is an ϵ-transition from S then
         make a recursive call with the state pointed to
         by the ϵ transition as the current state
         if this call succeeds then
           return true
         else
           remove the ϵ-transition we just tried
           from S, and make a recursive call with S
           as the current state
         fi
       else
         there are no ϵ-transitions either;
         return false
       fi
     fi
   else
     the token T1 is ϵ:
     if there is a transition over ϵ in TS then
       make a recursive call to Match with S as the
       current state
       if the call succeeds then
         return true
       else
         remove the transition we just tried,
         and make a recursive call to Match
       fi
     else
       there are no transitions over ϵ;
       return false
 fi fi fi
```

Figure 8.6: A pseudo code outline of the general recursive *Match* equation for NFA matching

```
msg 'Data(int(B), int(D))
```

Correspondingly, the receiver sends acknowledgments containing only a sequence number:

```
msg 'Ack(int(B))
```

The behavior of the sender is specified by the following rewrite rules for sending a packet, resending a packet and receiving an acknowledgment from the receiver, respectively:

```
rl [send-data] :
  < 'S : Sender | Bit: I, Data: D >
=>
  < 'S : Sender | Bit: (I + 1) rem 2, Data: D + 1 >
  msg 'Data(int((I + 1) rem 2), int(D + 1)) from 'S to 'R .

rl [resend-data] :
  < 'S : Sender | Bit: I, Data: D >
=>
  < 'S : Sender | Bit: I, Data: D >
  msg 'Data(int(I), int(D)) from 'S to 'R .

rl [recv-ack] :
   < 'S : Sender | Bit: I, Data: D >
   msg 'Ack(int(J)) from 'R to 'S
=>
   < 'S : Sender | Bit: I, Data: D > .
```

For each new packet that the sender sends, the data value is increased by one. Note that neither the data value nor the sequence number is changed when a packet is resent in the *resend-data* rule.

The actions of the receiver are specified as follows:

```
rl [recv-data] :
   < 'R : Receiver | Bit: I >
   msg 'Data(int(J), int(D)) from 'S to 'R
=>
   < 'R : Receiver | Bit: I > .

  rl [send-ack] :
  < 'R : Receiver | Bit: I >
=>
  < 'R : Receiver | Bit: (I + 1) rem 2  >
  msg 'Ack(int((I + 1) rem 2)) from 'R to 'S .
```

```
rl [resend-ack] :
  < 'R : Receiver | Bit: I >
=>
  < 'R : Receiver | Bit: I >
  msg 'Ack(int(I)) from 'R to 'S .
```

Note how both the sender and the receiver are under-specified. For example, the sender in this specification may very well send several packets without having received any acknowledgments from the receiver. Similarly, the receiver may send acknowledgments for packets it has never seen.

The underlying network is unreliable, and may loose arbitrary messages. This can be modeled by the following rewrite rules:

```
rl [loose-msg]:
   msg 'Data(int(J) int(D)) from 'S to 'R
=>
   none .

rl [loose-ack]:
   msg 'Ack(int(J)) from 'R to 'S
=>
   none .
```

An invariant for the alternating bit protocol is that the sender should never have more than one distinct unacknowledged message in transit. To conform to this invariant, we will specify a predicate for our example that makes use of variables.

First, we define a scoped regular expression for one legal message exchange between sender and receiver, called one *cycle*:

```
op cycle : -> Pattern .
eq cycle =
   scope(int(x) int(y))
     (msg 'Data(int(x), int(y)) from 'S to 'R) ::
      (
        (msg 'Data(int(x), int(y)) from 'S to 'R) |
        (msg 'Ack(int(x)) from 'R to 'S)
      ) * ::
     (msg 'Ack(int(x)) from 'R to 'S)
   endscope .
```

This regular expression states that in a valid cycle, there must first be a *Data* message, then zero or more *Data* and/or *Ack* messages, and finally an *Ack* message.

The variable x is the sequence number, and y is the data value. The variable scope will ensure that these values stay the same throughout the cycle once they are bound, and this will prevent the sender from sending more than one packet at the time, since the new packet would have a different sequence number (and a different data value).

Since more than one message exchange is allowed, we define a new expression *cycles*, that makes use of the *cycle* expression:

```
op cycles : -> Pattern .
eq cycles =
   ((cycle where int(x) = int(1)) ::
    (cycle where int(x) = int(0))) * .
```

We observe that in the *cycles* predicate, there are *where conditions* stating that the sequence number must first be 1, then 0, then 1 again etc. Since the where condition encapsulates the *cycle* sub-predicate, in which the scope for x is defined, the condition will initially be checked when x is bound inside a *cycle*.

The entire predicate P can then be defined by a prefix of the regular sequence *cycles*:

```
op P : -> Pred .
eq P = H prs cycles .
```

Below, we show some samples from the non-deterministic finite automaton that is created from the *cycles* regular expression. (Because the automaton is rather large, we only show parts of it.)

```
{State: "1",Accepting: false,
  Transitions: (epsilon -> "21") (epsilon -> "3"),
  InScope: nil,
  WhereCondition: nil}

{State: "21",Accepting: false,
  Transitions: epsilon -> "221",
  InScope: nil,
  WhereCondition: nil}

{State: "221",Accepting: false,
  Transitions: epsilon -> "2221",
  InScope: nil,
  WhereCondition: nil}

{State: "2221",Accepting: false,
  Transitions: epsilon -> "22221",
```

```
  InScope: nil,
  WhereCondition: int((x).DataVariable) = int(1)}

{State: "22221",Accepting: false,
  Transitions: epsilon -> "222221",
  InScope: int((x).DataVariable) int((y).DataVariable),
  WhereCondition: int((x).DataVariable) = int(1)}
```

The first thing we note in the listing above, is state 1, in which there are two transitions; one to the next state 21, and one to state 3, which is actually the last state of the entire automaton. The reason for this is that the outermost operator is the Kleene star `*`. Hence, an input list that is empty is valid.

Moving on to state 2221, we see that the where condition $x = 1$ has been set. The variable x is not yet in scope, which may seem a bit odd. This, however, is due to the fact that the states are constructed recursively, from outermost to innermost, and hence the where condition will be populated before the scope list. This is not a problem, since an ϵ-transition never will make use of any variables anyway.

In state 22221, we see that both the scope list and the where condition list are populated.

Below, we see the next state, 222221.

```
{State: "222221",Accepting: false,
  Transitions: msg 'Data(int((x).DataVariable),
    int((y).DataVariable)) from 'S to 'R -> "222222",
  InScope: int((x).DataVariable) int((y).DataVariable),
  WhereCondition: int((x).DataVariable) = int(1)}
```

In this state, there is an actual non-ϵ transition that includes the variable x. Since this is the first time that x is encountered, it will now be bound to the value in the recorded communication history, and then the where condition will be checked.

Finally, in the last state, named 3, we see that the scope and where condition lists are both empty. Furthermore, the state is accepting, and there is a transition back to the first state of the automaton to start another cycle:

```
{State: "3",Accepting: true,
  Transitions: epsilon -> "1",
  InScope: nil,
  WhereCondition: nil}
```

This example concludes our discussion about parameters and variables. To summarize, we have extended the predicates in such a way that messages

can contain parameters, and predicates can contain both variables, expressions and parameters. Adding variable scoping information to the individual NFA states, rendered the subset construction algorithm useless, and hence we had to devise a method for checking an NFA that can contain variables and expressions. NFA matching will in many cases be slower than DFA matching, due to backtracking and a larger automaton. Therefore, the DFA creation and matching algorithms from sections 6.4 and 6.5, respectively, are still useful for predicates without variables.

Chapter 9

Non-deterministic Execution

In all the previous chapters of this thesis, our meta rewriting strategy has chosen which rules to apply to the current term in a round robin manner. This has worked fairly well, however, there are some issues with this method that we will address in this chapter.

Since both Maude and hence also our engine are deterministic, two subsequent executions of a given specification from a given start term will, naturally, yield the exact same resulting term. Although this perhaps is fine for modeling sequential systems, our main focus is on concurrent objects that communicate through asynchronous message passing. In such a highly non-deterministic scenario, one would perhaps expect that two executions of the same system might yield different results or different execution paths, at least when considering the order in which messages are sent and received by the individual objects, for example due to external factors such as network latency etc. Furthermore, there are several other problems with both Maude's built-in deterministic strategies and our own strategy up till this point, perhaps most notably that the execution is unfair and more often than not yields a very skewed result.

In order to amend this situation, we have until now had to make changes to the actual module that models our problem domain. As an example, let us look again at the initial configuration in the dining philosophers example from Section 7.2 on page 78.

```
< 'Socrates : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 1, eatcount: 0 >
< 'Plato : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 2, eatcount: 0 >
< 'Aristotle : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 3, eatcount: 0 >
< 'Anaximander : Philosopher | state: thinking, leftStick: no,
  rightStick: no, butler: 'James, seat: 4, eatcount: 0 >
< 'Pythagoras : Philosopher | state: thinking, leftStick: no,
```

```
  rightStick: no, butler: 'James, seat: 5, eatcount: 0 >
< 'James : Butler | sticks: 1 2 3 4 5 > .
```

We recall that this initial configuration models the five philosophers Socrates, Plato, Aristotle, Anaximander and Pythagoras sitting around a table on which there is a bowl of food in the middle, and the butler James keeping track of the chopsticks needed by the philosophers in order to eat. Using this initial configuration to perform a given number of rewrites with our meta rewriting engine, the result will always be the same: Socrates will always be the only philosopher that eats (assuming that we perform enough rewrites to allow execution of the respective rewrite rules), and the others will starve.

In order to amend this in subsequent executions, we might try to change the order in which the philosophers are listed in the initial configuration. We might let, say, Pythagoras, be listed first instead of Socrates. Although this will yield a different result, the only difference will be that now Pythagoras will be the only one that gets to eat, and Socrates will starve with the rest of the philosophers.

As well as changing the order of the objects in the initial configuration, we might also try to change the order of the rewrite rules themselves within our specification. This may or may not, depending on the changes we make, affect the outcome of a following execution. However, these approaches seem unnecessarily clumsy, and still they do not address the fact that the rewrite rules will always be performed in the same order during an entire execution (i.e., rewrite rule A will always be applied, or at least checked for applicability, before rule B is attempted applied).

As mentioned in Chapter 2, Maude contains a so-called *fair* rewrite strategy, *frew*. This strategy is using a method known as position-fair rewriting (for more information on this, see [7]). While *frew* is clearly fairer than the strategy we have been using this far, the Maude Manual [11] states the following:

> Position-fair rewriting is not substitution fair; this is particularly apparent if you have a multiset of messages and objects.

Seeing as multisets of messages and objects are at the very core of our problem domain, some other non-manual, non-built-in method of fairness and randomization of the rule selection process is needed.

9.1 A Maude module for generating pseudo-random numbers

To address the problems discussed in the previous section, we will make use of a general purpose pseudo-random number generator. Since Maude, unlike many other popular programming languages, has no built-in access to

such a generator, we will have to implement one ourselves. In this section, we introduce a simple pseudo-random number generator based on the algorithm presented in *Numerical Recipes in C*, p. 278 [33]:

$I_{j+1} = a\ I_j(mod\ m)$

Furthermore, in [33] the authors argue that choosing

$a = 2^5$

and

$m = 2^{31} - 1$

will yield a generator that has passed all the important theoretical tests, and that has been put to successful use in a large number of real-world applications.

Translating this algorithm to Maude syntax, we introduce the module *RANDOM*, which will allow us to draw pseudo-random numbers in our meta-level rewrite strategy, as shown in Figure 9.1.

```
fmod RANDOM is
    protecting NAT .

    op rand : Nat -> Nat .

    op seed : -> Nat .
    eq seed = 1 . *** May be any positive integer

    ops a m : -> Nat .
    eq a = 16807 . *** = 2^5
    eq m = 2147483647 . *** = 2^31 - 1

    var N : Nat .
    eq rand(N) = (a * N) rem m .
endfm
```

Figure 9.1: A Maude module for generating pseudo-random numbers

In the module in this figure, the equation *rand* returns a pseudo-random natural number based on the value of the supplied natural number N. Obviously, for equal values of N the return value will always be the same. Hence, the previous return value from *rand* should be used as the input parameter when generating the next value. For the cases in which no input is given, the seed value specified by the constant *seed* may be used. Seeing as this algorithm is implemented as a separate functional module in Maude, it can easily be replaced with a more involved generator, should the need arise.

9.2 Pseudo-random selection of rewrite rules

In this section, we will use the module in Figure 9.1 on the preceding page to randomize the selection of rewrite rules from the *labels* list of the *Engine* object. Regarding the actual implementation, there are several approaches that are worth considering:

- We may randomly select one rule from the list of all rules, and then use this rule as a starting point for applying the rest of the rules in a FIFO manner. This will allow us to easily start an execution in different manners, but all the rules will still be applied in the original order during subsequent rewrites.
- On the other hand, upon every possible rule invocation, we may randomly select one rule from the list of rules in the *labels* list. This will result in an execution order that is totally randomized with respect to the original list of rule labels, but will introduce some extra overhead involving having to use the random number generator and some method for retrieving the randomly selected rule from the rule list based on the drawn number for each possible rule invocation.
- As a third alternative, we may combine the two approaches above. To begin with, we randomly select a rule from the list of rule labels, and apply it to the current configuration. Now, one of two things may happen; (1) the rule may be applied, resulting in a new configuration, or (2) the rule may fail, either due to the fact that it is not applicable with regards to the current configuration (Maude's *metaXapply* returns *failure*), or because applying it to the current configuration would result in a communication history that does not conform with the specified invariant.

  If the rule is applied, we randomly select a new rule. On the other hand, if the rule application fails, we try subsequent rules from the rule list in a round robin manner, until a rule is applied successfully. Then, we start over, and randomly select a new rule.

When the random rule-selection strategy was first implemented during the work with this thesis, option number three in the list above was chosen, to minimize the overhead involved with the randomization process. However, later experiments suggest that this overhead is negligible compared to the cost of maintaining the lists of rule labels and failed rules. Hence, we will choose option number two; upon every new rule application, the rule is drawn randomly using the *RANDOM* module.

As with any random number generator, our generator needs a so-called seed, in other words, an initial value. Once given, the entire sequence of random numbers is determined, hence the term *pseudo*-random number generator. Therefore, we need to let the user specify the initial seed. Alternatively,

we could have let the system clock supply the seed value, but Maude[1] does not allow access to it. Consequently, we have to change our signature for the *start* function to allow for a user-specified seed (the final integer parameter):

```
op start : Qid Term QidList Pred Int -> EngineConfig .
```

Furthermore, the rule rewrite rule in our strategy that performs the actual execution of a given specification, has now become quite a bit more complex. Nevertheless, since it is vital to the understanding of our solution, we will take a closer look at it, though in small pieces at the time. Let us first take a look at the left hand side of the rewrite rule, where there should be few surprises:

```
crl [exec-random-rules] :
   Engine[curTerm: T , curModule: MOD, labels: LABEL LABELS,
     failedRules: FAILEDRULES, numRules: NUMRULES,
     randomNum: RANDOMNUM]
   History[h: ML, pred: PRED]
=>
```

As we can see, there is a new natural number property in the engine; *randomNum*. This property allows us to hold the current random value, as subsequent random values depend on the previous value. The attribute *randomNum* will be initialized to the seed value.

Moving on to the right hand side of the rewrite rule, things are starting to get a bit more intricate, and we will split the right hand side into separate pieces that will be explained separately. For this purpose, we will first give a pseudo-code skeleton of the code. The individual parts that we will look at in detail below are enclosed within brackets, and are numbered from 1 to 4:

```
(1) { if <the selected rule cannot be applied> then }
(2)    { <add the selected rule to the list of failed rules>
         <draw a new random number> }
      else
(3)    { if <predicate check> == true then
           <update the current term>
           <update the history>
           <draw a new random number> }
         else
(4)        { <handle the failure> }
         fi
      fi
```

[1] as of version 2.1

The if test in part (1) checks whether the selected rule can be applied to the current configuration, meaning that Maude's *metaXapply* returns a term and not the constant *failure*. A given rule is selected randomly by using the remainder when dividing the pseudo-random number in *RANDOMNUM* by the number of rule labels in the module *MOD*. The *findItem* : *QidList* $\times$ *Nat* $\rightarrow$ *Qid* returns a given item, identified by its number, from a list of quoted identifiers:

```
  if metaXapply([MOD], T, findItem(LABELS,
    RANDOMNUM rem NUMRULES), none, 0, 1, 0) == failure
  then
```

If the test above returns *true*, the rule that was randomly selected could not be applied. Hence, in part (2), we need to add it to the list of failed rules, and draw another random number:

```
Engine[curTerm: T, curModule: MOD, labels:
  LABELS, failedRules:
    if findItem(LABELS, RANDOMNUM rem NUMRULES)
      in FAILEDRULES
    then
      FAILEDRULES
    else
      FAILEDRULES findItem(LABELS, RANDOMNUM
        rem NUMRULES)
    fi,
  numRules: NUMRULES, randomNum: rand(RANDOMNUM)]
History[h: ML, pred: PRED]
```

Since we are drawing rules from the entire *labels* list at random, the rule that has failed in the code fragment above, might already be in the *failedRules* list. Therefore, it is necessary to test for this situation before we add it to the list, which is done by using the *in* operator, which returns *true* if a given quoted identifier exists in a given list of quoted identifiers.

If, on the other hand, the if test in part (1) of the right hand side of the *exec-random* rewrite rule returns *false*, the selected rule is applicable to the current configuration. In part (3), we will then have to check whether the application of the rule will violate the predicate. If this is not the case, we update the history and the current term in the same way as before, only that now the rule label is of course chosen by using the random number in *RANDOMNUM*:

```
if bool(CheckPredicate(PRED, ML @
     getNewMessages(T, getTerm(metaXapply([MOD],
```

```
       T, findItem(LABELS, RANDOMNUM rem NUMRULES),
       none, 0, 1, 0)), MOD, ML))
   ) == true
then
  Engine[curTerm: getTerm(metaXapply([MOD], T,
    findItem(LABELS, RANDOMNUM rem NUMRULES),
    none, 0, 1, 0)), curModule: MOD, labels:
    LABELS, failedRules: nil, numRules: NUMRULES,
    randomNum: rand(RANDOMNUM)]
  History[h: ML @
    getNewMessages(T, getTerm(metaXapply([MOD], T,
    findItem(LABELS, RANDOMNUM rem NUMRULES), none, 0, 1, 0)),
    MOD, ML), pred: PRED]
```

Part (4) of the skeleton, vaguely described above as “handle the failure”, comes into play when the predicate is about to be violated by a given rule application. Depending on the chosen *mode* in which the predicate is to be checked, the strategy must act accordingly, as explained in Section 8.3. If the mode is *force*, the failure results in a new random number being drawn, and the label of the rule that failed is added to the *failedRules* list. On the other hand, if the mode is *force*, the execution is terminated, and the chosen rule label is inserted into the *Fail* object.

```
if mode(CheckPredicate(PRED, ML @
  getNewMessages(T, getTerm(metaXapply([MOD],
  T, findItem(LABELS, RANDOMNUM rem NUMRULES),
  none, 0, 1, 0)), MOD, ML))) == force
then
  Engine[curTerm: T, curModule: MOD, labels:
    LABELS,  failedRules:
      if findItem(LABELS, RANDOMNUM rem NUMRULES)
        in FAILEDRULES
      then
        FAILEDRULES
      else
        FAILEDRULES findItem(LABELS, RANDOMNUM
          rem NUMRULES)
      fi,
    numRules: NUMRULES, randomNum: rand(RANDOMNUM)]
else
  Fail[label: findItem(LABELS, RANDOMNUM rem NUMRULES),
        state: T]
fi
History[h: ML, pred: PRED]
```

To summarize, we are now able to execute a specification in such a way that the rewrite rules are selected pseudo-randomly at run-time, instead of having the order statically defined by the order in which they appear in the specification. The user may specify the initial seed value to easily produce different executions. The entire rewrite strategy is shown in the *exec-random* rewrite rule in Figure 9.2 on the next page.

9.2.1 An example: The dining philosophers with randomized rule selection

Now that we have defined all the necessary functionality for pseudo-random selection of rewrite rules, it is time to look at an example. Once more, we turn to our beloved dining philosophers (as specified in the example in Section 7.2). We will look at two distinct executions, each consisting of 1000 rewrites. Using our newly introduced fifth parameter of the *start* function, we are now able to supply two different seeds for the random number generator. Apart from this last parameter, represented by *SEED* below, the two calls to the *start* function will be identical:

```
rew [1000] start('DINING-PHILOSOPHERS, 'init.Configuration,
  nil,
  (if length(H / from('James) / msgtype(Stick)) minus
     length(H / from('James) / msgtype(ThankYou)) eq 5
    then
       exists x |
         (length(H / from('James) / msgtype(Stick) / to(x))
         minus
          length(H / from('James) / msgtype(ThankYou) / to (x))
         eq 2)
    else
      TRUE
    fi), SEED
) .
```

When using $SEED = 1$, the execution yields the following result (the *History* object is omitted for the sake of brevity, and we focus only on the *Engine* object):

```
Engine[
 curTerm: '__[
  '<_:`Butler`|`sticks:_>[''James.Qid,'none.IntSet],

  'msg_from_to_['ThankYou.Msg,''James.Qid,
                ''Pythagoras.Qid],
  'msg_from_to_['Stick['s_['0.Zero]],''James.Qid,
```

```
crl [exec-random] :
  Engine[curTerm: T, curModule: MOD, labels:
    LABELS, failedRules: FAILEDRULES,
    numRules: NUMRULES, randomNum: RANDOMNUM]
  History[h: ML, pred: PRED]
=>
  if metaXapply([MOD], T, findItem(LABELS, RANDOMNUM rem NUMRULES),
    none, 0, 1, 0) == failure then
    Engine[curTerm: T, curModule: MOD, labels:
      LABELS, failedRules:
        if findItem(LABELS, RANDOMNUM rem NUMRULES) in FAILEDRULES
        then FAILEDRULES else
          FAILEDRULES findItem(LABELS, RANDOMNUM rem NUMRULES)
        fi,
      numRules: NUMRULES, randomNum: rand(RANDOMNUM)]
    History[h: ML, pred: PRED]
  else
    if bool(CheckPredicate(PRED, ML @
      getNewMessages(T, getTerm(metaXapply([MOD],
      T, findItem(LABELS, RANDOMNUM rem NUMRULES),
      none, 0, unbounded, 0)), MOD, ML))) == true
    then
      Engine[curTerm: getTerm(metaXapply([MOD], T,
        findItem(LABELS, RANDOMNUM rem NUMRULES),
        none, 0, 1, 0)), curModule: MOD, labels:
        LABELS, failedRules: nil, numRules: NUMRULES,
        randomNum: rand(RANDOMNUM)]
      History[h: ML @
        getNewMessages(T, getTerm(metaXapply([MOD], T,
        findItem(LABELS, RANDOMNUM rem NUMRULES),
        none, 0, unbounded, 0)), MOD, ML), pred: PRED]
    else
      if mode(CheckPredicate(PRED, ML @
        getNewMessages(T, getTerm(metaXapply([MOD],
        T, findItem(LABELS, RANDOMNUM rem NUMRULES),
        none, 0, 1, 0)), MOD, ML))) == force
      then
        Engine[curTerm: T, curModule: MOD, labels:
          LABELS,  failedRules:
            if findItem(LABELS, RANDOMNUM rem NUMRULES) in FAILEDRULES
            then FAILEDRULES else
              FAILEDRULES findItem(LABELS, RANDOMNUM rem NUMRULES)
            fi,
          numRules: NUMRULES, randomNum: rand(RANDOMNUM)]
      else
        Fail[label: findItem(LABELS, RANDOMNUM rem NUMRULES), state: T]
      fi
      History[h: ML, pred: PRED]
    fi
  fi
if length(FAILEDRULES) < length(LABELS) .
```

Figure 9.2: A rewrite strategy with randomized rule selection

```
                ''Socrates.Qid],
  'msg_from_to_['Stick['s_^5['0.Zero]],''James.Qid,
                ''Anaximander.Qid],
  'msg_from_to_['RequestStick['s_^2['0.Zero]],''Plato.Qid,
                ''James.Qid],
  'msg_from_to_['RequestStick['s_^3['0.Zero]],''Aristotle.Qid,
                ''James.Qid],
  'msg_from_to_['RequestStick['s_^4['0.Zero]],''Anaximander.Qid,
                ''James.Qid],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Anaximander.Qid,'hungry.State, 'requested.StickState,
       'requested.StickState,''James.Qid,'s_^4['0.Zero],
    's_^15['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Aristotle.Qid,'hungry.State,'requested.StickState,
       'yes.StickState,''James.Qid,'s_^3['0.Zero],'s_^2['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Plato.Qid,'hungry.State,'requested.StickState,
       'yes.StickState,''James.Qid,'s_^2['0.Zero],'s_^2['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Pythagoras.Qid,'hungry.State,'no.StickState,
       'no.StickState,''James.Qid,'s_^5['0.Zero],'s_^5['0.Zero]],'

  <_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
      butler:_`,seat:_`,eatcount:_>
    [''Socrates.Qid,'hungry.State,'requested.StickState,
    'yes.StickState,''James.Qid,'s_['0.Zero],'s_['0.Zero]]],

 curModule: 'DINING-PHILOSOPHERS, labels: 'getting-hungry
    'recv-right-stick 'recv-left-stick 'had-enough 'eat
    'req-left-stick 'accept-thanks 'req-right-stick
    'give-stick 'get-returned-stick, failedRules:
    'get-returned-stick, numRules: 10, randomNum: 2028841238
]
```

The current configuration is meta-represented in the property *curTerm* of the *Engine* object. From this property, we note that Anaximander has eaten 15 times, Aristotle 2 times, Plato 2 times, Pythagoras 5 times and Socrates 1 time.

Using *SEED* = 282475249 (corresponding to *rand(rand(1))*), we get the

following results:[2]

```
Engine[
 curTerm: '__[
  '<_:`Butler`|`sticks:_>[''James.Qid,'s_^4['0.Zero]],

  'msg_from_to_['ThankYou.Msg,''James.Qid,
               ''Anaximander.Qid],
  'msg_from_to_['Stick['s_^5['0.Zero]],''Anaximander.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_['0.Zero]],''Socrates.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_^2['0.Zero]],''Plato.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_^3['0.Zero]],''Plato.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_^4['0.Zero]],''Anaximander.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_^4['0.Zero]],''Aristotle.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_^5['0.Zero]],''Anaximander.Qid,
               ''James.Qid],
  'msg_from_to_['RequestStick['s_^5['0.Zero]],''Pythagoras.Qid,
               ''James.Qid],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Anaximander.Qid,'hungry.State,'requested.StickState,
       'requested.StickState,''James.Qid,'s_^4['0.Zero],
       's_^11['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Aristotle.Qid,'hungry.State,'yes.StickState,
       'requested.StickState,''James.Qid,'s_^3['0.Zero],
       's_^6['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Plato.Qid,'hungry.State,'requested.StickState,
      'requested.StickState,''James.Qid,'s_^2['0.Zero],
      's_^2['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
```

[2] The value *rand(rand(1))* was chosen in order to get some interesting and illustrative results; choosing for instance *SEED* = 2 will also produce a randomized execution that is different from the case where *SEED* = 1, but the differences are not as immediately evident as in this case.

```
    [''Pythagoras.Qid,'hungry.State,'requested.StickState,
       'yes.StickState,''James.Qid,'s_^5['0.Zero],
       's_^4['0.Zero]],

  '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
       butler:_`,seat:_`,eatcount:_>
    [''Socrates.Qid,'hungry.State,'requested.StickState,
       'yes.StickState,''James.Qid,'s_['0.Zero],
       's_['0.Zero]]],

 curModule: 'DINING-PHILOSOPHERS, labels:
    'accept-thanks 'had-enough 'eat 'give-stick 'req-left-stick
    'req-right-stick 'recv-left-stick 'recv-right-stick
    'get-returned-stick 'getting-hungry, failedRules:
    'getting-hungry, numRules: 10, randomNum: 843998877
]
```

From this execution, we see that Anaximander has eaten 11 times, Aristotle 6 times, Plato 2 times, Pythagoras 4 times and Socrates 1 time. We also note that the messages in transit in the configurations resulting from the two executions differ both in number and in contents.

9.3 Randomized rule application

In the previous section, we modified our rewrite strategy so that it could choose which rules to apply to the current term in a pseudo-random manner, and as we saw in the dining-philosopher's example, this allowed us to easily produce different executions by providing different seed values. The strategy was obviously fairer than our previous round robin based one. However, as the following small example will illustrate, this is not enough to provide an execution that is truly randomized.

9.3.1 An example: The dining philosophers with only one rewrite rule

The point of this example is to illustrate that randomization at the rule-level is not enough for fairness; we also need randomization at the object level. Consider once more the example from Section 7.2. What we will do is to change this example, and make the philosophers a bit "ruder", meaning that they are always able to eat, even if it means that they must use their bare hands in lack of chopsticks. To achieve this, we replace all the rewrite rules from the previous example with only one new *greedy-eat* rule (irrelevant attributes are ignored):

```
rl [greedy-eat] :
  < P : Philosopher | eatcount: C >
```

```
=>
  < P : Philosopher | eatcount: C + 1 > .
```

If we try to execute this specification with the original initial configuration (consisting of Socrates, Plato, Aristotle, Anaximander, Pythagoras and the butler James) using our strategy from Figure 9.2 on page 129, the results are devastating: the only philosopher that gets to eat is Anaximander, and the others will starve, regardless of what seed we supply for the randomization, as shown in the *Engine* object below after a thousand rewrites:

```
Engine[
  curTerm: '__[
    '<_:`Butler`|`sticks:_>
    [''Barry.Sort,'__['s_['0.Zero],'s_^2['0.Zero],'s_^3[
    '0.Zero],'s_^4['0.Zero],'s_^5['0.Zero]]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Anaximander.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^4['0.Zero],'s_^1000['0.Zero]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Aristotle.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^3['0.Zero],'0.Zero],

    '<_:`Philosopher`|`:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Plato.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^2['0.Zero],'0.Zero],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Pythagoras.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^5['0.Zero],'0.Zero],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Socrates.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_['0.Zero],'0.Zero]
  ],
  curModule: 'DINING-PHILOSOPHERS,labels: 'greedy-eat,
  failedRules: nil, numRules: 1,randomNum: 270655128
]
```

As we can see from the *eatcount* attribute of the philosopher objects, Anaximander has eaten a thousand times (represented at the meta-level as `'s_^1000['0.Zero]`), and all the others zero times (meta-represented as

'0.Zero). It is now quite clear that we need to extend the randomization to include objects (and terms within objects) as well as rewrite rules.

9.3.2 Randomization between and within objects

A given rule or equation may be applicable to a given term in several different *positions* within the term. A rule applied to a term at a given position is called a *solution*. As an example, consider the idempotency equation for sets, where *ELEMENT* is a variable that can hold an arbitrary element in a given set:

```
eq ELEMENT ELEMENT = ELEMENT .
```

Now, let's say we have a set of integers consisting of the following elements: 1 5 2 2 2 3. The idempotency equation can now be applied at two different positions in the set. The positions are underlined below:

- position 0: 1 5 $\underline{2\ 2}$ 2 3
- position 1: 1 5 2 $\underline{2\ 2}$ 3

In the same way, the *greedy-eat* rewrite rule from the previous dining philosophers example can be applied at several positions within the configuration at each rewrite (one for each philosopher). Additionally, other rewrite rules may be applicable at several different positions *within* a single object. Hence, to make our strategy fair, we need to randomize which solution is used as well as which rule is used.

Maude's built-in *metaXapply*($\overline{\mathcal{R}}$, $\bar{t}$, $\bar{l}$, σ, n, b, m) function, that we have been making heavy use of in our strategy to perform the actual meta-level rewrites, allows us to control at which position within a given term the rewrite takes place by supplying a natural number parameter m. In the previous implementations of our strategy, we have supplied the value 0 for m, meaning that we have used the first (according to Maude's internal sorting algorithm) available solution.

In order to pick a randomized solution number, the first thing we need to know is how many solutions there are, and this is something that Maude does not make directly available to us (probably because this value is not calculated before it is needed by Maude's internal engine). To find this value, we make use of the fact that *metaXapply* returns *failure* when a given rule cannot be applied to a given term at a given position m, and that the solution numbers are always increasing by one and starting from zero (in other words, there are no gaps in the sequence of solution numbers). Knowing this, we can find the number of possible solutions for a rule with label $\bar{l}$ applied to a term $\bar{t}$ by repeatedly calling *metaXapply* with increasing values for m, until the function returns *failure*.

Knowing the number of solutions s, we can choose a solution number randomly by drawing a number r from the *RANDOM* module of Figure 9.1 on page 123 and calculating $r\ mod\ s$. We could have used the number in the *randomNum* property of the *Engine* object for r, but since we are already using this for drawing rule labels at random and we want this to be independent of the solution number, we add another natural number property to the *Engine* object; *randomNum2*. Furthermore, we allow the user to specify two separate seed values to the *start* function.

Figure 9.3 on page 137 shows the entire *exec-random-2* rewrite strategy, which now selects the solution number as well as the rule label pseudo-randomly. The main difference from our previous implementation (Figure 9.2 on page 129), is the calls to a new auxiliary function *ChooseSolution* : $Qid \times Term \times Qid \times Int \times Pred \times MsgList \rightarrow Int$, found in lines 7, 11, 33 and 42. The *ChooseSolution* function takes the name of a module MOD, the meta-representation of the current configuration $\bar{t}$, a rule label $\bar{l}$, a random number r, a predicate p and a communication history ML in the form of a message list, and returns the chosen solution number. It determines the number of possible solutions s for applying $\bar{l}$ to $\bar{t}$, and then selects one of these by a modulo operation on r. The *CheckPredicate* function is then called to see whether the attempted rule application is in concordance with the predicate p. If this is the case, the solution number is returned and the history is updated with any new messages. If, on the other hand, it is not in accordance with the predicate p, the *CheckPredicate* does one of two things depending on the mode in which the predicate is checked:

- If the mode is *force*, and there are solutions that have not yet been tried, the next solution $r' = (r + 1)\ mod\ s$ is checked for compliance with the predicate. If this check succeeds, r' is returned, otherwise the next solution is tried. If no solution number can be found, -1 is returned.

- On the other hand, if the mode is *fail-stop*, the execution is terminated immediately if the predicate check fails, and -2 is returned to indicate a fail stop to the *exec-random-2* rule of Figure 9.3 on page 137. If the check succeeds, the solution number is returned in the same way as in the previous bullet.

The detailed code for the new versions of the auxiliary functions *Choose-Solution* and *CheckPredicate* can be found in Appendix A.

If we try to execute the example with only one greedy *eat* rule from Section 9.3.1, we see that our new rewrite strategy is capable of applying the rule fairly between the philosophers, and that we are able to provide different executions by changing the two seed values. Below is the resulting configuration from running the example with seed values 1 and 2 and performing 1000 rewrites:

```
Engine[
  curTerm: '__[
    '<_:`Butler`|`sticks:_>
    [''Barry.Sort,'__['s_['0.Zero],'s_^2['0.Zero],'s_^3[
    '0.Zero],'s_^4['0.Zero],'s_^5['0.Zero]]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Anaximander.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^4['0.Zero],'s_^237['0.Zero]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Aristotle.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^3['0.Zero],'s_^195['0.Zero]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Plato.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^2['0.Zero],'s_^172['0.Zero]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Pythagoras.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_^5['0.Zero],'s_^212['0.Zero]],

    '<_:`Philosopher`|`state:_`,leftStick:_`,rightStick:_`,
    butler:_`,seat:_`,eatcount:_>
    [''Socrates.Sort,'thinking.State,'no.StickState,
    'no.StickState,''Barry.Sort,'s_['0.Zero],'s_^184['0.Zero]]
  ],
  curModule:'DINING-PHILOSOPHERS,labels: 'greedy-eat,failedRules:
  nil, numRules: 1,randomNum:2021703321,randomNum2: 1895922995
]
```

We observe that the values of the eat counters are quite evenly distributed, with Anaximander, Aristotle, Plato, Pythagoras and Socrates having eaten 237, 195, 172, 212 and 184 times, respectively.

```
1.  crl [exec-random-2] :
2.     Engine[curTerm: T , curModule: MOD, labels: LABELS ,
3.       failedRules: FAILEDRULES, numRules: NUMRULES,
4.       randomNum: RANDOMNUM, randomNum2: RANDOMNUM2]
5.     History[h: ML, pred: PRED]
6.  =>
7.     if ChooseSolution(MOD, T,
8.        findItem(LABELS, RANDOMNUM rem NUMRULES),
9.        RANDOMNUM2, PRED, ML) < 0
10.    then
11.       if ChooseSolution(MOD, T,
12.          findItem(LABELS, RANDOMNUM rem NUMRULES),
13.          RANDOMNUM2, PRED, ML) == -1
14.       then
15.          Engine[curTerm: T , curModule: MOD, labels: LABELS ,
16.            failedRules:
17.              if findItem(LABELS, RANDOMNUM rem NUMRULES) in FAILEDRULES
18.              then
19.                FAILEDRULES
20.              else
21.                FAILEDRULES findItem(LABELS, RANDOMNUM rem NUMRULES)
22.              fi,
23.            numRules: NUMRULES, randomNum: rand(RANDOMNUM),
24.            randomNum2: rand(RANDOMNUM2)]
25.       else
26.          Fail[label: findItem(LABELS, RANDOMNUM rem NUMRULES), state: T]
27.       fi
28.       History[h: ML, pred: PRED]
29.    else
30.      Engine[curTerm: getTerm(metaXapply([MOD], T,
31.          findItem(LABELS, RANDOMNUM rem NUMRULES),
32.          none, 0, unbounded,
33.          ChooseSolution(MOD, T,
34.            findItem(LABELS, RANDOMNUM rem NUMRULES),
35.            RANDOMNUM2, PRED, ML))) ,
36.       curModule: MOD , labels: LABELS,
37.       failedRules: nil, numRules: NUMRULES, randomNum: rand(RANDOMNUM),
38.       randomNum2: rand(RANDOMNUM2)]
39.      History[h: ML @ getNewMessages(T, getTerm(metaXapply([MOD], T,
40.          findItem(LABELS, RANDOMNUM rem NUMRULES),
41.          none, 0, unbounded,
42.          ChooseSolution(MOD, T,
43.            findItem(LABELS, RANDOMNUM rem NUMRULES),
44.            RANDOMNUM2, PRED, ML))), MOD, ML),
45.          pred: PRED]
46.    fi
47. if length(FAILEDRULES) < length(LABELS) .
```

Figure 9.3: A rewrite strategy with randomized rule and solution number selection

Chapter 10

Communication over Sockets

Maude comes with very few built-in possibilities for communicating with "the rest of the world". For example, there is no support for writing to files, very limited GUI support, no support for external object creation (like for instance COM [37] or SOAP [19]) etc. Furthermore, there is no way for a Maude specification to communicate with another process, be it local or remote, and in this chapter, we will look at how we can work around this specific limitation.

In [30], an approach to this problem based on the actor model of distributed computation [1] is presented. In this paper, a Maude actor *IMaude* [22] is presented, that can interact with other processes through the *InterOperability Platform (IOP)* [23].

We will, in this chapter, take a somewhat more straightforward approach, in which there is really no extra Maude framework code that needs to be written, and no changes need to be done to the actual Maude or Creol specifications.

The general idea is that since Maude writes to *standard out*, and reads from *standard in* [36], we can add communication capabilities by "wrapping" the Maude process in an external process that controls standard in and out. By reading Maude's output we can identify which messages are destined for another process, and send them over the network via socket communication. If we at the same time listen for incoming messages on the network, and are able to insert such messages into the Maude configuration in-between rewrites, we have established network communication support for Maude specifications. An outline of this concept is shown in Figure 10.1 on the next page.

We will write the wrapper program in Python [34], since this language has good socket and threading support, and in general is a good RAD (rapid application development) tool.

The wrapper program will have one queue for incoming messages, *inq*, one queue for outgoing messages, *outq*, and a table *objectLocations* that

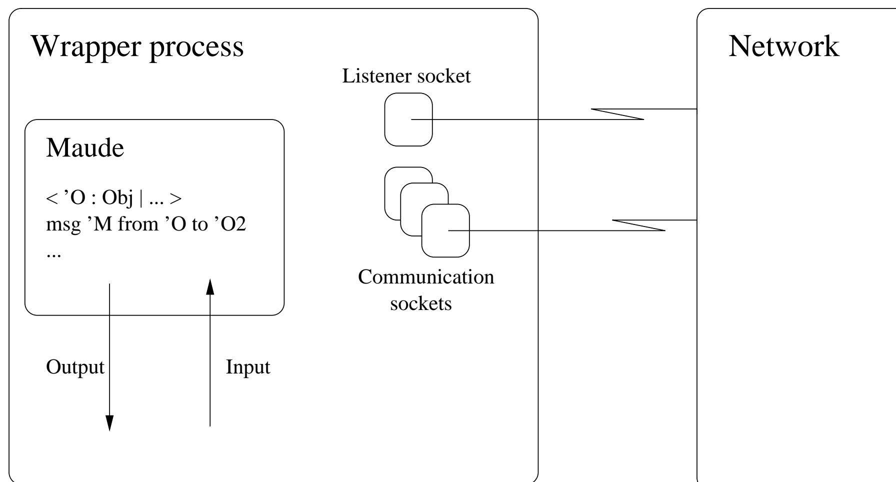

Figure 10.1: A schematic view of socket communication in Maude

specifies the locations on the network for the remote objects. The program will consist of several threads with specific tasks:

The main thread The task of the main thread is to accept command line parameters from the user, and to start the thread that controls the Maude process, the listener thread and the sender thread. The command line parameters for the main thread are:

- The file name of the Maude source code file in which the specification that is to be executed is stored.
- The file name of an XML file that specifies where on the network each remote object is located. An example of such a file is shown in Figure 10.2 on the facing page. This file will be read into the *object-Locations* table.
- The Maude module in which the rewrites are to be performed (optional).
- The maximum number of rewrites that are to be performed (optional).

The Maude controller thread The task of this thread is to control a Maude process by providing rewrite commands and terms as input and capturing its output. It will keep the current Maude configuration in a local variable *curTerm.* The thread will be running in a loop, and for each iteration in this loop, the following operations will be performed:

```
<objectLocations>
  <object>
    <name>'Sender</name>
    <machine>ringhorni.ifi.uio.no</machine>
    <port>5001</port>
  </object>
  <object>
    <name>'Receiver</name>
    <machine>cammarata.ifi.uio.no</machine>
    <port>5003</port>
  </object>
</objectLocations>
```

Figure 10.2: An example XML file with locations on the network for two objects

- Check the *inq* for any new messages, and if there are any, add them to the current configuration in *curTerm*.
- Start Maude, and instruct it to perform one rewrite of the current configuration. If this is the very first iteration of the loop, the initial configuration is assumed to be specified by an $init : \rightarrow Configuration$ constant.
- Read Maude's output, extract the resulting configuration, and store it in the local variable *curTerm*.
- Quit Maude.
- Check the new configuration for any new messages that are destined for a remote object, by looking up the individual receivers in the *objectLocations* table. If there are any such messages, remove them from the configuration, and put them in the out queue *outq*.

It may seem a bit odd that Maude is started and terminated for each iteration of the loop, and indeed this was not the initial design. To begin with the communication with Maude was done through *Unix pipes*, and Maude was only started once in the lifetime of the wrapper program. This did, however, prove to be very unstable, and Maude would lock up at arbitrary times. The reason for this is not known.

The listener thread The listener thread is started by the main thread, and lives as long as the program is executing. It instantiates a listener socket, and listens for incoming connection requests from other processes. When a

connection request arrives, the connection is accepted, and a new instance of a receiver thread is started to handle the communication with the remote process.

The receiver thread The task of a receiver thread is to handle a specific communication request. It is started by the listener thread, and reads incoming data from a socket dedicated to this connection. The client that requested the connection transfers a Maude message (as a string) over the network. When the transmission is complete, the new message is added to the in queue *inq*, the socket is closed and the thread terminates.

The sender thread This thread is instantiated at startup by the main thread, and monitors the out queue *outq*. For each new message in *outq*, the sender thread

- establishes a socket connection to the remote process,
- sends the message,
- removes the message from the out queue, and
- closes the socket.

By using two or more of these wrapper processes, and setting up the *objectLocations* tables accordingly, we are able to let objects in one Maude specification send standard Maude messages to objects in another specification, that might be running on a different machine on the network. An example setup is shown in Figure 10.3 on the next page.

In Chapter 9 we introduced random execution based on a pseudo-random number generator. By using the approach from this chapter, we can have *true* non-determinism when it comes to exchange of messages, seeing as the network might introduce arbitrary delays, and CPUs run at different speeds. This makes for a much more realistic testing environment for specifications.

Furthermore, there is, in principle, no need for the processes with which the Maude process communicates to be other Maude processes. As long as the messages that are put in the Maude configuration are of a form that Maude can understand, the other objects may be implemented in any language. Not only does this make Maude able to communicate with programs and components of various kinds, but it also makes it possible to *test* that third-party objects behave according to their specification, by using the predicate checking mechanisms introduced in this thesis.

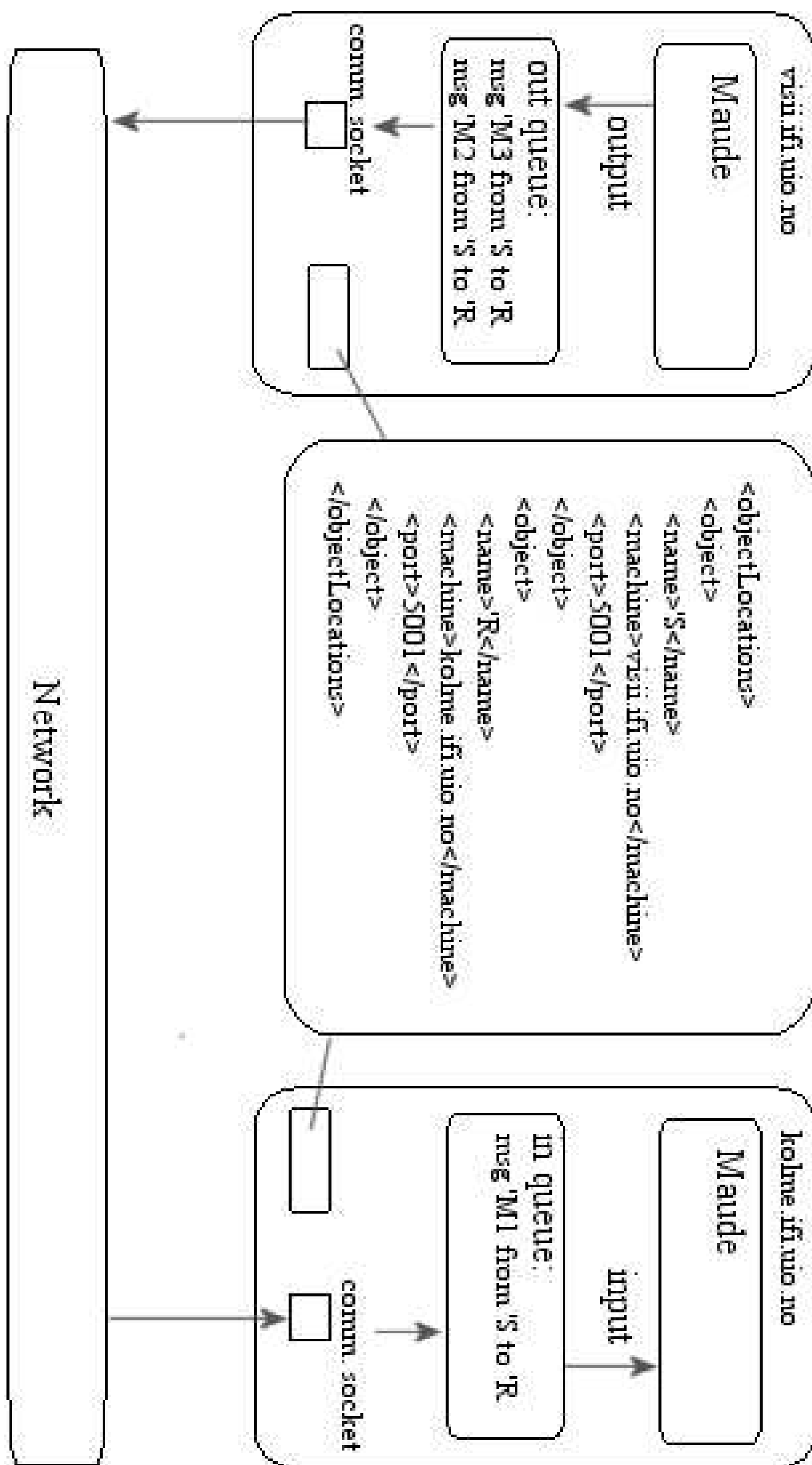

Figure 10.3: An example setup with two Maude processes communicating across the network through sockets

Chapter 11

Conclusion

In this thesis we have presented a non-deterministic rewrite strategy that can record a communication history during runtime, and control the execution of a specification according to predicates on this history.

The rewrite strategy has evolved throughout the thesis, from the basic round robin strategy of Figure 4.2 on page 36 to the full-fledged strategy that supports history logging, predicate checking, predicate enforcement, execution of both Maude and Creol specifications and randomization of rules and rule applications of Figure 9.3 on page 137.

11.1 Main results from this thesis

In Section 1.3 of the introduction, we presented our primary goal for this thesis, which was

to develop a framework for executing specifications modeling distributed systems, that can record and utilize a communication history.

The work towards reaching this goal has resulted in a meta-level Maude framework that can execute meta-representations of both Maude and Creol specifications. During runtime, the communication history is logged, and predicates can be specified to control the behavior of the executing specification, or to test that the specification behaves according to its invariant.

To further concretize the goal, we presented four questions, which we will try to answer in a little more detail below:

Transparent history logging

How can we execute Maude specifications and transparently*, in the sense that the original specification remains unchanged, build a communication trace as the execution proceeds?*

This question is addressed in Chapter 4. The key points can be summarized as follows: A meta-level representation of the object-level configuration is stored in an *Engine* object, and the history is stored in a *History* object. A meta-level rewrite strategy controls the execution, and extracts any new messages from the configuration, and records them in the history. The fact that we are utilizing a meta-level strategy to execute the specifications, and a separate object to store the history, means that *no* changes are needed in the original specifications.

Predicates on the communication history

How can we define predicates on this trace, and use such predicates to control and test the behavior of objects?

Predicates on the communication history are introduced in Chapter 5. In chapters 6 and 8, we increase the expressiveness of our predicates by adding additional constructs, such as full support for regular expressions with parameters and scoped variables, and quantifiers.

An important point from this discussion, is that to specify predicates that are to be checked during execution, we must define *constructors* that cannot be further reduced by Maude, since at the time of specification the communication history H has not yet been recorded. Hence, we will have to parse the predicate specifications ourselves during runtime.

We also consider how the predicates can be used. In Chapter 5, the predicates are used by our rewrite strategy to control the execution so that illegal rewrites are not performed. Another approach is taken in Section 8.3, in which a mechanism for testing that an object conforms to its specification is presented. These two methods of predicate checking are called *force* mode and *fail-stop* mode, respectively. Modes may be be combined, such that different projections on the history can be checked in different modes. This technique can be used for component testing.

In Chapter 10, we also briefly consider how the predicate checking mechanisms can be applied to external components through the use of a socket extension to Maude.

Application to the Creol language

How can these techniques be applied to the Creol language, and more specifically, to the Creol interpreter developed in Maude?

The concepts introduced in this thesis, can all be applied to Creol Machine Code executed by the Creol interpreter as well as to standard Maude object based specifications. Since a meta-level approach is used, we need not worry about how the underlying specification is implemented or executed.

There are, however, some minor considerations regarding the structure of messages and how the interpreter handles local calls, that are dealt with in Section 8.2. The important thing, though, is that no *conceptual* change is needed to make the framework function with the Creol interpreter. This fact supports the claim that the framework we have developed is general enough to be useful for a wide variety of object based rewriting logic specifications.

Non-determinism

How can we execute models of highly non-deterministic concurrent systems, such as Creol programs, in the deterministic rewrite tool Maude?

We have seen that Maude's built-in strategies *rew* and *frew* are not suitable for executing rewriting logic models of non-deterministic problems. In Chapter 9, we introduced a pseudo-random rewrite strategy based on a pseudo-random number generator. We showed how a two-level randomization was needed, both a the rule level and at the solution level. With this strategy, non-deterministic problems are treated fairly, and we are also able to provide different executions by specifying different seed values.

With the socket extension to Maude presented in Chapter 10, we are also able to introduce true non-determinism by allowing Maude specifications to communicate over a real network, with varying latency and CPU speeds, possible packet loss etc.

11.2 Future work

In Chapter 4 we introduced the *Engine* object, that contains the current term, the name of the current module, a list of rule labels from the current module etc. This object is used by the rewrite strategy when performing rewrites. It could perhaps be interesting to extend this concept to allow for more than one *Engine* object, so that several independent specifications could be executed at the same time (or interchangeably).

Also in Chapter 4, we looked at how to record the communication history by checking the configuration recursively for new messages in-between rewrites. As we recall, we left it as an open question if this method was the best way to go. An alternative approach could be to analyze the rewrite rules in a specification at the meta-level, to see which rules may produce a message under which conditions, and log the messages in the history based on this analysis.

The predicates specified in this thesis have been independent from the actual object specifications, in the sense that they are given explicitly as a parameter to the rewrite strategy, as opposed to being a part of the specifications themselves. In Creol, invariants and assumptions are specified as parts of an interface. A tighter integration with Creol in this respect could

be considered, in which the strategy would read the predicate directly from the Creol Machine Code. For this to work, the CMC from [3] would also have to be modified to include this information.

The predicate checking in this thesis is performed dynamically during run-time. A static approach could also be an interesting study, in which the predicate is checked against *all* possible states for a given specification (from a given initial state). For this, Maude's built-in *search* command might perhaps be utilized.

The socket extension to Maude presented in Chapter 10 allows Maude to communicate with other processes. Developing some examples of interaction between Maude and other types of software could be an interesting task, especially with focus on component testing using the predicate framework from this thesis.

Bibliography

[1] G. Agha. *A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] Marte Arnestad. En abstrakt maskin for Creol i Maude. Master's thesis, Department of Informatics, University of Oslo, November 2003. In Norwegian. Available from `http://heim.ifi.uio.no/~creol`.

[4] Eyvind W. Axelsen, Einar Broch Johnsen, and Olaf Owe. Toward reflective application testing in open environments. Submitted for publication, 2004.

[5] R. Ben-Natan. *CORBA : a guide to the common object request broker architecture*. McGraw-Hill, New York, 1995.

[6] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[9] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marí-Oliet, and J. Meseguer. Metalevel computation in maude. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[11] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *Maude Manual - Version 2.1*. SRI International, 2004.

[12] The Creol web page. `http://home.ifi.uio.no/~creol`.

[13] O.-J. Dahl, K. Nygaard, and B. Myrhaug. Simula 67 common base language. Technical Report S-2, Norwegian Computing Center, Oslo, 1968.

[14] O.-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, pages 57–114. Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, Dec. 1977.

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[16] F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, SRI International, 1999.

[17] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer Academic Publishers, 2003.

[18] James Gosling, Bill Joy, Guy L. Steele, and Gila Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, Mass., second edition, 2000.

[19] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. The SOAP version 1.2 specification. Technical report, The World Wide Web Consortium, 2003. Available from `http://www.w3.org`.

[20] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C♯ Programming Language*. Microsoft .NET Development Series. Addison-Wesley Professional, 2003.

[21] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ., 1985.

[22] The IMaude web page. `http://www.csl.sri.com/~clt/IMaudeWeb/`.

[23] The IOP web page. `http://mcs.une.edu.au/~iam/IOP/`.

[24] Einar Broch Johnsen, Olaf Owe, and Eyvind W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2004.

[25] Einar Broch Johnsen and Olaf Owe. A compositional formalism for object viewpoints. In Bart Jacobs and Arend Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer Academic Publishers, March 2002.

[26] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*. IEEE Computer Society Press, September 2004. To appear.

[27] Einar Broch Johnsen and Olaf Owe. Object-oriented specification and open distributed systems. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[28] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, UK, June l981.

[29] K. C. Louden. *Compiler Construction - principles and practice*. PWS Publishing Company, 1997.

[30] Ian A Mason and Carolyn L. Talcott. IOP: The interoperability platform & IMaude: An interactive extension of Maude. In *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2004.

[31] The Maude web page. `http://maude.cs.uiuc.edu/`.

[32] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[33] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Falnnery. *Numerical Recipes in C - The Art of Scientific Computing*. Cambridge University Press, second edition, 2002. First edition originally published in 1988.

[34] The Python web page. `http://www.python.org`.

[35] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[36] The Unix web page. `http://www.unix.org`.

[37] Sara Willliams and Charlie Kindel. *The Component Object Model: A Technical Overview*. Microsoft Corporation, 1994.

[38] P. C. Ølveczky. *Formal Modeling and Analysis of Distributed Systems In Maude. Lecture notes for INF220.* Department of Informatics, University of Oslo, 2002.

Appendix A

Source Code

A.1 Rewrite strategy

```
1   mod META-ENGINE is
2
3       protecting META-LEVEL .
4       protecting QID-LIST-EXT .
5       protecting QID .
6       protecting RANDOM .
7       protecting PRED .
8       protecting CONVERSION .
9       protecting STRING .
10
11      var MOD : Qid .
12      var T : Term .
13      var R : Rule .
14      var RS : RuleSet .
15      vars LABEL SUBSCRIPTION : Qid .
16      vars LABELS SUBSCRIPTIONS FAILEDRULES : QidList .
17      vars T1 T2 T3 T4 : Term .
18      var AS : AttrSet .
19      var Q : Qid .
20      vars QL QL2 QL3 FORCE-LABELS FAILSTOP-LABELS : QidList .
21      var EC : EngineConfig .
22      var CONDITION : Condition .
23      vars I NUMRULES SEED SEED2 RANDOMNUM RANDOMNUM2 : Int .
24      vars PRED PRED2 : Pred .
25      vars ML ML2 : MsgList .
26      var MODE : Mode .
27
28
29      sort EngineConfig .
30      sort EngineObject .
31      subsort EngineObject < EngineConfig .
32
33      *** Definitions of the EngineObjects:
34      op Engine[curTerm:_ , curModule:_ , labels:_ , failedRules:_ ,
35          numRules:_, randomNum:_, randomNum2:_] :
```

```
36          Term Qid QidList QidList Int Int Int -> EngineObject .
37       op History[h:_ , pred:_] : MsgList Pred -> EngineObject .
38       op Fail[label:_ , state:_] : Qid Term -> EngineObject .
39
40       op __ : EngineConfig EngineConfig -> EngineConfig
41          [ctor assoc comm id: noEngine] .
42       op noEngine : -> EngineConfig [ctor] .
43
44
45       ***(
46       This is the equation that is used to start
47       meta-level rewriting.
48      ***)
49       op start : Qid Term -> EngineConfig .
50       op start : Qid Term Pred -> EngineConfig .
51       op start : Qid Term Pred Int -> EngineConfig .
52       op start : Qid Term Pred Int Int -> EngineConfig .
53
54       eq start(MOD, T) =
55          start(MOD, T, TRUE) .
56       eq start(MOD, T, PRED) =
57          start(MOD, T, PRED, seed) .
58       eq start(MOD, T, PRED, SEED) =
59          start(MOD, T, PRED, SEED, seed) .
60
61       eq start(MOD, T, PRED, SEED, SEED2) =
62          Engine[curTerm: T, curModule: MOD, labels: getRuleLabels(MOD),
63            failedRules: nil, numRules: length(getRuleLabels(MOD)),
64            randomNum: rand(SEED), randomNum2: rand(SEED2)]
65          History[h: nil, pred: PRED] .
66
67       op getLabelsFromRuleSet : RuleSet -> QidList .
68       eq getLabelsFromRuleSet(none) = nil .
69       eq getLabelsFromRuleSet(rl T1 => T2 [label(LABEL) AS] . RS) =
70          LABEL getLabelsFromRuleSet(RS) .
71       eq getLabelsFromRuleSet(crl T1 => T2 if CONDITION
72          [label(LABEL) AS] . RS) =
73          LABEL getLabelsFromRuleSet(RS) .
74
75       op getRuleLabels : Qid -> QidList .
76       eq getRuleLabels(MOD) = getLabelsFromRuleSet(upRls(MOD, true)) .
77
78       crl [exec] :
79          Engine[curTerm: T , curModule: MOD, labels: LABELS ,
80            failedRules: FAILEDRULES, numRules: NUMRULES,
81            randomNum: RANDOMNUM, randomNum2: RANDOMNUM2]
82          History[h: ML, pred: PRED]
83       =>
84          *** To begin with, we check if the rule is enabled,
85          *** and that it can be applied according to the
86          *** predicate:
87          if ChooseSolution(MOD, T,
88             findItem(LABELS, RANDOMNUM rem NUMRULES),
89             RANDOMNUM2, PRED, ML) < 0
```

```
 90          then
 91             *** Error codes:
 92             ***  -1: The rule could not be applied, that is
 93             ***      there were no enabled solutions
 94             ***  -2: The rule was enabled, but was stopped by
 95             ***      the predicate:
 96             if ChooseSolution(MOD, T,
 97                findItem(LABELS, RANDOMNUM rem NUMRULES),
 98                RANDOMNUM2, PRED, ML) == -1
 99             then
100                Engine[curTerm: T , curModule: MOD, labels: LABELS ,
101                  failedRules:
102                    if findItem(LABELS, RANDOMNUM rem NUMRULES)
103                       in FAILEDRULES
104                    then
105                      FAILEDRULES
106                    else
107                      FAILEDRULES findItem(LABELS, RANDOMNUM
108                        rem NUMRULES)
109                    fi,
110                  numRules: NUMRULES, randomNum: rand(RANDOMNUM),
111                  randomNum2: rand(RANDOMNUM2)]
112             else
113                *** Choosesolution returned -2,
114                *** we were stopped by the predicate:
115                Fail[label: findItem(LABELS, RANDOMNUM rem NUMRULES),
116                  state: T]
117             fi
118             History[h: ML, pred: PRED]
119          else
120            *** The test of this rule was ok, we apply it:
121            Engine[curTerm: getTerm(metaXapply([MOD], T,
122                findItem(LABELS, RANDOMNUM rem NUMRULES),
123                none, 0, unbounded,
124                ChooseSolution(MOD, T,
125                  findItem(LABELS, RANDOMNUM rem NUMRULES),
126                    RANDOMNUM2, PRED, ML))) ,
127              curModule: MOD , labels: LABELS,
128              failedRules: nil, numRules: NUMRULES, randomNum:
129              rand(RANDOMNUM), randomNum2: rand(RANDOMNUM2)]
130            History[h: ML @ getNewMessages(T, getTerm(metaXapply(
131                [MOD], T, findItem(LABELS, RANDOMNUM rem NUMRULES),
132                none, 0, unbounded,
133                ChooseSolution(MOD, T,
134                  findItem(LABELS, RANDOMNUM rem NUMRULES), RANDOMNUM2,
135                  PRED, ML))), MOD, ML),
136              pred: PRED]
137          fi
138       *** If all the rules have failed, we cannot do anything more:
139       if length(FAILEDRULES) < length(LABELS) .
140
141
142       vars Q1 Q2 Q3 : Qid .
143       vars GTL1 GTL2 : GroundTermList .
```

```
144      vars GT1 GT2 GT3 : GroundTerm .
145      var C : Constant .
146      vars SOLUTION MAXSOLUTIONINDEX : Int .
147      var M : Msg .
148
149      op ChooseSolution : Qid Term Qid Int Pred MsgList -> Int .
150      eq ChooseSolution(MOD, T, LABEL, RANDOMNUM2, PRED, ML) =
151         if MaxSolutionIndex(MOD, T, LABEL) == -1 then -1
152         else
153           *** Must check if the chosen solution is compatible
154           *** with the predicate:
155           CheckSolutionAgainstPredicate(MOD, T, LABEL, PRED, ML,
156              rand(RANDOMNUM2) rem (MaxSolutionIndex(MOD, T, LABEL) + 1),
157              MaxSolutionIndex(MOD, T, LABEL), 0)
158         fi .
159
160      *** Checks whether a given solution is valid according to
161      *** the predicate. If not, the remaining solutions are tried.
162      *** If no valid solutions can be found, -1 is returned.
163      op CheckSolutionAgainstPredicate : Qid Term Qid Pred
164         MsgList Int Int -> Int .
165      op CheckSolutionAgainstPredicate : Qid Term Qid Pred
166         MsgList Int Int Int -> Int .
167
168      eq CheckSolutionAgainstPredicate(MOD, T, LABEL, PRED, ML,
169         SOLUTION, MAXSOLUTIONINDEX) =
170         CheckSolutionAgainstPredicate(MOD, T, LABEL, PRED, ML,
171           SOLUTION, MAXSOLUTIONINDEX, 0) .
172
173      eq CheckSolutionAgainstPredicate(MOD, T, LABEL, PRED, ML,
174         SOLUTION, MAXSOLUTIONINDEX, I) =
175         if I > MAXSOLUTIONINDEX + 1 then
176           *** We have tried every solution without
177           *** finding any that could be used with the
178           *** predicate
179           -1
180         else
181           *** Test the current solution:
182           if bool(CheckPredicate(PRED, ML @
183             getNewMessages(T, getTerm(metaXapply([MOD],
184               T, LABEL, none, 0, unbounded, SOLUTION)),
185               MOD, ML)))
186           then s
187             SOLUTION
188           else
189             *** Current solution failed
190             if mode(CheckPredicate(PRED, ML @
191               getNewMessages(T, getTerm(metaXapply([MOD],
192               T, LABEL, none, 0, unbounded, SOLUTION)),
193               MOD, ML))) == force
194             then
195                *** Test the next solution
196               CheckSolutionAgainstPredicate(MOD, T, LABEL, PRED, ML,
197                (SOLUTION + 1) rem (MAXSOLUTIONINDEX + 1),
```

```
198                 MAXSOLUTIONINDEX, I + 1)
199             else
200               *** Mode = fail-stop;
201               *** Return an error code when an illegal rule
202               *** application is attempted.
203                 -2
204             fi
205           fi
206         fi
207        .
208
209       *** Returns the highest available solution index.
210       *** If no solution can be found, -1 is returned.
211       op MaxSolutionIndex : Qid Term Qid -> Int .
212       op MaxSolutionIndex : Qid Term Qid Int -> Int .
213
214       eq MaxSolutionIndex(MOD, T, LABEL) =
215          MaxSolutionIndex(MOD, T, LABEL, 0) .
216
217       eq MaxSolutionIndex(MOD, T, LABEL, I) =
218          if (metaXapply([MOD], T, LABEL, none, 0, unbounded, I)
219             =/= failure)
220          then
221            MaxSolutionIndex(MOD, T, LABEL, I + 1)
222          else
223            I + (- 1)
224          fi .
225
226
227
228       op isMetaMessage : Term Qid -> Bool .
229       eq isMetaMessage(T1, MOD) = wellFormed([MOD], 'M:Msg <- T1) .
230
231       op countGroundTerms : GroundTerm GroundTermList -> Int .
232       op countGroundTerms : GroundTerm GroundTermList Int -> Int .
233       eq countGroundTerms(GT1, GTL2) = countGroundTerms(GT1, GTL2, 0) .
234       eq countGroundTerms(GT1, GT2, I) = if GT1 == GT2 then 1 else 0 fi .
235       eq countGroundTerms(GT1, (GT2, GTL2), I) =
236          if GT1 == GT2 then
237            1 + countGroundTerms(GT1, GTL2, I + 1)
238          else
239            if I > 0 then
240              0
241            else
242              countGroundTerms(GT1, GTL2, 0)
243            fi
244          fi .
245
246       op strip : Qid -> Qid .
247       eq strip (Q) = qid (substr (string (Q), 1, length (string (Q)))) .
248
249       vars GTLABEL GTLABEL2 GTFROM GTTO GTMETHOD GTPARAM : GroundTerm .
250       vars FROM1 FROM2 TO : Qid .
251       var L : List .
```

```
252
253     op GroundTermToMessage : GroundTerm MsgList -> Msg .
254     eq GroundTermToMessage('msg_from_to_[GT1, GT2, GT3 ], ML) =
255        msg strip(getName(GT1)) from strip(getName(GT2))
256        to strip(getName(GT3)) . ***label ''noLabel.Qid .
257      eq GroundTermToMessage('msg_from_to_['_`(_`)[GT1, GTPARAM],
258         GT2, GT3 ], ML) =
259        msg strip(getMsgName(GT1))(downParams(GTPARAM))
260          from strip(getName(GT2))
261          to strip(getName(GT3)) label ''noLabel.Qid .
262     eq GroundTermToMessage('invoc`(_`,_`,_`,_`,_`)[GTLABEL,
263        GTFROM, GTTO, GTMETHOD, GTPARAM], ML) =
264        msg invoc (strip(getName(GTMETHOD))(downParams(GTPARAM)))
265          from strip(getName(GTFROM))
266          to strip(getName(GTTO))
267          label GTLABEL .
268     eq GroundTermToMessage('comp`(_`,_`,_`)[GTLABEL,
269        GTTO, GTPARAM], ML) =
270        msg comp (getMsgName(GTLABEL, GTTO, ML)(downParams(GTPARAM)))
271          from getMsgReciever(GTLABEL, GTTO, ML)
272          to strip(getName(GTTO))
273          label GTLABEL .
274
275     *** Transform a meta-level parameter list to
276     *** an object-level parameter list.
277     op downParams : GroundTerm -> List .
278     eq downParams('nil.List) = (nil).List .
279     eq downParams('int['s_['0.Zero]]) = int(1) .
280     eq downParams('int['0.Zero]) = int(0) .
281     eq downParams('int[Q['0.Zero]]) =
282        int(rat(substr(string(Q),3,100000000),10)) .
283     eq downParams('__[GT1, GTL1]) = downParams(GT1) downParams(GTL1) .
284     eq downParams(Q) = oid(strip(getName(Q))) [otherwise] .
285
286     ops getMsgName getMsgReciever : GroundTerm Qid MsgList -> Qid .
287     eq getMsgName(GTLABEL, FROM1, nil) = 'LocalCall .
288     eq getMsgName(GTLABEL, FROM1, (msg M from FROM2 to
289        TO label GTLABEL2) @ ML) =
290        if strip(getName(FROM1)) == FROM2 and GTLABEL == GTLABEL2
291        then getMethod(M) else
292          getMsgName(GTLABEL, FROM1, ML) fi .
293
294     eq getMsgReciever(GTLABEL, FROM1, nil) = strip(getName(FROM1)) .
295     eq getMsgReciever(GTLABEL, FROM1, (msg M from FROM2 to
296        TO label GTLABEL2) @ ML) =
297        if strip(getName(FROM1)) == FROM2 and GTLABEL == GTLABEL2
298        then TO else
299          getMsgReciever(GTLABEL, FROM1, ML) fi .
300
301     op getMethod : Msg -> Qid .
302     eq getMethod(invoc (Q(L))) = Q .
303     eq getMethod(comp (Q(L))) = Q .
304
305     op getMsgName : GroundTerm -> Qid .
```

```
306      eq getMsgName(Q[GTL1]) = Q .
307      eq getMsgName(GT1) = getName(GT1) [otherwise] .
308
309      *** The first parameter is the old term, the second is
310      *** the new one after the rewrite:
311      op getNewMessages : TermList TermList Qid MsgList -> MsgList .
312
313      eq getNewMessages('__[GTL1], '__[GT2, GTL2], MOD, ML) =
314         if isMetaMessage(GT2, MOD) and countGroundTerms(GT2, GTL1) <
315            countGroundTerms(GT2, (GT2, GTL2))
316         then
317            GroundTermToMessage(GT2, ML) @ getNewMessages('__[GTL1],
318            '__[GTL2], MOD, ML)
319         else
320            getNewMessages('__[GTL1], '__[GTL2], MOD, ML)
321         fi .
322
323      eq getNewMessages(GT1, '__[GT2, GTL2], MOD, ML) =
324         if isMetaMessage(GT2, MOD) and countGroundTerms(GT2, GT1) <
325            countGroundTerms(GT2, (GT2, GTL2))
326         then
327            GroundTermToMessage(GT2, ML) @ getNewMessages('__[GT1],
328              '__[GTL2], MOD, ML)
329         else
330            getNewMessages('__[GT1], '__[GTL2], MOD, ML)
331         fi .
332
333      eq getNewMessages('__[GTL1], '__[GT2], MOD, ML) =
334         if isMetaMessage(GT2, MOD) and countGroundTerms(GT2, GTL1) < 1
335         then
336            GroundTermToMessage(GT2, ML)
337         else
338            nil
339         fi .
340
341     eq getNewMessages(GT1, GT2, MOD, ML) =
342         if isMetaMessage(GT2, MOD) and countGroundTerms(GT2, GT1) < 1
343         then
344            GroundTermToMessage(GT2, ML)
345         else
346            nil
347         fi .
348
349
350  endm
```

A.2 Predicates

```
1  fmod PRED is
2
3      protecting MSG-LIST-FUNCS .
4      protecting INT .
5      protecting TRUTH-VALUE .
6      protecting STATE-SET .
7      protecting TOKEN .
8      protecting NFA-MATCH .
9      protecting QID-LIST-EXT .
10     protecting META-TERM .
11
12     sorts Pred BoolExp IntExp .
13     subsorts BoolExp IntExp < Pred .
14     subsort Int < IntExp .
15
16     sort History .
17     sort Projection .
18
19     sort ObjectVariable .
20     ops x y z : -> ObjectVariable .
21
22     sort QidOrVariable .
23     subsort Qid < QidOrVariable .
24     subsort ObjectVariable < QidOrVariable .
25
26     op _and_ : BoolExp BoolExp -> BoolExp [ctor assoc prec 54] .
27     op _or_ : BoolExp BoolExp -> BoolExp [ctor assoc prec 54] .
28
29     op _eq_  : IntExp IntExp -> BoolExp [ctor prec 53] .
30     op _lte_ : IntExp IntExp -> BoolExp [ctor prec 53] .
31     op _gte_ : IntExp IntExp -> BoolExp [ctor prec 53] .
32     op _lt_  : IntExp IntExp -> BoolExp [ctor prec 53] .
33     op _gt_  : IntExp IntExp -> BoolExp [ctor prec 53] .
34
35     op _plus_   : IntExp IntExp -> IntExp [assoc comm prec 33] .
36     op _minus_  : IntExp IntExp -> IntExp [prec 33 gather (E e)] .
37     op _times_  : IntExp IntExp -> IntExp [assoc comm prec 31] .
38
39     op FALSE : -> BoolExp .
40     op TRUE : -> BoolExp .
41
42     op H : -> History [ctor] .
43
44     op length : History -> IntExp [ctor] .
45
46     op if_then_else_fi : BoolExp Pred Pred -> Pred [ctor] .
47
48     op _/_ : History Projection -> History [ctor] .
49     op from : QidOrVariable -> Projection [ctor] .
50     op to : QidOrVariable -> Projection [ctor] .
51     op msgtype : Qid -> Projection [ctor] .
52
```

```
53      op _or_ : Projection Projection -> Projection [ctor assoc] .
54
55      sort Mode .
56      op force : -> Mode .
57      op fail-stop : -> Mode .
58      op Mode : Pred Mode -> BoolExp .
59
60      vars P1 P2 : Pred .
61      vars M M2 : Msg .
62      var ML : MsgList .
63      var I : Int .
64      var HIST : History .
65      var PROJ : Projection .
66      vars Q1 Q2 Q3 : Qid .
67      vars QL QL2 : QidList .
68      vars IE IE2 : IntExp .
69      var BE : BoolExp .
70      var PATTERN : Pattern .
71      var NFA : NFA .
72      vars PR1 PR2 : Projection .
73      var B : Bool .
74      vars OV1 OV2 : ObjectVariable .
75      var L : GroundTerm .
76      vars MODE MODE2 : Mode .
77
78      *** Prefix of regular sequence
79      op _prs_ : History Pattern ->  BoolExp .
80
81      op _prs_ : History NFA -> BoolExp .
82      eq HIST prs PATTERN = HIST prs MakeNFA(PATTERN) .
83
84      op MatchRE : History Pattern -> BoolExp .
85
86      op MatchRE : History NFA -> BoolExp .
87      eq MatchRE(HIST, PATTERN) = MatchRE(HIST, MakeNFA(PATTERN)) .
88
89      *** substitutions
90      sort Subst SubstitutionList .
91      subsort Subst < SubstitutionList .
92      op subst_with_ : ObjectVariable Qid -> Subst .
93      op nil : -> SubstitutionList .
94      op _,_ : SubstitutionList SubstitutionList -> SubstitutionList
95        [ctor assoc id: nil] .
96
97      *** quantifiers
98      op forAll_|_ : ObjectVariable Pred -> Pred [ctor] .
99      op exists_|_ : ObjectVariable Pred -> Pred [ctor] .
100     op forAll_elementOf_|_  : ObjectVariable QidList
101        Pred -> Pred [ctor] .
102     op exists_elementOf_|_  : ObjectVariable QidList
103        Pred -> Pred [ctor] .
104
105     var X : ObjectVariable .
106     var SL : SubstitutionList .
```

```
107     var SUBST : Subst .
108
109     sort BoolAndModeTuple .
110     op (_;_) : Bool Mode -> BoolAndModeTuple .
111
112     op bool : BoolAndModeTuple -> Bool .
113     eq bool( (B ; MODE) ) = B .
114     op mode : BoolAndModeTuple -> Mode .
115     eq mode( (B ; MODE) ) = MODE .
116
117     op CheckPredicate : Pred MsgList -> BoolAndModeTuple .
118     op CheckPredicate : Pred MsgList SubstitutionList ->
119        BoolAndModeTuple .
120     op CheckPredicate : Pred MsgList SubstitutionList Mode ->
121        BoolAndModeTuple .
122
123     eq CheckPredicate(P1, ML) = CheckPredicate(P1, ML, nil) .
124     eq CheckPredicate(P1, ML, SL) = CheckPredicate(P1, ML, SL, force) .
125
126     eq CheckPredicate(Mode(P1, MODE), ML, SL, MODE2) =
127        CheckPredicate(P1, ML, SL, MODE) .
128
129     eq CheckPredicate(P1 and P2, ML, SL, MODE) =
130        if bool(CheckPredicate(P1, ML, SL, MODE)) then
131           CheckPredicate(P2, ML, SL, MODE)
132        else
133           (false ; mode(CheckPredicate(P1, ML, SL, MODE)))
134        fi .
135
136     eq CheckPredicate(P1 or P2, ML, SL, MODE) =
137        if bool(CheckPredicate(P1, ML, SL, MODE)) then
138           CheckPredicate(P1, ML, SL, MODE)
139        else
140           CheckPredicate(P2, ML, SL, MODE)
141        fi  .
142
143     *** We are matching with an NFA instead of a DFA
144     ***(
145     eq CheckPredicate(HIST prs DFA, ML, SL) =
146        removeLabelFromMsg(ParseProjection(HIST, ML, SL)) prs DFA .
147     eq CheckPredicate(MatchRE(HIST, DFA), ML, SL) =
148        Match(removeLabelFromMsg(ParseProjection(HIST, ML, SL)), DFA) .
149     ***)
150
151     eq CheckPredicate(HIST prs NFA, ML, SL, MODE) =
152        ( (removeLabelFromMsg(ParseProjection(HIST, ML, SL))
153          prs NFA) ; MODE ) .
154     eq CheckPredicate(MatchRE(HIST, NFA), ML, SL, MODE) =
155        ( Match(removeLabelFromMsg(ParseProjection(HIST, ML, SL)),
156          NFA) ; MODE ) .
157
158
159     eq CheckPredicate(TRUE, ML, SL, MODE) = ( true ; MODE ) .
160     eq CheckPredicate(FALSE, ML, SL, MODE) = ( false ; MODE ) .
```

```
161
162
163     eq CheckPredicate(forAll X | P1, ML, SL, MODE) =
164        CheckPredicate(forAll X elementOf (GetObjectIDs(ML)) |
165          P1, ML, SL, MODE) .
166
167     eq CheckPredicate(forAll X elementOf (Q1 QL) | P1, ML, SL, MODE) =
168        if bool(CheckPredicate(P1, ML, (SL, subst X with Q1), MODE)) then
169           CheckPredicate(forAll X elementOf (QL) | P1, ML, SL, MODE)
170        else
171           (false ;  mode(CheckPredicate(P1, ML,
172             (SL, subst X with Q1), MODE)))
173        fi .
174     eq CheckPredicate(forAll X elementOf (nil) | P1, ML, SL, MODE) =
175        ( true ; MODE ) .
176
177     eq CheckPredicate(exists X | P1, ML, SL, MODE) =
178        CheckPredicate(exists X elementOf (GetObjectIDs(ML)) |
179         P1, ML, SL, MODE) .
180
181     eq CheckPredicate(exists X elementOf (Q1 QL) | P1, ML, SL, MODE) =
182        if bool(CheckPredicate(P1, ML, (SL, subst X with Q1), MODE)) then
183           CheckPredicate(P1, ML, (SL, subst X with Q1), MODE)
184        else
185          CheckPredicate(exists X elementOf (QL) | P1, ML, SL, MODE)
186        fi .
187     eq CheckPredicate(exists X elementOf (nil) | P1, ML, SL, MODE) =
188        ( false ; MODE ) .
189
190     op ReduceIntExp : IntExp MsgList SubstitutionList -> Int .
191
192     eq ReduceIntExp(length(HIST), ML, SL) =
193        len(ParseProjection(HIST, ML, SL)) .
194     eq ReduceIntExp(IE minus IE2, ML, SL) =
195        ReduceIntExp(IE, ML, SL) - ReduceIntExp(IE2, ML, SL) .
196     eq ReduceIntExp(I, ML, SL) = I .
197
198     eq CheckPredicate(IE eq IE2, ML, SL, MODE) =
199        ( (ReduceIntExp(IE, ML, SL) ==
200        ReduceIntExp(IE2, ML, SL)) ; MODE ) .
201     eq CheckPredicate(IE lte IE2, ML, SL, MODE) =
202        ( (ReduceIntExp(IE, ML, SL) <=
203        ReduceIntExp(IE2, ML, SL)) ; MODE ) .
204     eq CheckPredicate(IE gte IE2, ML, SL, MODE) =
205        ( ReduceIntExp(IE, ML, SL) >=
206        ReduceIntExp(IE2, ML, SL) ; MODE ) .
207     eq CheckPredicate(IE lt IE2, ML, SL, MODE) =
208        ( ReduceIntExp(IE, ML, SL) <
209        ReduceIntExp(IE2, ML, SL) ; MODE ) .
210     eq CheckPredicate(IE gt IE2, ML, SL, MODE) =
211        ( ReduceIntExp(IE, ML, SL) >
212        ReduceIntExp(IE2, ML, SL) ; MODE ) .
213     eq CheckPredicate(if BE then P1 else P2 fi, ML, SL, MODE) =
214        if bool(CheckPredicate(BE, ML, SL, MODE)) then
```

```
215            CheckPredicate(P1, ML, SL, MODE) else
216            CheckPredicate(P2, ML, SL, MODE) fi .
217
218      op GetObjectIDs : MsgList -> QidList .
219      op GetObjectIDs : MsgList QidList -> QidList .
220      eq GetObjectIDs(ML) = GetObjectIDs(ML, nil) .
221      eq GetObjectIDs(nil, QL) = nil .
222      eq GetObjectIDs((msg M from Q1 to Q2 label L) @ ML, QL) =
223         (if Q1 in QL then nil else Q1 fi)
224         (if Q2 in QL then nil else Q2 fi)
225         GetObjectIDs(ML, QL (if Q1 in QL then nil else Q1 fi)
226           (if Q2 in QL then nil else Q2 fi)) .
227     eq GetObjectIDs((msg Q3 from Q1 to Q2) @ ML, QL) =
228         (if Q1 in QL then nil else Q1 fi)
229         (if Q2 in QL then nil else Q2 fi)
230         GetObjectIDs(ML, QL (if Q1 in QL then nil else Q1 fi)
231           (if Q2 in QL then nil else Q2 fi)) .
232
233      op Project : Msg Projection SubstitutionList -> MsgList .
234      eq Project(M, PR1 or PR2, SL) =
235         if Project(M, PR1, SL) =/= nil then
236            Project(M, PR1, SL)
237         else
238            Project(M, PR2, SL)
239         fi .
240      eq Project(M, from(Q1), SL) = from(Q1, M) .
241      eq Project(M, from(OV1), SL) = from(Subst(OV1, SL), M) .
242      eq Project(M, to(Q1), SL) = to(Q1, M) .
243      eq Project(M, to(OV1), SL) = to(Subst(OV1, SL), M) .
244      eq Project(M, msgtype(Q1), SL) = msgtype(Q1, M) .
245
246      op Project : MsgList Projection SubstitutionList -> MsgList .
247      eq Project(nil, PR1, SL) = nil .
248
249      op Subst : ObjectVariable SubstitutionList -> Qid .
250      eq Subst(OV1, (subst OV1 with Q1), SL) = Q1 .
251      eq Subst(OV1, (SUBST, SL)) = Subst(OV1, SL) [otherwise] .
252
253      op ParseProjection : History MsgList SubstitutionList -> MsgList .
254      eq ParseProjection(H, ML, SL) = ML .
255      eq ParseProjection(HIST / PROJ, nil, SL) = nil .
256      eq ParseProjection(HIST / PROJ, M @ ML, SL) =
257         Project(ParseProjection(HIST, M, SL), PROJ, SL)
258         @ ParseProjection(HIST / PROJ, ML, SL) .
259
260
261  endfm
```

A.3 Regular expressions

```
1
2   in obj.maude .
3   in msglist.maude .
4
5
6   *****************************************
7   ***(
8   Pattern constructors for regular expressions
9   :: is used to concatenate patterns,
10  *, +, ? and | are used for 0 or more occurences,
11  1 or more occurences, 0 or 1 occurence and
12  choise between alternatives, respectively.
13  )***
14  *****************************************
15  fmod PATTERN is
16
17    protecting OBJ .
18
19    sort Pattern .
20    subsorts Msg < Pattern .
21
22    op _::_ : Pattern Pattern -> Pattern [ctor assoc prec 54] .
23    op _*   : Pattern -> Pattern [ctor prec 53] .
24    op _+   : Pattern -> Pattern [ctor prec 53] .
25    op _?   : Pattern -> Pattern [ctor prec 53] .
26    op _|_  : Pattern Pattern -> Pattern [ctor prec 55] .
27
28    op _where_ : Pattern Expr -> Pattern [ctor prec 56] .
29    op scope(_)_endscope : List Pattern -> Pattern [ctor prec 53] .
30  endfm
31
32
33  *****************************************
34  ***(
35  Set of state names. Used by the DFA and
36  NFA modules
37  )***
38  *****************************************
39  fmod STATE-SET is
40
41    protecting STRING .
42    sort StateSet .
43
44    subsort String < StateSet .
45    op emptyStateSet : -> StateSet .
46    op _,_ : StateSet StateSet ->
47      StateSet [ctor assoc comm id: emptyStateSet] .
48
49    op _in_ : String StateSet -> Bool .
50
51    var SS : StateSet .
52    vars S1 S2 : String .
```

```
53    eq S1, S1 = S1 .
54
55    eq S1 in emptyStateSet = false .
56    eq S1 in S2, SS = S1 == S2 or S1 in SS .
57  endfm
58
59
60  fmod TOKEN is
61    protecting OBJ .
62    protecting MSG-LIST .
63
64    sort Token .
65    subsort Msg < Token .
66    op epsilon : -> Token .
67    op start : -> Token .
68
69    sort TokenList .
70    subsort MsgList < TokenList .
71
72  endfm
73
74
75
76  *****************************************
77  ***
78  *** Non-deterministic automaton
79  ***
80  *****************************************
81  fmod NFA is
82
83    protecting PATTERN .
84    protecting INT .
85    protecting STRING .
86    protecting TOKEN .
87
88    sort NFA .
89    sorts Transition TransitionSet NFA-State .
90    subsort NFA-State < NFA .
91    subsort Transition < TransitionSet .
92
93    op emptyTransitionSet : -> TransitionSet .
94    op __ : TransitionSet TransitionSet ->
95      TransitionSet [ctor assoc comm id: emptyTransitionSet] .
96
97    op emptyNFA : -> NFA .
98    op MakeNFA : Pattern String String List List -> NFA .
99    op MakeNFA : Pattern -> NFA .
100   op __ : NFA NFA -> NFA [ctor assoc comm id: emptyNFA] .
101
102   *** Transition over a given token to a state
103   *** identified by its name as a string.
104   *** The state from which the transition originates
105   *** is implicitly given, since the transitions are
106   *** contained within the states.
```

```
107     op _->_ : Token String -> Transition .
108
109     *** State with name, accepting flag, a set of
110     *** transitions to other states, a list of
111     *** variables in scope and a list of where
112     *** conditions.
113     op {State:_, Accepting:_, Transitions:_, InScope:_, WhereCondition:_} :
114       String Bool TransitionSet List List -> NFA-State .
115
116     var S1 S2 : String .
117     var NFA : NFA .
118     var B : Bool .
119     var TS : TransitionSet .
120     vars L L2 : List .
121     vars W1 W2 : List .
122     var E : Expr .
123
124     var  T : Token .
125     vars P1 P2 : Pattern .
126     var  I : Int .
127     vars NAME NEXTSTATE :  String .
128
129     op _in_ : String NFA -> Bool .
130     eq S1 in emptyNFA = false .
131     eq S1 in {State: S2, Accepting: B, Transitions: TS, InScope:
132       L, WhereCondition: W1} NFA =
133       S1 == S2 or S1 in NFA .
134
135
136     ***(
137     MakeNFA takes a sub-pattern, a name and the
138     name of the state that comes after the sub-pattern
139     as its parameters. The first state that is made by
140     a call to MakeNFA, will be named NAME + "1".
141     NEXTSTATE is the name of the state following this pattern.
142     If NEXTSTATE = "", then the accpeting flag will be set
143     to true, since this is the last state in the automaton.
144     ***)
145
146     eq MakeNFA(P1) = MakeNFA(P1, "", "", nil, nil) .
147
148     *** The current pattern is a single token
149     eq MakeNFA(T, NAME, NEXTSTATE, L, W1) =
150       {State: NAME + "1", Accepting: false, Transitions:
151       (T -> NAME + "2"), InScope: L, WhereCondition: W1
152       }
153       {State: NAME + "2", Accepting: NEXTSTATE == "", Transitions:
154       if NEXTSTATE =/= "" then
155          (epsilon -> NEXTSTATE)
156       else
157          emptyTransitionSet
158       fi,
159       InScope: L, WhereCondition: W1
160       } .
```

```
161
162   *** Repetition (*)
163   eq MakeNFA(P1 *, NAME, NEXTSTATE, L, W1) =
164     {State: NAME + "1", Accepting: false, Transitions:
165     (epsilon -> NAME + "3")
166     (epsilon -> NAME + "21"),
167      InScope: L, WhereCondition: W1
168     }
169     MakeNFA(P1, NAME + "2", NAME + "3", L, W1)
170     {State: NAME + "3", Accepting: NEXTSTATE == "", Transitions:
171     (epsilon -> NAME + "1")
172     if NEXTSTATE =/= "" then
173        (epsilon -> NEXTSTATE)
174     else
175        emptyTransitionSet
176     fi,
177     InScope: L, WhereCondition: W1
178     } .
179
180
181   *** Forced repetition (+)
182   eq MakeNFA(P1 +, NAME, NEXTSTATE, L, W1) =
183     {State: NAME + "1", Accepting: false, Transitions:
184      (epsilon -> NAME + "21"),
185      InScope: L, WhereCondition: W1
186     }
187     MakeNFA(P1, NAME + "2", NAME + "3", L, W1)
188     {State: NAME + "3", Accepting: false, Transitions:
189      (epsilon -> NAME + "41"),
190      InScope: L, WhereCondition: W1
191     }
192     MakeNFA(P1 *, NAME + "4", NAME + "5", L, W1)
193     {State: NAME + "5", Accepting:  NEXTSTATE == "", Transitions:
194     if NEXTSTATE =/= "" then
195        (epsilon -> NEXTSTATE)
196     else
197        emptyTransitionSet
198     fi,
199      InScope: L, WhereCondition: W1
200     } .
201
202
203   *** Zero or one (?)
204   eq MakeNFA(P1 ?, NAME, NEXTSTATE, L, W1) =
205     {State: NAME + "1", Accepting: false, Transitions:
206      (epsilon -> NAME + "21")
207      (epsilon -> NAME + "3"),
208      InScope: L, WhereCondition: W1
209     }
210     MakeNFA(P1, NAME + "2", NAME + "3", L, W1)
211     {State: NAME + "3", Accepting:  NEXTSTATE == "", Transitions:
212     if NEXTSTATE =/= "" then
213        (epsilon -> NEXTSTATE)
214     else
```

```
215         emptyTransitionSet
216       fi,
217        InScope: L, WhereCondition: W1
218       } .
219
220     *** Alternative
221     eq MakeNFA(P1 | P2, NAME, NEXTSTATE, L, W1) =
222       {State: NAME + "1", Accepting: false, Transitions:
223       (epsilon -> NAME + "21")
224       (epsilon -> NAME + "31"),
225        InScope: L, WhereCondition: W1
226       }
227       MakeNFA(P1, NAME + "2", NAME + "4", L, W1)
228       MakeNFA(P2, NAME + "3", NAME + "4", L, W1)
229       {State: NAME + "4", Accepting: NEXTSTATE == "", Transitions:
230       if NEXTSTATE =/= "" then
231          (epsilon -> NEXTSTATE)
232       else
233          emptyTransitionSet
234       fi,
235        InScope: L, WhereCondition: W1
236       } .
237
238     *** Concatenation
239     eq MakeNFA(P1 :: P2, NAME, NEXTSTATE, L, W1) =
240       {State: NAME + "1", Accepting: false, Transitions:
241       (epsilon -> NAME + "21"),
242        InScope: L, WhereCondition: W1
243       }
244       MakeNFA(P1, NAME + "2", NAME + "3", L, W1)
245       {State: NAME + "3", Accepting: false, Transitions:
246       (epsilon -> NAME + "41"),
247        InScope: L, WhereCondition: W1
248       }
249       MakeNFA(P2, NAME + "4", NAME + "5", L, W1)
250       {State: NAME + "5", Accepting: NEXTSTATE == "", Transitions:
251       if NEXTSTATE =/= "" then
252          (epsilon -> NEXTSTATE)
253       else
254          emptyTransitionSet
255       fi,
256        InScope: L, WhereCondition: W1
257       } .
258
259
260     *** Scope:
261     eq MakeNFA(scope( L ) P1 endscope, NAME, NEXTSTATE, L2, W1) =
262       {State: NAME + "1", Accepting: false, Transitions:
263       (epsilon -> NAME + "21"),
264        InScope: L2, WhereCondition: W1
265       }
266       MakeNFA(P1, NAME + "2", NAME + "3", L L2, W1)
267       {State: NAME + "3", Accepting: NEXTSTATE == "", Transitions:
268        if NEXTSTATE =/= "" then
```

```
269         (epsilon -> NEXTSTATE)
270       else
271         emptyTransitionSet
272       fi,
273        InScope: L2, WhereCondition: W1
274       } .
275
276     *** Where condition:
277     eq MakeNFA(P1 where E, NAME, NEXTSTATE, L, W1) =
278       {State: NAME + "1", Accepting: false, Transitions:
279       (epsilon -> NAME + "21"),
280        InScope: L, WhereCondition: W1
281       }
282       MakeNFA(P1, NAME + "2", NAME + "3", L, W1 E)
283       {State: NAME + "3", Accepting: NEXTSTATE == "", Transitions:
284        if NEXTSTATE =/= "" then
285          (epsilon -> NEXTSTATE)
286       else
287          emptyTransitionSet
288       fi,
289        InScope: L, WhereCondition: W1
290       } .
291
292   endfm
293
294
295   ******************************
296   ***
297   *** NFA matching of regular expression
298   ***
299   ******************************
300   fmod NFA-MATCH is
301     protecting NFA .
302     protecting STRING .
303     protecting STATE-SET .
304
305     op Match : TokenList NFA -> Bool .
306     op Match : TokenList NFA Bool -> Bool .
307     op Match : TokenList NFA String Bool -> Bool .
308     op Match : TokenList NFA String Bool StateSet BindingSet -> Bool .
309
310     op _prs_ : TokenList NFA -> Bool .
311
312     var TL : TokenList .
313     vars T1 T2 : Token .
314     vars S S2 : String .
315     var NFA : NFA .
316     var B : Bool .
317     var TS : TransitionSet .
318     var T : Token .
319     var PRS : Bool .
320     var L : List .
321     var SS : StateSet .
322
```

```
323   var P : Pattern .
324   vars Q1 Q2 FROM1 FROM2 TO1 TO2 : Oid .
325   var BS : BindingSet .
326   vars D1 D2 : Data .
327   vars PLIST1 PLIST2 : List .
328   var X : DataVariable .
329   var I : Int .
330   vars P1 P2 E : Expr .
331   var W1 : List .
332
333   eq TL prs NFA = Match(TL, NFA, true) .
334
335   eq Match(TL, NFA) = Match(TL, NFA, false) .
336   eq Match(TL, NFA, PRS) = Match(TL, NFA, "1", PRS,
337     emptyStateSet, emptyBindingSet) .
338
339   eq Match(nil, NFA, S, PRS, SS, BS) =
340      if PRS or IsAccepting(S, NFA) then
341      true
342      else
343      Match(epsilon, NFA, S, PRS, SS, BS)
344      fi .
345
346   eq Match(T1 @ TL, {State: S, Accepting: B, Transitions: TS, InScope: L,
347      WhereCondition: W1} NFA, S, PRS, SS, BS) =
348      *** If we have already visited this state without eating
349      *** any input, or there are no more transitions originating
350      *** from it that we have not yet tried, we return false:
351      if S in SS or TS == emptyTransitionSet then
352       false
353      else
354      if T1 =/= epsilon then
355        if stateName(FindTransition(T1, TS, BS, L, W1)) =/= "" then
356         if Match(TL, {State: S, Accepting: B, Transitions:
357            TS, InScope: L, WhereCondition: W1} NFA,
358            stateName(FindTransition(T1, TS, BS, L, W1)), PRS,
359            emptyStateSet, bindingSet(FindTransition(T1, TS, BS,
360            L, W1)))
361         then
362           true
363         else
364           Match(T1 @ TL, {State: S, Accepting: B, Transitions:
365             RemoveTransition(T1,stateName(FindTransition(
366             T1, TS, BS, L, W1)), TS,
367             bindingSet(FindTransition(T1, TS, BS, L, W1)), L),
368             InScope: L, WhereCondition: W1} NFA, S, PRS, SS, BS)
369         fi
370        else
371        *** There are no transitions in the current
372        *** state S over token T1 - we check if there
373        *** is an epsilon-transition:
374        if stateName(FindTransition(epsilon, TS, BS, L, W1))
375         =/= "" then
376         if Match(T1 @ TL, {State: S, Accepting: B,
```

```
377            Transitions: TS, InScope: L,
378            WhereCondition: W1} NFA,
379            stateName(FindTransition(epsilon, TS, BS, L, W1)),
380            PRS, (S, SS),
381            bindingSet(FindTransition(epsilon, TS, BS, L, W1)))
382         then
383           true
384         else
385           *** We tried an epsilon-transition that did not work,
386           *** but there might be more transitions on the same
387           *** token.
388           *** We remove the transition we just tried, and
389           *** continue the checking:
390           Match(T1 @ TL, {State: S, Accepting: B, Transitions:
391           RemoveTransition(epsilon,stateName(
392             FindTransition(epsilon, TS, BS, L, W1)), TS,
393             bindingSet(FindTransition(epsilon, TS, BS, L, W1)), L),
394           InScope: L, WhereCondition: W1} NFA, S, PRS, SS,BS)
395         fi
396        else
397          *** There are noe epsilon transitions either
398          false
399        fi
400        fi
401      else *** T1 == epsilon:
402        if stateName(FindTransition(epsilon, TS, BS, L, W1)) =/= ""
403        then
404         if Match(nil, {State: S, Accepting: B, Transitions:
405            TS, InScope: L, WhereCondition: W1} NFA,
406            stateName(FindTransition(epsilon, TS, BS, L, W1)),
407            PRS, (S, SS),
408            bindingSet(FindTransition(epsilon, TS, BS, L, W1)))
409         then
410           true
411         else
412           Match(epsilon, {State: S, Accepting: B, Transitions:
413             RemoveTransition(epsilon,stateName(
414              FindTransition(epsilon, TS, BS, L, W1)), TS,
415              bindingSet(FindTransition(epsilon, TS, BS, L, W1)), L),
416             InScope: L, WhereCondition: W1} NFA, S, PRS, SS, BS)
417         fi
418        else
419          false
420        fi
421       fi
422      fi .
423
424
425   *** Is the current state accepting?
426   op IsAccepting : String NFA -> Bool .
427   eq IsAccepting(S, {State: S, Accepting: B,
428      Transitions: TS, InScope: L,
429      WhereCondition: W1} NFA) = B .
430
```

```
431
432     op RemoveTransition : Token String TransitionSet
433       BindingSet List -> TransitionSet .
434     eq RemoveTransition(T1, S, (T1 -> S) TS, BS, L) = TS .
435
436     eq RemoveTransition(msg (Q1(PLIST1)) from FROM1 to TO1, S,
437        ((msg (Q2 (PLIST2)) from FROM2 to TO2) -> S) TS, BS, L) =
438        if Q1 == Q2 and FROM1 == FROM2 and TO1 == TO2 and
439        bool(ParameterCheck(PLIST1, PLIST2, BS, L))
440        then
441        TS
442        else
443        ((msg (Q2 (PLIST2)) from FROM2 to TO2) -> S)
444        RemoveTransition(msg (Q1(PLIST1))
445          from FROM1 to TO1, S, TS, BS, L)
446        fi .
447
448     eq RemoveTransition((Q1(PLIST1)), S,
449        ((Q2(PLIST2)) -> S) TS, BS, L) =
450        if Q1 == Q2 and bool(ParameterCheck(
451          PLIST1, PLIST2, BS, L))
452        then
453        TS
454        else
455        ((Q2(PLIST2)) -> S)
456        RemoveTransition((Q1(PLIST1)), S, TS, BS, L)
457        fi .
458
459     eq RemoveTransition(T1, S, TS, BS, L) = TS [otherwise] .
460
461
462     *** Returns an NFA state that can be reached
463     *** via a transition over a given token
464     *** in a transitionsete. If there is no
465     *** such transition, the an empty transition
466     *** set is returned. The final two parameter lists
467     *** are the variables in scope and the where
468     *** condition that must be true for the
469     *** the transition.
470     op FindTransition : Token TransitionSet
471        BindingSet List List -> StateNameAndBindingSet .
472     eq FindTransition(T1, emptyTransitionSet, BS, L, W1) =
473        ( "" ; scope(BS, L) ) .
474
475     eq FindTransition(msg (Q1(PLIST1)) from FROM1 to TO1,
476        ((msg (Q2 (PLIST2)) from FROM2 to TO2) -> S) TS, BS, L, W1) =
477        if Q1 == Q2 and FROM1 == FROM2 and TO1 == TO2 and
478         bool(ParameterCheck(PLIST1, PLIST2, BS, L)) and
479         CheckCondition(W1, bindingSet(ParameterCheck(
480           PLIST1, PLIST2, BS, L)))
481        then
482        ( S ; bindingSet(ParameterCheck(PLIST1, PLIST2, BS, L)) )
483        else
484        FindTransition(msg (Q1(PLIST1)) from FROM1 to TO1, TS, BS, L, W1)
```

```
485        fi .
486
487
488     eq FindTransition((Q1(PLIST1)), ((Q2 (PLIST2)) -> S) TS, BS, L, W1) =
489        if Q1 == Q2 and bool(ParameterCheck(PLIST1, PLIST2, BS, L)) and
490         CheckCondition(W1, bindingSet(ParameterCheck(
491           PLIST1, PLIST2, BS, L)))
492        then
493        ( S ; bindingSet(ParameterCheck(PLIST1, PLIST2, BS, L)) )
494        else
495        FindTransition((Q1(PLIST1)), TS, BS, L, W1)
496        fi .
497
498
499     eq FindTransition(T1, (T2 -> S) TS, BS, L, W1) = if T1 == T2 then
500       ( S ; scope(BS, L) )
501        else
502        FindTransition(T1, TS, BS, L, W1)
503        fi [otherwise] .
504
505
506     sort BoolAndBindingSet .
507     sort DataAndBindingSet .
508     sort StateNameAndBindingSet .
509     op (_;_) : Bool BindingSet -> BoolAndBindingSet .
510     op (_;_) : Data BindingSet -> DataAndBindingSet .
511     op (_;_) : String BindingSet -> StateNameAndBindingSet .
512
513     op bool : BoolAndBindingSet -> Bool .
514     op bindingSet : BoolAndBindingSet -> BindingSet .
515     op bindingSet : DataAndBindingSet -> BindingSet .
516     op bindingSet : StateNameAndBindingSet -> BindingSet .
517     op stateName : StateNameAndBindingSet -> String .
518     op data : DataAndBindingSet -> Data .
519
520     eq bool(( B ; BS )) = B .
521     eq bindingSet(( B ; BS )) = BS .
522     eq bindingSet(( D1 ; BS )) = BS .
523     eq bindingSet(( S ; BS )) = BS .
524     eq stateName(( S ; BS )) = S .
525     eq data(( D1 ; BS )) = D1 .
526
527     op CheckCondition : List BindingSet -> Bool .
528     eq CheckCondition(nil, BS) = true .
529     eq CheckCondition(E W1, BS) =
530        evalTest(E, BS) and evalB(E, BS) and
531        CheckCondition(W1, BS) .
532
533     op scope : BindingSet List -> BindingSet .
534     eq scope(emptyBindingSet, L) = emptyBindingSet .
535     eq scope(BS, nil) = emptyBindingSet .
536     eq scope(binding(D1, D2) BS, L) =
537        if D1 in L then
538        binding(D1, D2) scope(BS, L)
```

```
539        else
540        scope(BS, L)
541        fi .
542
543     op ParameterCheck : List List BindingSet List -> BoolAndBindingSet .
544
545     eq ParameterCheck(nil, nil, BS, L) = (true ; scope(BS, L)) .
546
547     eq ParameterCheck(P1 PLIST1, P2 PLIST2, BS, L) =
548        if P1 == data(evalWrapper(P2, scope(BS, L), P1))
549        then
550        ((true and
551         bool(ParameterCheck(PLIST1, PLIST2,
552         bindingSet(evalWrapper(P2, scope(BS, L), P1)), L)))  ;
553         (bindingSet(ParameterCheck(PLIST1, PLIST2,
554         bindingSet(evalWrapper(P2, scope(BS, L), P1)), L))))
555        else
556        (false ; scope(BS, L))
557       fi .
558
559
560     op evalWrapper : Expr BindingSet Expr -> DataAndBindingSet .
561     eq evalWrapper(P2, BS, P1) =
562        if evalTest(P2, BS) == false then
563             (eval(P2, addBinding(P2, P1, BS)) ; addBinding(P2, P1, BS))
564        else
565             *** The binding allready exists, or no variable
566             *** in parameter
567             (eval(P2, BS) ; BS)
568        fi .
569
570
571   endfm
572
573
574   ******************************************
575   ***
576   *** Deterministic automaton
577   ***
578   ******************************************
579   fmod DFA is
580
581       protecting NFA .
582       protecting STATE-SET .
583       sort DFA .
584
585
586       sorts DFA-State, DFA-Transition .
587       subsort DFA-State < DFA .
588       subsort DFA-Transition < Transition .
589
590       op emptyDFA : -> DFA .
591       op __ : DFA DFA -> DFA [ctor assoc comm id: emptyDFA] .
592       op start : -> Token .
```

```
593
594     *** A DFA state has name of sort StateSet,
595     *** since it represents a set of NFA states.
596     op {State:_, Accepting:_, Transitions:_} :
597       StateSet Bool TransitionSet -> DFA-State .
598     op _->_ : Token StateSet -> DFA-Transition .
599
600
601     vars S1 S2 : String .
602     var NFA : NFA .
603     var B : Bool .
604     var T : Token .
605     var T2 : Token .
606     var TS : TransitionSet .
607     var SS : StateSet .
608     var SS2 :  StateSet .
609     var TS2 : TransitionSet .
610     vars B2 B3 : Bool .
611     var RESULT-SO-FAR : DFA .
612     var DS : DFA-State .
613     var DFA : DFA .
614
615     op MakeDFA : NFA -> DFA .
616     eq MakeDFA(emptyNFA) = emptyDFA .
617     eq MakeDFA(NFA) = RenameTransitions(
618        {State: "START", Accepting: false, Transitions: start -> "1"}
619        SubsetConstruction(NFA), NFA) .
620
621     *** The subset construction is used to transform
622     *** an NFA to a DFA
623     op SubsetConstruction : NFA -> DFA .
624     eq SubsetConstruction(NFA) =
625        SubsetStart(NFA) Subset2(SubsetStart(NFA),
626        NFA, SubsetStart(NFA)) .
627
628     *** Start of subset construction
629     op SubsetStart : NFA -> DFA .
630     eq SubsetStart({State: "1", Accepting: B, Transitions: TS} NFA) =
631       MakeDFAState(eClosure("1", {State: "1",
632       Accepting: B, Transitions: TS} NFA),
633       {State: "1", Accepting: B, Transitions: TS} NFA) .
634
635     *** "Main function" for the subset construction. Takes the
636     *** current state, the NFA and a set of DFA states that
637     *** have been generated this far as its parameters.
638     op Subset2 : DFA-State NFA DFA -> DFA .
639     eq Subset2({State: SS, Accepting: B, Transitions: (T -> S1) TS},
640        NFA, RESULT-SO-FAR) =
641
642         *** Makes a DFA-state from the first transition,
643         *** if it does not already exist.
644        if MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA)
645           in RESULT-SO-FAR
646        then
```

```
647             emptyDFA
648           else
649               MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA)
650           fi
651
652            *** Recursive call for the remaining transitions in
653            *** this state, if any.
654            Subset2({State: SS, Accepting: B, Transitions: TS},
655                NFA, RESULT-SO-FAR
656                MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA))
657
658            *** Recursive call for the first new state, if it
659            *** does not already exist.
660            if MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA)
661              in RESULT-SO-FAR
662            then
663              emptyDFA
664            else
665              Subset2(MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA),
666               NFA, RESULT-SO-FAR
667               MakeDFAState(eClosures(T, (T -> S1) TS, NFA), NFA))
668            fi
669        .
670
671        eq Subset2({State: SS, Accepting: B, Transitions:
672           emptyTransitionSet}, NFA, DFA) =
673           emptyDFA .
674
675        *** Removes all transitions over a given token
676        op RemoveTransitions : Token TransitionSet -> TransitionSet .
677        eq RemoveTransitions(T, emptyTransitionSet) = emptyTransitionSet .
678        eq RemoveTransitions(T, (T2 -> S1) TS) =
679           if T == T2 then emptyTransitionSet else (T2 -> S1) fi
680           RemoveTransitions(T, TS) .
681
682
683        *** epsilon closure for every state that can be reached
684        *** from transitions in a transition set over a given token:
685        op eClosures : Token TransitionSet NFA -> StateSet .
686        eq eClosures(T, emptyTransitionSet, NFA) = emptyStateSet .
687        eq eClosures(T, (T2 -> S1) TS, NFA) =
688           if T == T2 then eClosure(S1, NFA) else emptyStateSet fi,
689           eClosures(T, TS, NFA) .
690
691        *** epsilon closure for a given state
692        op eClosure : String NFA -> StateSet .
693        op eClosure : String NFA StateSet -> StateSet .
694
695        eq eClosure(S1, {State: S1, Accepting: B, Transitions: TS} NFA) =
696           eClosure(S1, {State: S1, Accepting: B, Transitions: TS} NFA,
697            emptyStateSet) .
698
699        eq eClosure(S1, {State: S1, Accepting: B, Transitions:
700          emptyTransitionSet} NFA, SS) = S1 .
```

```
701
702      eq eClosure(S1, {State: S1, Accepting: B, Transitions:
703       (T -> S2) TS} NFA, SS) =
704       S1,
705       if T == epsilon then
706         if not S2 in SS, S1 then
707           S2, eClosure(S2, ({State: S1, Accepting: B, Transitions:
708             (T -> S2) TS} NFA), (SS, S1, S2))
709         else
710           emptyStateSet
711         fi
712       else
713         emptyStateSet
714       fi,
715       eClosure(S1, {State: S1, Accepting: B, Transitions: TS} NFA, SS,
716         S1,
717         if T == epsilon then
718           if not S2 in SS, S1 then
719             S2, eClosure(S2, ({State: S1, Accepting: B, Transitions:
720               (T -> S2) TS} NFA), (SS, S1, S2))
721           else
722             emptyStateSet
723           fi
724         else
725           emptyStateSet
726         fi
727       ) .
728
729      ceq eClosure(S1, NFA, SS) = emptyStateSet if not S1 in NFA .
730
731
732      *** Makes a DFA state from a set of NFA state names
733      op MakeDFAState : StateSet NFA -> DFA-State .
734      eq MakeDFAState(SS, NFA) =
735         {State: SS, Accepting: AnyAccepting(SS, NFA), Transitions:
736          RemoveEpsilonTransitions(GetTransitions(SS, NFA))} .
737
738      *** Checks if there is at least one accepting state in a
739      *** given state set.
740      op AnyAccepting : StateSet NFA -> Bool .
741      eq AnyAccepting( (S1, SS), {State: S1, Accepting: B, Transitions: TS}
742         NFA) =
743        B or AnyAccepting(SS, NFA) .
744      eq AnyAccepting(emptyStateSet, NFA) = false .
745
746      op GetTransitions : StateSet NFA -> TransitionSet .
747      eq GetTransitions( (S1, SS), {State: S1, Accepting: B,
748         Transitions: TS}
749         NFA) =
750          (TS GetTransitions(SS, NFA)) .
751      eq GetTransitions(emptyStateSet, NFA) = emptyTransitionSet .
752
753      op RemoveEpsilonTransitions : TransitionSet -> TransitionSet .
754      eq RemoveEpsilonTransitions((T -> S1) TS) =
```

```
755         if T =/= epsilon then (T -> S1) else emptyTransitionSet fi
756         RemoveEpsilonTransitions(TS) .
757      eq RemoveEpsilonTransitions(emptyTransitionSet) =
758         emptyTransitionSet .
759
760      *** In a DFA, two equal states are not allowed:
761      eq DS DS = DS .
762
763      *** Checks if a state already exists in a given
764      *** DFA
765      op _in_ : DFA-State DFA -> Bool .
766      eq DS in emptyDFA = false .
767      eq {State: SS, Accepting: B, Transitions: TS} in
768         {State: SS2, Accepting: B2, Transitions: TS2} DFA =
769         SS == SS2 and B == B2 and TS == TS2 or
770         {State: SS, Accepting: B, Transitions: TS} in DFA .
771
772
773      *** Renames all transitions in a DFA so that they point
774      *** to the e-closure of the state(s) that they
775      *** originally pointed to:
776      op RenameTransitions : DFA NFA -> DFA .
777      op RenameTransitions : TransitionSet NFA -> TransitionSet .
778
779      eq RenameTransitions(emptyDFA, NFA) = emptyDFA .
780
781      eq RenameTransitions(emptyTransitionSet, NFA) = emptyTransitionSet .
782
783      eq RenameTransitions({State: SS, Accepting: B,
784         Transitions: TS} DFA, NFA) =
785         {State: SS, Accepting: B, Transitions:
786         RenameTransitions(TS, NFA)}
787         RenameTransitions(DFA, NFA) .
788
789      eq RenameTransitions((T -> S1) TS, NFA) =
790         (T -> eClosures(T, (T -> S1) TS, NFA))
791         RenameTransitions(RemoveTransitions(T, TS), NFA) .
792
793  endfm
794
795
796  ****************************************
797  ***                                  ***
798  *** Module for DFA matching          ***
799  ***                                  ***
800  ****************************************
801  fmod REGEXP is
802
803      protecting DFA .
804      protecting STRING .
805      protecting PARAMETERS .
806
807      var P : Pattern .
808      var TL : TokenList .
```

```
809     vars T1 T2 : Token .
810     vars Q1 Q2 FROM1 FROM2 TO1 TO2 : Qid .
811     var SS : StateSet .
812     var DFA : DFA .
813     var B : Bool .
814     var TS : TransitionSet .
815     var T : Token .
816     var PRS : Bool .
817     var BS : BindingSet .
818     vars D1 D2 : Data .
819     vars PLIST1 PLIST2 : List .
820     var X : DataVariable .
821     var I : Int .
822     vars P1 P2 : Expr .
823
824     *** CompileRE is used to compile a regular expression into
825     *** the corresponding DFA. This function can be used as an
826     *** interface to the outside.
827     op CompileRE : Pattern -> DFA .
828     eq CompileRE(P) = MakeDFA(MakeNFA(P)) .
829
830     *** Standard regular matching:
831     op Match : TokenList DFA -> Bool .
832     op Match : TokenList DFA Bool -> Bool .
833     op Match : TokenList DFA StateSet Bool -> Bool .
834     op Match : TokenList DFA StateSet Bool BindingSet -> Bool .
835
836     *** Match of prefix of regular sequence:
837     op _prs_ : TokenList DFA -> Bool .
838     eq TL prs DFA = Match(TL, DFA, true) .
839
840     eq Match(TL, DFA) = Match(TL, DFA, false) .
841     eq Match(TL, {State: "START", Accepting: B,
842        Transitions: (start -> SS)} DFA, PRS)  =
843        Match(TL, DFA, SS, PRS, emptyBindingSet) .
844
845     *** If there is nothing left of the input, we must
846     *** be in an accepting state if the language
847     *** is to be accepted
848     eq Match(nil, DFA, SS, PRS, BS) = PRS or IsAccepting(SS, DFA) .
849
850     *** Checks if there is a transition from a given state to
851     *** another state over the current token T1:
852     eq Match(T1 @ TL,
853        {State: SS, Accepting: B, Transitions: TS} DFA, SS, PRS, BS) =
854        if stateSet(FindTransition(T1, TS, BS)) =/= emptyStateSet then
855           Match(TL, {State: SS, Accepting: B, Transitions: TS} DFA,
856             stateSet(FindTransition(T1, TS, BS)), PRS,
857             bindingSet(FindTransition(T1, TS, BS)))
858        else
859           false
860        fi .
861
862     *** Is the current state accepting?
```

```
863     op IsAccepting : StateSet DFA -> Bool .
864     eq IsAccepting(SS, {State: SS, Accepting: B,
865       Transitions: TS} DFA) = B .
866
867
868     *** Returns a set of NFA states (that corresponds
869     *** to a DFA state), that can be reached by
870     *** transitions over a given token in a
871     *** given transition set.
872     op FindTransition : Token TransitionSet BindingSet ->
873       StateSetAndBindingSet .
874     eq FindTransition(T1, emptyTransitionSet, BS) =
875       ( emptyStateSet ; BS ) .
876
877     eq FindTransition(msg call (Q1(PLIST1)) from FROM1 to TO1,
878        ((msg call (Q2 (PLIST2)) from FROM2 to TO2) -> SS) TS, BS) =
879        if Q1 == Q2 and FROM1 == FROM2 and
880          TO1 == TO2 and bool(ParameterCheck(PLIST1, PLIST2, BS))
881        then
882          ( SS ; bindingSet(ParameterCheck(PLIST1, PLIST2, BS)) )
883        else
884          FindTransition(msg call (Q1(PLIST1))
885            from FROM1 to TO1, TS, BS)
886        fi .
887
888     eq FindTransition(msg return (Q1(PLIST1)) from FROM1 to TO1,
889        ((msg return (Q2 (PLIST2)) from FROM2 to TO2) -> SS) TS, BS) =
890        if Q1 == Q2 and FROM1 == FROM2 and TO1 == TO2
891          and bool(ParameterCheck(PLIST1, PLIST2, BS))
892        then
893          ( SS ; bindingSet(ParameterCheck(PLIST1, PLIST2, BS)) )
894        else
895          FindTransition(msg return (Q1(PLIST1))
896            from FROM1 to TO1, TS, BS)
897        fi .
898
899     eq FindTransition(msg (Q1(PLIST1)) from FROM1 to TO1,
900        ((msg (Q2 (PLIST2)) from FROM2 to TO2) -> SS) TS, BS) =
901        if Q1 == Q2 and FROM1 == FROM2 and TO1 == TO2
902          and bool(ParameterCheck(PLIST1, PLIST2, BS))
903        then
904          ( SS ; bindingSet(ParameterCheck(PLIST1, PLIST2, BS)) )
905        else
906          FindTransition(msg (Q1(PLIST1)) from FROM1 to TO1, TS, BS)
907        fi .
908     eq FindTransition(T1, (T2 -> SS) TS, BS) =
909        if T1 == T2 then ( SS ; BS ) else
910        FindTransition(T1, TS, BS) fi [otherwise] .
911
912
913
914     sort BoolAndBindingSet .
915     sort DataAndBindingSet .
916     sort StateSetAndBindingSet .
```

```
917       op (_;_) : Bool BindingSet -> BoolAndBindingSet .
918       op (_;_) : Data BindingSet -> DataAndBindingSet .
919       op (_;_) : StateSet BindingSet -> StateSetAndBindingSet .
920
921       op bool : BoolAndBindingSet -> Bool .
922       op bindingSet : BoolAndBindingSet -> BindingSet .
923       op bindingSet : DataAndBindingSet -> BindingSet .
924       op bindingSet : StateSetAndBindingSet -> BindingSet .
925       op stateSet : StateSetAndBindingSet -> StateSet .
926       op data : DataAndBindingSet -> Data .
927
928       eq bool(( B ; BS )) = B .
929       eq bindingSet(( B ; BS )) = BS .
930       eq bindingSet(( D1 ; BS )) = BS .
931       eq bindingSet(( SS ; BS )) = BS .
932       eq stateSet(( SS ; BS )) = SS .
933       eq data(( D1 ; BS )) = D1 .
934
935       *** The first list is the parameters from the history,
936       *** the second is the expression from the automaton
937       op ParameterCheck : List List BindingSet -> BoolAndBindingSet .
938       *** If no parameters are passed, we return true.
939       eq ParameterCheck(nil, nil, BS) = (true ; BS) .
940
941       eq ParameterCheck(P1 PLIST1, P2 PLIST2, BS) =
942          if P1 == data(evalWrapper(P2, BS, P1))
943          then
944            (true ; bindingSet(evalWrapper(P2, BS, P1)))
945          else
946            (false ; BS)
947           fi .
948
949
950       op evalWrapper : Expr BindingSet Expr -> DataAndBindingSet .
951       eq evalWrapper(P2, BS, P1) =
952          if evalTest(P2, BS) == false then
953             (eval(P2, addBinding(P2, P1, BS)) ; addBinding(P2, P1, BS))
954          else
955             *** Binding already exists, or no variable in parameter.
956             (eval(P2, BS) ; BS)
957          fi .
958
959   endfm
```

A.4 Pseudo-random number generator

```
1   ***(
2   Implementation from Numerical Recipes in C, page 278
3   ***)
4   fmod RANDOM is
5
6       protecting NAT .
7
8       op rand : Nat -> Nat .
9
10      op seed : -> Nat .
11      eq seed = 1 . *** May be any positive odd natural number
12
13      ops a m R : -> Nat .
14      eq a = 16807 . *** 7^5
15      eq m = 2147483647 . *** 2 ^ 31 - 1
16      eq R = 8 .
17      var N : Nat .
18      eq rand(N) = (a * N) rem m .
19
20
21  endfm
22
```

A.5 Auxiliary modules

```
1  fmod QID-LIST-EXT is
2      protecting QID-LIST .
3
4      protecting INT .
5
6      op _in_ : Qid QidList -> Bool .
7      op length : QidList -> Int .
8
9      vars Q1 Q2 : Qid .
10     var  QL : QidList .
11
12     eq Q1 in nil = false .
13     eq Q1 in Q2 QL = if Q1 == Q2 then true else Q1 in QL fi .
14
15     eq length(nil) = 0 .
16     eq length(Q1 QL) = 1 + length(QL) .
17
18     op head : QidList -> Qid .
19     eq head(Q1 QL) = Q1 .
20
21     op tail : QidList -> QidList .
22     eq tail(Q1 QL) = QL .
23
24
25     vars I J : Int .
26
27     op swapHead : QidList Int -> QidList .
28     eq swapHead(Q1 QL, I) =
29       removeItem(findItem(Q1 QL, I) insertItem(Q1,
30       removeItem(Q1 QL,I), I), 1) .
31
32     op removeItem : QidList Int -> QidList .
33     op removeItem : QidList Int Int -> QidList .
34     eq removeItem(QL, I) = removeItem(QL, I, 0) .
35     eq removeItem(Q1 QL, I, J) =
36        if I == J then QL else Q1 removeItem(QL, I, J + 1) fi .
37
38     op findItem : QidList Int -> Qid .
39     op findItem : QidList Int Int -> Qid .
40     eq findItem(QL, I) = findItem(QL, I, 0) .
41     eq findItem(Q1 QL, I, J) =
42        if I == J then Q1 else findItem(QL, I, J + 1) fi .
43
44     op insertItem : Qid QidList Int -> QidList .
45     op insertItem : Qid QidList Int Int -> QidList .
46     eq insertItem(Q1, QL, I) = insertItem(Q1, QL, I, 0) .
47     eq insertItem(Q1, nil, I, J) = Q1 .
48     eq insertItem(Q1, Q2 QL, I, J) =
49        if I == J then (Q1 Q2) QL else Q2
50        insertItem(Q1, QL, I, J + 1) fi .
51 endfm
52
```

```
53  fmod PARAMETERS is
54
55      protecting INT .
56      protecting QID .
57      protecting STRING .
58
59      sorts Binding BindingSet .
60      subsort Binding < BindingSet .
61      op __ : BindingSet BindingSet -> BindingSet
62        [ctor assoc comm id: emptyBindingSet] .
63      op emptyBindingSet : -> BindingSet .
64      op binding : Data Data -> Binding [ctor] .
65
66      *** example binding: binding(int(x), int(1)) .
67
68      vars D1 D2 : Data .
69      var BS : BindingSet .
70
71      op _in_ : Data BindingSet -> Bool .
72      eq D1 in binding(D1, D2) BS = true .
73      eq D1 in BS = false [otherwise] .
74
75      op addBinding : Data Data BindingSet -> BindingSet .
76      eq addBinding(D1, D2, BS) = binding(D1, D2) BS .
77
78      op getBinding : Data BindingSet -> Data .
79      eq getBinding(D1, binding(D1, D2) BS) = D2 .
80      eq getBinding(D1, BS) = noBinding [otherwise] .
81
82      sorts Data List Oid Expr .
83      subsort Data < Expr .
84      subsort Expr < List .
85      subsort Qid < Oid .
86
87      op nil : -> List [ctor] .
88      op __ :  List List -> List [ctor assoc id: nil] .
89
90      op null : -> Data [ctor] .
91      op int : Int -> Data [ctor] .
92      op str : String -> Data [ctor] .
93      op bool : Bool -> Data [ctor] .
94      op oid : Qid -> Data [ctor] .
95      op noBinding : -> Data [ctor] .
96
97      sort DataVariable .
98      ops x y z : -> DataVariable .
99
100     op int : DataVariable -> Data [ctor] .
101     op str : DataVariable -> Data [ctor] .
102     op bool : DataVariable -> Data [ctor] .
103
104     *** expressions
105     ops not_ neg_ : Expr -> Expr .
106     ops _+_ _-_ _*_ _/_ _cat_ _%_ : Expr Expr -> Expr .
```

```
107       ops _<_ _<=_ _>_ _>=_ : Expr Expr -> Expr .
108       ops _and_ _or_ _/=_ _=_ : Expr Expr -> Expr .
109
110       *** Calculate values of expressions
111       *** Some of these functions were originally
112       *** written by Marte Arnestad in 2003. Later
113       *** adapted for this thesis:
114       op eval : Expr -> Data .
115       op eval : Expr BindingSet -> Data .
116
117       op evalB : Expr BindingSet -> Bool .
118       op evalI : Expr BindingSet -> Int .
119       op evalS : Expr BindingSet -> String .
120
121       vars Q Q' R : Qid .
122       vars D E : Data .
123       var S : String .
124       vars I J : List .
125       vars X X' : Expr .
126       vars B B' : Bool .
127       vars N N' : Nat .
128       vars C C' : Int .
129       var DV : DataVariable .
130       var L : List .
131
132       op _in_ : List List -> Bool .
133       eq D in nil = false .
134       eq D in E L = D == E or D in L .
135
136       *** data
137       eq eval(D) = eval(D, emptyBindingSet) .
138
139       eq eval(null, BS) = null .
140       eq eval(bool(B), BS) = bool(B) .
141       eq eval(int(C), BS) = int(C) .
142       eq eval(str(S), BS) = str(S) .
143
144       eq eval(int(DV), BS) = getBinding(int(DV), BS) .
145       eq eval(str(DV), BS) = getBinding(str(DV), BS) .
146           eq eval(bool(DV), BS) = getBinding(bool(DV), BS) .
147
148       *** data-bool
149       eq eval(not X, BS) = bool(not evalB(X, BS)) .
150       eq eval(X and X', BS) = bool(evalB(X, BS) and evalB(X', BS)) .
151       eq eval(X or X', BS) = bool(evalB(X, BS) or evalB(X', BS)) .
152
153       eq eval((X > X'), BS) = bool(evalI(X, BS) > evalI(X', BS)) .
154       eq eval((X >= X'), BS) = bool(evalI(X, BS) >= evalI(X', BS)) .
155       eq eval((X < X'), BS) = bool(evalI(X, BS) < evalI(X', BS)) .
156       eq eval((X <= X'), BS) = bool(evalI(X, BS) <= evalI(X', BS)) .
157
158       eq eval(X = X', BS) = bool((eval(X, BS) == eval(X', BS))) .
159       eq eval(X /= X', BS) = bool((eval(X, BS) =/= eval(X', BS))) .
160
```

```
161      *** data-string
162      eq eval(X cat X', BS) = str(evalS(X, BS) + evalS(X', BS) ) .
163
164      *** data-int
165      eq eval((neg X), BS) = int(- evalI(X, BS)) .
166      eq eval((X + X'), BS) = int(evalI((X + X'), BS)) .
167      eq eval((X - X'), BS) = int(evalI((X - X'), BS)) .
168      eq eval((X * X'), BS) = int(evalI((X * X'), BS)) .
169      eq eval((X / X'), BS) = int(evalI((X / X'), BS)) .
170      eq eval((X % X'), BS) = int(evalI((X % X'), BS)) .
171
172      eq evalB(bool(B), BS) = B .
173      eq evalB(not X, BS) = not evalB(X, BS) .
174      eq evalB(X and X', BS) =  evalB(X, BS) and evalB(X', BS) .
175      eq evalB(X or X', BS) = evalB(X, BS) or evalB(X', BS) .
176      eq evalB((X > X'), BS) = evalI(X, BS) > evalI(X', BS) .
177      eq evalB((X >= X'), BS) = evalI(X, BS) >= evalI(X', BS) .
178      eq evalB((X < X'), BS) = evalI(X, BS) < evalI(X', BS) .
179      eq evalB((X <= X'), BS) = evalI(X, BS) <= evalI(X', BS) .
180      eq evalB(X = X', BS) = (eval(X, BS) == eval(X', BS)) .
181      eq evalB(X /= X', BS) = (eval(X, BS) =/= eval(X', BS)) .
182      eq evalB(bool(DV), BS) = evalB(getBinding(bool(DV), BS), BS) .
183
184      eq evalS(str(S), BS) = S .
185      eq evalS(X cat X', BS) = evalS(X, BS) + evalS(X', BS) .
186      eq evalS(str(DV), BS) = evalS(getBinding(str(DV), BS), BS) .
187
188      eq evalI(int(C), BS) = C .
189      eq evalI((neg X), BS) = (- evalI(X, BS)) .
190      eq evalI((X + X'), BS) = evalI(X, BS) + evalI(X', BS) .
191      eq evalI((X - X'), BS) = evalI(X, BS) - evalI(X', BS) .
192      eq evalI((X * X'), BS) = evalI(X, BS) * evalI(X', BS) .
193      eq evalI((X / X'), BS) = evalI(X, BS) quo evalI(X', BS) .
194      eq evalI((X % X'), BS) = evalI(X, BS) rem evalI(X', BS) .
195      eq evalI(int(DV), BS) = evalI(getBinding(int(DV), BS), BS) .
196
197
198      *** evalTest tests if the expression can be
199      *** evaluated with the current binding set.
200      op evalTest : Data -> Bool .
201      op evalTest : Data BindingSet -> Bool .
202
203      eq evalTest(D) = evalTest(D, emptyBindingSet) .
204
205      eq evalTest(null, BS) = true .
206      eq evalTest(bool(B), BS) = true .
207      eq evalTest(int(C), BS) = true .
208      eq evalTest(str(S), BS) = true .
209
210      eq evalTest(int(DV), BS) = getBinding(int(DV), BS) =/= noBinding .
211      eq evalTest(str(DV), BS) = getBinding(str(DV), BS) =/= noBinding .
212      eq evalTest(bool(DV), BS) = getBinding(bool(DV), BS) =/= noBinding .
213
214      eq evalTest(not X, BS) = evalTest(X, BS) .
```

```
215     eq evalTest(X and X', BS) = evalTest(X, BS) and evalTest(X', BS) .
216     eq evalTest(X or X', BS) = evalTest(X, BS) and evalTest(X', BS) .
217
218     eq evalTest((X > X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
219     eq evalTest((X >= X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
220     eq evalTest((X < X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
221     eq evalTest((X <= X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
222
223     eq evalTest(X = X', BS) = evalTest(X, BS) and evalTest(X', BS) .
224     eq evalTest(X /= X', BS) = evalTest(X, BS) and evalTest(X', BS) .
225
226     eq evalTest(X cat X', BS) = evalTest(X, BS) and evalTest(X', BS) .
227
228     eq evalTest((neg X), BS) = evalTest(X, BS) .
229     eq evalTest((X + X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
230     eq evalTest((X - X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
231     eq evalTest((X * X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
232     eq evalTest((X / X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
233     eq evalTest((X % X'), BS) = evalTest(X, BS) and evalTest(X', BS) .
234 endfm
235
236 fmod OBJ is
237
238     protecting META-TERM .
239     protecting QID .
240     protecting QID-LIST .
241     protecting INT .
242     protecting PARAMETERS .
243     sort Msg .
244
245     op invoc_ : Msg -> Msg .
246     op comp_ : Msg -> Msg .
247
248     var Q : Qid .
249     op _() : Qid -> Msg .
250     eq Q() = Q(nil) .
251     op _(_) : Qid List -> Msg .
252
253     op msg_from_to_ : Msg Oid Oid -> Msg .
254     op msg_from_to_label_ : Msg Oid Oid GroundTerm -> Msg .
255     op msg_from_to_ : Qid Qid Qid -> Msg .
256
257     sort Obj .
258     op <_> : Oid -> Obj .
259
260     sort Configuration .
261     subsorts Msg Obj < Configuration .
262     op __ : Configuration Configuration -> Configuration
263             [ctor assoc comm id: none] .
264     op none : -> Configuration [ctor] .
265
266 endfm
267
268
```

```
269 fmod OID-SET is
270
271         protecting OBJ .
272
273         sort OidSet .
274         subsort Oid < OidSet .
275         op none : -> OidSet .
276         op _;_ : OidSet OidSet -> OidSet [ctor assoc comm id: none] .
277
278         var O1  Oid .
279         var OS : OidSet .
280
281         op _in_ : Oid OidSet -> Bool .
282         eq O1 in (O1 ; OS) = true .
283         eq O1 in OS = false [otherwise] .
284
285 endfm
```

A.6 Python code for socket communication

```
1  #!/local/snacks/bin/python2.3 -d
2  # -*- coding: utf-8 -*-
3
4  import socket
5  import pre as re
6  import popen2
7  import os
8  import getopt
9  import sys
10 import thread
11 import string
12 import time
13 import random
14
15
16 objectlocations = {}
17
18 # example objectlocations table:
19 # {"R1": ("pantelleria.ifi.uio.no", 50007),
20 #  "R2": ("pantelleria.ifi.uio.no", 50007),
21 #  "R3": ("pantelleria.ifi.uio.no", 50007),
22 #  "W1": ("pantelleria.ifi.uio.no", 50007),
23 #  "SP": ("pantelleria.ifi.uio.no", 50007),
24 #  "RWServer": ("pantelleria.ifi.uio.no", 50008)}
25
26 # example objectlocations xml file:
27 # <objectlocations>
28 #   <object>
29 #     <name>R1</name>
30 #     <machine>pantelleria.ifi.uio.no</machine>
31 #     <port>5007</port>
32 #   </object>
33 #   <object>
34 #     <name>R2</name>
35 #     ... etc ...
36 # </objectlocations>
37
38 outq = []
39 inq = []
40 maudefile = ""
41 maudemodule = ""
42 xmlfile = ""
43
44 def receive_thread(socket):
45   # receive a message (of max 2048 bytes):
46   data = socket.recv(2048)
47   inq.append(data)
48   socket.close()
49
50 def listen(objectname):
51
52   host, port = objectlocations[objectname]
```

```
53
54    # crete socket
55    serversocket = socket.socket(
56          socket.AF_INET, socket.SOCK_STREAM)
57    serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
58    # bind the socket to the current machine and specified port:
59    serversocket.bind((socket.gethostname(), port))
60    # listen for incomming requests
61    serversocket.listen(5)
62
63    while 1:
64      # accept connections from the outside,
65      # and create a new socket
66      (clientsocket, address) = serversocket.accept()
67      # create new thread, that will handle the communication
68      # over the new socket:
69      thread.start_new_thread(recieve_thread, (clientsocket,))
70
71  def send_thread():
72
73    while 1:
74        for m in outq:
75          if sendmessage(m):
76            outq.remove(m)
77        sleep(100)
78
79  def sendmessage(msg):
80    msg, reciever = msg
81    host, port = objectlocations[reciever]
82
83    retval = 1
84    try:
85      cs = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
86      cs.connect((host, port))
87      cs.send(msg)
88    except:
89      retval = 0
90    cs.close
91    return retval
92
93
94  def maude_thread():
95
96    tempfilename = "tmp" + \
97      str(random.randrange(1000000,9000000,1)) + ".txt"
98    commandfilename = "tmp" + \
99      str(random.randrange(1000000,9000000,1)) + ".maude"
100
101   # the first state must be "init"
102   state = "init"
103   i = 0
104   while i < numberOfRewrites or numberOfRewrites == 0:
105
106     i = i + 1
```

```
107
108       # create temporary command file
109       commandfile = open(commandfilename, "w")
110       commandfile.write("load " + maudefile + " .\n")
111       if maudemodule != "":
112         commandfile.write("select " + maudemodule + " .\n")
113       commandfile.write("rew [1] " + state + " .\n")
114       commandfile.write("q \n")
115       commandfile.close()
116
117       lines = ""
118
119       # read the output from Maude:
120       rc = os.system('maude -interactive -no-prelude' + \
121         commandfilename + ' > ' + tempfilename)
122
123       tempfile=open(tempfilename,'r')
124       for line in tempfile:
125         lines = lines + line
126       tempfile.close()
127
128       # For some reason, Maude will from time to time not
129       # print all the lines; we check if this is the case:
130       if string.find(lines, "Bye.") != -1:
131
132           lines = string.replace(lines, "\n", "")
133           lines = string.replace(lines, "\r", "")
134           lines = string.replace(lines, "\t", "")
135
136           result = re.search(r".*result \[?\w*\]?:\
137             ((.|\n)*)Bye((.|\n)*)$", lines)
138           state = result.group(1)
139
140           # Put new messages from the inq into the configuration:
141           engine = re.search(
142              r"^(?P<enginePre>(.|\n)*Engine\[\s*curTerm:\s*'__\[\s*)\
143               (?P<curTerm>(.|\n)*)(?P<enginePost>\s*\]\s*,\s*curModule:
144               (.|\n)*\s*\])$", state)
145           try:
146             test = engine.group("eginePre")
147           except:
148             engine = re.search(
149               r"^(?P<enginePre>(.|\n)*Engine\[\s*curTerm:\s*) \
150                 (?P<curTerm>(.|\n)*)(?P<enginePost>\s*,\
151                 \s*curModule:(.|\n)*\s*\])$", state)
152
153           newMsgs = ""
154           for m in inq:
155             newMsgs = newMsgs + m
156           inq = []
157
158           if newMsgs != "":
159             if engine.group("curTerm") == "'init.Configuration":
160                 state = engine.group("enginePre") + "'__[" + \
```

```
161             engine.group("curTerm") + newMsgs + "]" + \
162             engine.group("enginePost");
163           print state + "\n\n"
164         else:
165           if not engine.group("curTerm")[0:4] == "'__[":
166             state = engine.group("enginePre") + \
167               "'__[" + engine.group("curTerm") + newMsgs + "]" + \
168               engine.group("enginePost")
169           else:
170             state = engine.group("enginePre") + \
171                       engine.group("curTerm") + newMsgs + \
172                       engine.group("enginePost")
173         print state + "\n\n"
174
175       # remove tab and cr:
176       state = string.replace(state, "\n", "")
177       state = string.replace(state, "\t", "")
178
179       failed = 0
180       while not failed:
181
182         engine = re.search(
183           r"^(?P<enginePre>(.|\n)*Engine\[\s*curTerm:\s*'__\[\s*)\
184             (?P<curTerm>(.|\n)*)(?P<enginePost>\s*\]\s*,\s*curModule:\
185             (.|\n)*\s*\])$", state)
186         try:
187           test = engine.group("eginePre")
188         except:
189           engine = re.search(
190             r"^(?P<enginePre>(.|\n)*Engine\[\s*curTerm:\s*)\
191             (?P<curTerm>(.|\n)*)(?P<enginePost>\s*,\s*curModule:\
192             (.|\n)*\s*\])$", state)
193
194         curTerm = engine.group("curTerm")
195
196         result = re.search(
197             r"^(?P<pre>(.|\n)*)(?P<msg>\s*,\\
198             s*'msg_from_to_\['_`\(_`\)\\
199             [.*\],\s*''(?P<from>\w*)\.Sort,\s*''(?P<to>\w*)\.Sort\])\
200             (?P<post>(.|\n)*)$", curTerm)
201
202         try:
203           msg = string.replace(result.group("msg"), "\r", "")
204           reciever = result.group("to")
205         except:
206           failed = 1
207
208         if not failed:
209           # check to see if the message is destined for another
210           # machine / process or not:
211           if objectlocations[objectname] != objectlocations[reciever]:
212             # to be sent:
213             outq.append((msg, reciever))
214           else:
```

```
215              # local message:
216              inq.append(msg)
217
218             # remove the message from the current state
219             state = engine.group("enginePre") + result.group("pre") + \
220               result.group("post") + engine.group("enginePost")
221     else:
222        # do another iteration of the loop with the same state
223
224
225 # command line arguments:
226 optlist, args = getopt.getopt(sys.argv[1:], '')
227
228 try:
229   objectname = args[0]
230   maudefile = args[1]
231   xmlfile = args[2]
232   try:
233     maudemodule = args[3]
234   except:
235     maudemodule = ""
236   try:
237     numberOfRewrites = int(args[4])
238   except:
239     numberOfRewrites = 0
240 except:
241   print "Usage: ", sys.argv[0], "ObjectName " + \
242     "MaudeFile XmlObjectLocations [Module] [NumberOfRewrites]"
243   sys.exit(1)
244
245 objectlocations = XmlToDictionary(xmlfile)
246
247 myIP = socket.gethostbyname(socket.gethostname())
248
249 # start listener thread
250 thread.start_new_thread(listen, (objectname,))
251
252 # start send thread
253 thread.start_new_thread(send_thread, ())
254
255 # start maude controller thread:
256 thread_start_new_thread(maude_controller, ())
```

Appendix B

Papers

Results from the work with this thesis have been included in two scientific papers:

- *A Run-Time Environment for Concurrent Objects with Asynchronous Methods Calls* [24] is published in the Proceedings of the 5th International Workshop on Rewriting Logic and its Applications (WRLA'04).
- *Toward Reflective Application Testing in Open Environments* [4] is submitted for publication at Norsk Informatikk-Konferanse 2004 (NIK'04).

Both papers are included in their entirety on the following pages.

A run-time environment for concurrent objects with asynchronous methods calls

Einar Broch Johnsen, Olaf Owe, and Eyvind W. Axelsen

Department of Informatics, University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
Email: {einarj,olaf,eyvindwa}@ifi.uio.no

Abstract

A distributed system may be modeled by objects that run concurrently, each with its own processor, and communicate by remote method calls. However objects may have to wait for response to external calls; which can lead to inefficient use of processor capacity or even to deadlock. This paper addresses this limitation by means of asynchronous method calls and conditional processor release points. Although at the cost of additional internal nondeterminism in the objects, this approach seems attractive in asynchronous or unreliable distributed environments. The concepts are illustrated by the small object-oriented language Creol and its operational semantics, which is defined using rewriting logic as a semantic framework. Thus, Creol specifications may be executed with Maude as a language interpreter, which allows an incremental development of the language constructs and their operational semantics supported by testing in Maude. However, for prototyping of highly nondeterministic systems, Maude's deterministic engine may be a limitation to practical testing. To overcome this problem, a rewrite strategy based on a pseudo-random number generator is proposed, providing Maude with nondeterministic behavior.

Key words: Object orientation, asynchronous method calls, operational semantics, rewriting logic, nondeterministic rewrite strategies

1 Introduction

The importance of inter-process communication is rapidly increasing with the development of distributed computing, both over the Internet and over local networks. Object orientation appears as a promising framework for concurrent and distributed systems [20], but object interaction by means of method calls is usually synchronous and therefore less suitable in a distributed setting. Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. In this paper programming constructs for concurrent objects are proposed with an object-oriented design

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

language Creol, based on *processor release points* and *asynchronous method calls*. Processor release points are used to influence the implicit internal control flow in concurrent objects. This reduces time spent waiting for replies to method calls in a distributed environment and allows objects to dynamically change between active and reactive behavior (client and server).

We consider how object-oriented method calls, returning output values in response to input values, can be adapted to the distributed setting. With the *remote procedure call* (RPC) model, an object is brought to life by a procedure call [6]. Control is transferred with the call so there is a master-slave relationship between the caller and the callee. Concurrency is achieved by multiple execution threads, e.g. Hybrid [26] and Java [19]. In Java the interference problem related to shared variables reemerges when threads operate concurrently in the same object, and reasoning about programs in this setting is a highly complex matter [1,11]. Reasoning considerations suggest that all methods should be serialized [9], which is the approach taken by Hybrid. But with serialized methods, the caller must *wait* for the return of a call, blocking for any other activity in the object. In a distributed setting this limitation is severe; delays and instabilities due to distribution may cause considerable waiting. In contrast, message passing is a communication form without transfer of control. For synchronous message passing, as in Ada's Rendezvous mechanism, both sender and receiver must be ready before communication can occur. Method calls may be modeled by pairs of messages, on which the two objects must synchronize [6]. For distributed systems, this synchronization still results in much waiting. In the asynchronous setting, messages may always be emitted regardless of when the receiver accepts the message. Communication by asynchronous message passing is well-known from e.g. the Actor model [2,3]. However, method calls imply an ordering on communication not easily captured in the Actor model.

In this paper, method calls are taken as the communication primitive for concurrent objects and given an operational semantics reflected by pairs of asynchronous messages, allowing message overtaking. The result resembles programming with so-called future variables [8,10,16,28,29]; computation may continue until the return value of the call is explicitly needed in the code. To avoid blocking the object at this point, we propose interleaved method evaluations in objects by defining potential processor release points in method bodies using inner guards. Hence, present activity may be suspended, allowing the object's invoked and enabled methods to compete for the free processor.

The operational semantics of Creol has been defined in rewriting logic [23], which is supported by the executable modeling and analysis tool Maude [13]. Rewriting logic is a logic of concurrent change. A number of concurrency models have been successfully represented in rewriting logic and Maude [23,24], including the ODP computational model [25] and structural operational semantics [17]. We have used rewriting logic and Maude as a tool for development of high-level programming constructs for distributed concurrent objects. As

our aim is to consider constructs for a traditional imperative setting, rewrite rules capture the behavior of the abstract machine, rather than method calls as in the object model of rewriting logic and Maude [13,23]. Our experiments suggest that rewriting logic and Maude provide a well-suited framework for experimentation with language constructs and concurrent environments. However, in order to capture the nondeterminism of distributed systems, Maude's deterministic engine may be a limitation. Therefore, the paper proposes a new rewrite strategy for Maude, based on a pseudo-random number generator. This allows nondeterministic executions, selecting not only the rewrite rule according to the random number, but also where it is applied. The strategy seems well-suited for testing any nondeterministic Maude specification, as several runs of the same specification give rise to different executions.

Paper overview. Section 2 introduces the imperative level of Creol. Section 3 provides an example. Section 4 defines Creol's operational semantics using rewriting logic. Section 5 presents a nondeterministic rewrite strategy for Maude. Section 6 considers related work and Section 7 concludes.

2 Programming Constructs

This section proposes programming constructs for distributed concurrent objects, based on asynchronous method calls and processor release points. Concurrent objects are potentially active, encapsulating execution threads; consequently, elements of basic data types are not considered objects. In this sense, our objects resemble top-level objects in e.g. Hybrid [26]. Objects have explicit identifiers: communication takes place between named objects and object identifiers may be exchanged between objects. All object interaction is by means of method calls. Creol objects are typed by abstract interfaces [21,22]. These resemble CORBA's IDL, but extended with semantic requirements and mechanisms for type control in dynamically reconfigurable systems. The language supports strong typing, e.g. invoked methods are supported by the called object (when not null), and formal and actual parameters match.

In order to focus the discussion on asynchronous method calls and processor release points in method bodies, other language aspects will not be discussed in detail, including inheritance and typing. To simplify the exposition, we assume a common type Data of basic data values which may be passed as arguments to methods, including as subtypes the object identifiers Obj and data types such as Bool. Expressions Expr evaluate to Data. We denote by Var the set of program variables, by Mtd the set of method names, and by Label the set of method call identifiers.

2.1 Classes and Objects

At the programming level, attributes (object variables) and method declarations are organized in classes in a standard way. Objects are dynamically

created instances of classes. The attributes of an object are encapsulated and can only be accessed via the object's methods. Among the declared methods, we distinguish two methods *init* and *run*, which are given special treatment operationally. The *init* method is invoked at object creation to instantiate attributes and may not contain processor release points. After initialization, the *run* method, if provided, is started. Apart from *init* and *run*, declared methods may be invoked by other objects of appropriate interfaces. These methods reflect passive or reactive behavior in the object, whereas *run* reflects active behavior. Object activity is organized around an *external message queue* and an *internal process queue* which contains *pending processes*. Methods need not terminate and, apart from *init*, all methods may be temporarily *suspended* on the internal process queue.

2.2 Asynchronous Methods

An object offers methods to its environment, specified through a number of interfaces. All interaction with the object happens through the methods of its interfaces. In the asynchronous setting method calls can always be emitted, because the receiving object cannot block communication. Method overtaking is allowed in the sense that if methods offered by an object are invoked in one order, the object may react to the invocations in another order. Methods are, roughly speaking, implemented by nested guarded commands $G \longrightarrow C$, to be evaluated in the context of locally bound variables. Guarded commands are treated in detail in Section 2.3.

Due to the possible interleaving of different method executions, the values of an object's instance variables are not entirely controlled by a method instance if it suspends itself before completion. However, a method may create local variables supplementing the object variables. In particular, the values of formal parameters are stored locally, but other local variables may also be declared. Semantically, an instantiated method is a *process*, represented as a pair $\langle GC, L\rangle$ where GC is a (guarded) sequence of commands and $L : \mathsf{Var} \rightarrow \mathsf{Data}$ the local variable bindings. Consider an object o which offers the method

$$\mathbf{op}\ m(\mathbf{in}\ x : \mathsf{Data}\ \mathbf{out}\ y : \mathsf{Data}) == \mathbf{var}\ z : \mathsf{Data} := 0; G \longrightarrow C\ .$$

to the environment. Accepting a call $invoc(l, o', o, m, 2)$ from an object o' adds the pair $\langle G \longrightarrow C, \{label \mapsto l, caller \mapsto o', x \mapsto 2, y \mapsto nil, z \mapsto 0\}\rangle$ to the internal process queue of object o, where pending processes wait for the object processor. An object can have several pending calls to the same method, possibly with different values for local variables. The local variables *label* and *caller* are reserved to identify the call and the caller for the reply, which is emitted at method termination.

An asynchronous method call is made with the command $l!o.m(e)$, where $l \in \mathsf{Label}$ is a unique reference to the call, o an object identifier, m a method name, and e an expression list with the supplied actual parameters. Labels are used to identify replies, and may be omitted if a reply is not explicitly

requested. As no synchronization is involved, process execution may proceed after calling an external method until the return values are needed by the process. To fetch the return values from the queue, say in a variable list x, we ask for the reply to the call: $l?(x)$. If the reply has arrived, return values are assigned to x and execution continues without delay. If no reply to the call has been received, the process must now wait. This interpretation of $l?(x)$ gives the same effect as treating x as a future variable. However, waiting in the asynchronous case can be avoided altogether by introducing processor release points for reply requests. In the case without reply, execution is *suspended*, placing the active process and its local variables on the internal process queue.

Although remote and local calls can be handled operationally in the same way, it is clear that for execution of local calls the calling process must eventually suspend its own execution. In particular, synchronous local calls are given direct access to the object processor. The syntax $o.m(e;x)$ is adopted for synchronous (RPC) method calls, blocking the processor while waiting for the reply. Local calls need not be prefixed by an object identifier.

2.3 A Language with Processor Release Points

In Creol, the control flow inside concurrent objects may be influenced by potential processor release points. These are explicitly declared in method bodies using guarded commands, as introduced by Dijkstra [18], and may be nested within the same local variable scope. When an inner guard evaluates to false during process execution, the remaining process code is suspended to the internal process queue and the processor is released. After processor release, an enabled process from the internal process queue is selected for execution.

Definition 2.1 The type Guard is constructed inductively as follows:

- $wait \in \mathsf{Guard}$ (explicit release)
- $l?(x) \in \mathsf{Guard}$, where $l \in \mathsf{Label}$ and $x \in \mathsf{Var}$
- $\phi \in \mathsf{Guard}$, where ϕ is a boolean expression over local and object variables

Here, *wait* is a construct for explicit release of the processor. The reply guard $l?(x)$ checks whether the reply to a method call has been received, as further execution of a process will often depend on the arrival of a certain reply. If this is the case, $l?(x)$ returns true and instantiates x with the return values. Evaluation of guards is done atomically.

Guarded commands can be *composed* in different ways, reflecting the requirements to the internal control flow in objects. Let GC_1 and GC_2 denote the guarded commands $G_1 \longrightarrow C_1$ and $G_2 \longrightarrow C_2$. Nesting of guards is obtained by sequential composition; in a program statement $GC_1; G_2 \longrightarrow C_2$, the guard G_2 corresponds to a potential inner processor release point. Nondeterministic choice between guarded commands is expressed by $GC_1 \Box GC_2$, which may compute C_1 if G_1 evaluates to true, C_2 if G_2 evaluates to true, and is otherwise suspended. Nondeterministic merge is expressed by $GC_1 \| GC_2$,

Syntactic categories.	*Definitions.*
C in Com	$C ::= \varepsilon \mid x := e \mid GC \mid C_1; C_2 \mid \textbf{new}\ \mathit{classname}(e)$
GC in Gcom	$\mid \textbf{if}\ G\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}$
G in Guard	$\mid \textbf{while}\ G\ \textbf{do}\ C\ \textbf{od}$
x in VarList	$\mid m(e;x) \mid l!m(e) \mid !m(e) \mid l?(x)$
e in ExprList	$\mid o.m(e;x) \mid l!o.m(e) \mid !o.m(e)$
m in Mtd	$GC ::= G \longrightarrow C$
o in Obj	$\mid GC_1 \Box GC_2$
l in Label	$\mid GC_1 \Vert\!\vert GC_2$

Fig. 1. An outline of the syntax for the proposed language Creol, focusing on the main syntactic categories Com of commands and Gcom of guarded commands.

which can be defined by $(GC_1; GC_2)\Box(GC_2; GC_1)$. Ordinary control flow is expressed by **if** and **while** constructs, and assignment to local and object variables is expressed as $x := e$ for a list x of program variables and a list e of expressions. Figure 1 summarizes the language syntax.

With nested processor release points, the processor need not wait actively for replies. This approach is more flexible than future variables: pending processes or new method calls may be evaluated instead of blocking the processor. However, when the reply has arrived, the *continuation* of the original process must compete with the other enabled and pending processes in the internal process queue.

3 Example: The Dining Philosophers

The well-known dining philosophers are now considered in Creol. The example will later be used to experiment with the language interpreter. A butler informs a philosopher of the identity of the philosopher's left neighbor. A philosopher may borrow and return its neighbor's chopstick. Interaction between the philosophers and the butler is restricted by interfaces. This results in a clear distinction between internal methods and methods externally available to other objects typed by so-called *cointerfaces* [21,22]. These express mutual dependency between interfaces, and are declared in the interfaces by means of a **with** construct. Strong typing and cointerfaces guarantee that only philosophers may call the methods *borrowStick* and *returnStick*.

```
interface Phil
begin
  with Phil
    op borrowStick
    op returnStick
end
```

```
interface Butler
begin
  with Phil
    op getNeighbor(out n:Phil)
end
```

In this approach, philosopher objects display both active and reactive behavior. Each philosopher controls one chopstick and must borrow its neighbor's chopstick in order to eat. Thus, philosophers have their internal activity as well as responding to calls from the environment. The standard configuration of the dining philosophers is most easily obtained by means of a single butler.

3.1 Implementing the Philosophers

Philosophers are active objects so the Philosopher class will include a *run* method. This method is defined in terms of several nonterminating internal methods representing different activities within a philosopher: *think*, *eat*, and *digest*. In *run*, the internal methods are invoked asynchronously, and will be interleaved in a nondeterministic and nonterminating manner, illustrating the processor release point construct. All three methods depend on the value of the internal variable *hungry*. The *think* method is a loop which suspends its own evaluation before each iteration, whereas *eat* attempts to grab the object's and the neighbor's chopsticks in order to satisfy the philosopher's hunger. In this case, the philosopher must wait until both chopsticks are available. In order to avoid blocking the object processor, the *eat* method is suspended after asking for the neighbor's chopstick; further processing of the method can first happen when the guard is satisfied. The *digest* method represents the action of becoming hungry. Classes may include class parameters, which become instance attributes bound at object creation, as in Simula. The Philosopher class is defined as follows:

```
class Philosopher(butler: Butler) implements Phil
begin
  var hungry: bool, chopstick: bool, neighbor: Phil
  op init == chopstick := true; hungry := false; butler.getNeighbor(;neighbor) .
  op run == true ⟶ !think ||| true ⟶ !eat ||| true ⟶ !digest .
  op think == not hungry ⟶ ⟨thinking...⟩; wait ⟶ !think .
  op eat == var l : label; hungry ⟶ l!neighbor.borrowStick;
              (chopstick ∧ l?()) ⟶ ⟨eating...⟩; hungry := false;
               !neighbor.returnStick; wait ⟶ !eat .
  op digest == not hungry ⟶ (hungry := true; wait ⟶ !digest) .

 with Phil
  op borrowStick == chopstick ⟶ chopstick := false .
  op returnStick == chopstick := true .
end
```

This implementation favors implicit control of the object's active behavior. Caromel and Rodier argue that facilities for both implicit and explicit control are needed in languages which address concurrent programming [10]. Explicit activity control can be programmed in Creol by using a **while** loop in the *run* method. However in asynchronous distributed systems, we believe that com-

munication introduces so much nondeterminism that explicit control structures quickly lead to program over-specification and possibly to unnecessary active waiting.

In contrast to the active philosophers, butlers are passive. After creating philosophers during initialization, a butler waits for philosophers to request the identity of their neighbors. The code of the butler class is straightforward and therefore omitted here.

4 A Rewriting Logic Semantics for Creol

The operational semantics of the proposed language constructs is given using the semantic framework provided by rewriting logic (RL). A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to local fragments of a state configuration. Rules may be applied in parallel to non-overlapping *subconfigurations*. We assume that RL is known to the reader and present the operational semantics in the syntax of Maude. Rewrite rules will capture the behavior of a Creol abstract machine, and not of the Creol objects as in Maude's object model. Hence, a configuration is a multiset combining Creol objects, messages, queues, and classes. Auxiliary functions are defined in equational logic, and evaluated in between rewrite steps [23]. As usual, the associative and commutative constructor for multisets and the associative constructor for lists are represented by whitespace.

An RL object is a term of the type $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [13]. We adopt this form of presentation and define Creol objects, classes, and external message queues as RL objects. Omitting RL types, a Creol object is represented by an RL object $\langle Id \mid Cl, Pr, PrQ, Lvar, Att, Lcnt \rangle$, where Id is the object identifier, Cl the class name, Pr the active process code, PrQ a multiset of pending processes (see Section 2.2) allowing all kinds of queue orderings, and $Lvar$ and Att the local and object variables, respectively. Finally, $Lcnt$ is the method call identifier corresponding to labels in the language. Thus, the object identifier Id and the local label value provide a globally unique identifier for each method call. External message queues have a name and contain a multiset of unprocessed messages. Each external message queue is associated with one specific Creol object.

Creol classes are represented by RL objects $\langle Cl \mid Att, Ocnt, init, run, Mtds \rangle$, where Cl is the class name, Att a list of attributes, $Ocnt$ the number of objects instantiated of the class, and $Mtds$ a set of methods. When an object needs a method, it is loaded from the $Mtds$ set of its class (overloading and virtual

binding issues connected to inheritance are ignored in this paper).

In RL's object model [23], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid by representing Creol classes in the configuration. The Creol command *new C(args)* will create a new object with a unique object identifier, object variables as listed in the class parameter list with values defined in *args* and in *Att*, and place the code from methods *init* and *run* in *Pr*. Uniqueness of the object identifier is ensured by appending the number *Ocnt* to the class name, and increasing *Ocnt*.

There are four different kinds of rewrite rules:

- *Rules that execute code from the active process:* For every program statement there is at least one rule. For example, the assignment rule for the program $X := E$ binds the value of the expression E to X in either the list of local or object variables.
- *Rules for suspension of the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
- *Rules that activate pending processes:* When *Pr* is empty, a pending process may be activated. When a process is loaded, its local variables are also loaded into memory.
- *Transport rules:* These rules move messages into and out of the external message queue. Because the external message queue is represented as a separate RL object, it can belong to a different subconfiguration from that of the object itself. Consequently, messages can be received in parallel with other activity in the object.

Specifications in RL are executable on the Maude tool, so Creol's operational semantics may be used as a language interpreter. The entire interpreter consists of 700 lines of code, including auxiliary functions and equational specifications, and it has 32 rewrite rules. A detailed presentation of the interpreter may be found in [7]. The rules for asynchronous method calls and guarded commands are now considered in more detail.

4.1 Asynchronous Method Calls

Objects communicate by asynchronous method calls. In the operational semantics, two messages are used to encode a method call. If an object o_1 calls a method m of an object o_2, with arguments in, and the execution of $m(in)$ results in the return values out, the call is reflected by two messages $invoc(l, o_1, o_2, m, in)$ and $comp(l, o_1, out)$, which represent the invocation and completion of the call, respectively. In the asynchronous setting, the invocation message must include the reply address of the caller, so the completion can be transmitted to the correct destination. As an object may have several pending calls to another object, the completion message includes a unique

label l, generated by the caller.

When an object calls an external method, a message is placed in the configuration. Transport rules eventually move the message to the callee's external message queue. After method execution, a completion message is emitted into the configuration, eventually arriving at the caller's external message queue.

The interpreter checks the external message queue of a Creol object for method invocations, and loads the corresponding method code from the object's class into the object's internal process queue PrQ The rewrite rule for this transition can be expressed as follows, ignoring irrelevant attributes in the style of Full Maude [13]:

rl [receivecall] :

$\langle O : \mathit{Id} \mid \mathit{Cl} : C, \mathit{PrQ} : W\rangle\ \langle q(O) : \mathit{QId} \mid \mathit{Ev} : Q\ \mathit{invoc}(N, O', O, M, I)\rangle$
$\langle C : \mathit{Cl} \mid \mathit{Mtds} : \mathit{MT}\rangle\ \Rightarrow$
$\langle O : \mathit{Id} \mid \mathit{Cl} : C, \mathit{PrQ} : (\mathit{get}(M, \mathit{MT}, (N\ O'\ I)))\ W\rangle\ \langle q(O) : \mathit{QId} \mid \mathit{Ev} : Q\rangle$
$\langle C : \mathit{Cl} \mid \rangle$.

The auxiliary function *get* fetches method M in the method set MT of the class, and returns a process with the method's code and local variables. Values of actual parameters I, the caller O', and the message label N, are stored as local variables. (The label cannot be modified by the process.) The rule for a local asynchronous call is similar, but the call comes from the active process code Pr instead of the external message queue. For a synchronous local call the code is loaded directly into the active process code Pr, since waiting actively in this case leads to deadlock.

4.2 Guarded Commands

Creol has three types of guards representing potential processor release points: The standard boolean expression, a wait guard, and a return guard. Rules for evaluation of return guards in the active process are now considered. Return guards allow process suspension when waiting for method completions, so the object may attend to other tasks while waiting. A return guard evaluates to `true` if the external message queue contains the completion of the method call, and execution of the process continues.

crl [returnguard] :

$\langle O : \mathit{Id} \mid \mathit{Pr} : X?(J) \longrightarrow P, \mathit{Lvar} : L\rangle\ \langle q(O) : \mathit{QId} \mid \mathit{Ev} : Q\ \mathit{comp}(N,\ O,\ K)\rangle\ \Rightarrow$
$\langle O : \mathit{Id} \mid \mathit{Pr} : (J := K); P, \mathit{Lvar} : L\rangle\ \langle q(O) : \mathit{QId} \mid \mathit{Ev} : Q\rangle$
if $N == \mathit{val}(X, L)$.

The condition ensures that the correct reply message is identified. The auxiliary function *val* fetches the value associated with the label variable X from the local variables L.

If the message is not in the queue, the active process is suspended. In this case other enabled processes may be activated while waiting for the method call completion.

crl [returnguard_notinqueue] :
$\langle O : Id \mid Pr : X?(J) \longrightarrow P, PrQ : W, Lvar : L\rangle \ \langle q(O) : QId \mid Ev : Q\rangle \ \Rightarrow$
$\langle O : Id \mid Pr : empty, PrQ : W(X?(J) \longrightarrow P, L), Lvar : no\rangle \ \langle q(O) : QId \mid Ev : Q\rangle$
if not $inqueue(val(X, L), Q)$.

where the function *inqueue* looks for the completion in the message queue Q.

If no process is active, the suspended process with the return guard can be tested against the external message queue again. If the completion message is present, the return value is matched to local or object attributes and the process is reactivated.

crl [return_guard_st] :
$\langle O : Id \mid Pr : empty, PrQ : (X?(J) \longrightarrow P, L')\ W\rangle$
$\langle q(O) : QId \mid Ev : Q\ comp(N,\ O,\ K)\rangle \ \Rightarrow$
$\langle O : Id \mid Pr : (J := K); P, PrQ : W, Lvar : L'\rangle \ \langle q(O) : QId \mid Ev : Q\rangle$
if $N == val(X, L')$.

Otherwise, another pending process from the multiset PrQ may be activated.

4.3 Execution of Creol Programs in Maude

The operational semantics of Creol is executable on the rewriting logic tool Maude, as an interpreter for Creol programs. This makes Maude well-suited for experimenting with programming constructs and language prototypes, combined with Maude's various rewrite strategies, search, and model-checking abilities.

Although the operational semantics is highly nondeterministic, Maude is deterministic in its choice of which rule to apply to a given configuration. The dining philosophers program of Section 3 is used to test the performance of the interpreter. Running the example on the interpreter, we observe that Maude selects processes from the internal process queue in an unfair manner. Even with the "fair" rewrite strategy [14], the philosophers would only *think* after 10 000 rewrites. This means that although the suspended instance of *digest* is enabled in each philosopher, it is not executed. In order to explore the full state space of the above program, Maude provides a breadth-first search facility. However, for highly nondeterministic systems, the naive use of this search mechanism will often become very resource demanding and hard to apply in practice.

5 Nondeterministic Execution

This section presents an approach to nondeterministic execution of Maude specifications. The approach is based on Maude's reflective capabilities in order to control the rewriting process. Informally, a configuration C and the set $\mathcal{R}$ of rewrite rules of a Maude specification may be represented as terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the metalevel, and metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by a

sequential interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Details on the theory and use of reflection in rewriting logic and Maude may be found in [12,15]. A strategy for rule selection which employs a pseudo-random number generator, is now defined in Maude syntax.

There is a vast selection of algorithms for generating pseudo-random numbers. For simplicity, we select a simple "minimal" general purpose algorithm from *Press et al.* [27, p. 278]: $I_{j+1} = a\ I_j\ (mod\ m)$. In [27], the authors argue that choosing $a = 7^5$ and $m = 2^{31} - 1$ yields a generator that has passed all important theoretical tests, and that has been put to successful use in a variety of practical applications. The algorithm is programmed in Maude as a functional module *RANDOM*:

fmod RANDOM is
  protecting NAT .
  op rand : *Nat* $\rightarrow$ *Nat .*
  ops a m : $\rightarrow$ *Nat .*
  eq a = 16807 . *** = 7^5
  eq m = 2147483647 . *** = $2^{31} - 1$
  var N : *Nat .*
  eq rand$(N) = (a * N)$ *rem* m .
endfm

The nondeterministic rewriting strategy is defined as a Maude module *META-ENGINE* that protects the built-in module *META-LEVEL*. Metalevel rewriting is carried out by a conditional rule *exec*, described below. An object *Engine* keeps track of the current state, and is defined as follows:

op ⟨*Engine* | *curTerm* : _, *curModule* : _, *labels* : _, *failedRules* : _,
  numRules : _, *randomNum* : _, *randomNum2* : _⟩ :
    Term Qid QidList QidList Int Int Int $\rightarrow$ *EngineObject .*

The object contains several attributes, whose values are set at run-time; *curTerm* contains the metarepresentation of the current configuration, *curModule* is the metarepresentation of the name of the base-level module in which the rewrites will be performed, *labels* is a *QidList* of rule labels from the module *curModule*, *failedRules* contains a *QidList* of rule labels for rules that could not be applied to *curTerm*, *numRules* is the length of the *labels* list, included for performance reasons, and finally *randomNum* and *randomNum2* are numbers generated by the pseudo-random number generator defined above.

The actual metalevel rewrite steps are handled by Maude's built-in descent function *metaXapply*$(\overline{\mathcal{R}}, \bar{t}, \bar{l}, \sigma, n, b, m)$, where $\mathcal{R}$ is module, t a term, l a rule label, σ a (partial) substitution, n a match number, b a bound, and m a solution number [14]. The last argument, the solution number m, is of particular interest for nondeterministic execution. Our strategy for performing a rewrite is two-fold:

(i) The engine will select, using the pseudo-random number generator, a rule label l corresponding to a rule in $\mathcal{R}$.

(ii) A rule identified by l may be applicable to a term t at several different positions within the term, referred to in Maude as solutions. To allow for "deep" randomization, within objects as well as between them, we will also select the solution number pseudo-randomly.

The conditional rewrite rule *exec* implements this strategy, using *metaXapply* and an auxiliary function *chooseSolution*$(\overline{\mathcal{R}}, \bar{t}, \bar{l}, r)$. The latter takes care of part (ii) of the strategy, and chooses a valid solution number using the pseudo-random number r. It makes use of the fact that *metaXapply*$(\overline{\mathcal{R}}, \bar{t}, \bar{l}, \sigma, n, b, m)$ returns *failure* when there is no solution m. It is therefore easy to find the number s of possible solutions by repeatedly calling *metaXapply* with increasing values for m, until it returns failure. Utilizing this information, selecting a solution randomly becomes a matter of calculating r *mod* s. If no solution can be found (i.e. the rule is not applicable), *chooseSolution* returns -1. The code for *exec* is given below:

```
crl [exec] :
   ⟨Engine | curTerm : T, curModule : MOD, labels : L, failedRules : FR,
   numRules : NR, randomNum : R, randomNum2 : R2⟩
⇒
   if chooseSolution(MOD, T, findItem(L, R rem NR), R2) == −1
   then
      ⟨Engine | curTerm : T, curModule : MOD, labels : L,
      failedRules : if findItem(L, R rem NR) in FR then FR
      else FR findItem(L, R rem NR) fi,
      numRules : NR, randomNum : rand(R), randomNum2 : rand(R2)⟩
   else
      ⟨Engine | curTerm : getTerm(metaXapply([MOD], T,
      findItem(L, Rrem NR), none, 0, unbounded,
      chooseSolution(MOD, T, findItem(L, R rem NR), R2))),
      curModule : MOD, labels : L, failedRules : nil, numRules : NR,
      randomNum : rand(R), randomNum2 : rand(R2)⟩
   fi
if length(FR) < NR .
```

Performing one rewrite at the time, the engine selects a rule randomly from its list of rule labels, and tries to apply it to the current configuration. If rule application fails (i.e., the left side of the rule does not match the current configuration and *chooseSolution* returns -1), the rule label is added to the list *failedRules*. If the length of *failedRules* equals the number *numRules* of rules in the module, the execution terminates, as no rule is applicable.

Once an applicable rule has been selected, the list of failed rules is reset to *nil*, and the rule is applied. The resulting term is assigned to *curTerm*, and another rewrite may be performed in the same manner. For specifications

where a majority of the rules will be nonapplicable to any given configuration, the strategy given in *exec* will prove to be inefficient. To amend this, rules that have already failed may be temporarily removed from the *labels* list until a rule application succeeds.

The metarewriting engine introduced in this section makes it easy to simulate a series of different executions of any valid Maude specification, by supplying different seeds for the pseudo-random number generator. Therefore, the engine provides a rewriting strategy for testing specifications of nondeterministic systems, complementary to Maude's standard rewrite and search facilities. For the Creol example of the Dining Philosophers (Section 3), this strategy provides much more informative testing than Maude's internal fair rewrite strategy on a single run, distributing rewrite steps evenly between the different philosophers and their different enabled methods, and easily provides a series of different "fair" executions of the program.

6 Related Work

Many object-oriented languages offer constructs for concurrency. A common approach has been to keep activity (threads) and objects distinct, as done in Hybrid [26] and Java [19]. These languages rely on the tightly synchronized RPC model of method calls, forcing the calling method instance to block while waiting for the reply to a call. Verification considerations suggest that methods should be serialized [9], which would block all activity in the calling object. Closely related are method calls based on the rendezvous concept in languages where objects encapsulate activity threads, such as Ada [6] and POOL-T [4]. The latter is interesting because of its emphasis on reasoning control and compositional semantics, allowing inter-object concurrency [5].

For distributed systems, with potential delays and even loss of communication, the tight synchronization of the RPC model seems less desirable. Hybrid offers *delegation* as an explicit programming construct to (temporarily) branch an activity thread. Asynchronous method calls can be implemented in e.g. Java by explicit concurrency control, creating new threads to handle calls. In order to facilitate the programmer's task and reduce the risk of errors, implicit control structures based on asynchronous method calls seem more attractive, allowing a higher level of abstraction in the language.

Languages based on the Actor model [2,3] take asynchronous messages as the communication primitive, focusing on loosely coupled concurrency with less synchronization. This makes Actor languages conceptually attractive for distributed programming. Representing method calls by asynchronous messages has lead to the notion of future variables, which is found in languages such as ABCL [29] and ConcurrentSmalltalk [28], as well as in Eiffel// [10], CJava [16], and Polyphonic C$^\sharp$ [8]. The proposed asynchronous method calls are similar to future variables, but the proposed nested processor release points further extend this approach to asynchrony.

In Maude's standard object model [13,23], object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules involving many objects) are allowed, which makes Maude's object model very flexible. However, asynchronous method calls and suspension points as proposed in this paper are hard to represent directly within this model.

7 Concluding Remarks

Whereas object orientation has been advocated as a promising framework for distributed systems, common approaches to combining object-oriented method invocations with distribution seem less satisfactory. High-level implicit control structures may facilitate the design of distributed concurrent objects. This paper proposes asynchronous method calls and nested processor release points in method bodies for this purpose. The approach is more flexible than serialized methods, while maintaining the ease of partial correctness reasoning about code which is lost for nonserialized methods. However liveness reasoning in our setting will require a fairness guarantee which is not provided by the RL semantics, suggesting the need for a fair scheduling strategy.

The executable semantic framework provided by rewriting logic and Maude offers valuable support for the development of program constructs, allowing experimentation with the language behavior during development. In order to simulate the highly nondeterministic environment targeted by the language, a nondeterministic rewrite strategy has been proposed, based on a pseudo-random number generator. We are currently extending the abstract machine with a metalayer which captures the semantic specifications of the abstract interfaces of the language [21,22]. This provides a prototyping environment for initial designs, allowing the observable behavior of objects to be controlled without explicit modelling, and a testing environment for imperative programs without explicit representation of the environment. Analysis techniques for Creol programs are under development. In future work, we plan to extend the operational semantics with class inheritance mechanisms, including method overloading, and use this model to experiment with dynamic features of open object systems such as run-time mechanisms for system upgrades.

References

[1] Ábrahám-Mumm, E., F. S. de Boer, W.-P. de Roever and M. Steffen, *Verification for Java's reentrant multithreading concept*, in: *International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02)*, Lecture Notes in Computer Science **2303** (2002), pp. 5–20.

[2] Agha, G. A., *Abstracting interaction patterns: A programming paradigm for open distributed systems*, in: E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP*

International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96) (1996), pp. 135–153.

[3] Agha, G. A., I. A. Mason, S. F. Smith and C. L. Talcott, *A foundation for actor computation*, Journal of Functional Programming **7** (1997), pp. 1–72.

[4] America, P., *POOL-T: A parallel object-oriented language*, in: A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, The MIT Press, Cambridge, Mass., 1987 pp. 199–220.

[5] America, P., J. de Bakker, J. N. Kok and J. Rutten, *Operational semantics of a parallel object-oriented language*, in: *13th ACM Symposium on Principles of Programming Languages (POPL'86)* (1986), pp. 194–208.

[6] Andrews, G. R., "Concurrent Programming: Principles and Practice," Addison-Wesley, Reading, Mass., 1991.

[7] Arnestad, M., "En abstrakt maskin for Creol i Maude," Master's thesis, Department of informatics, University of Oslo, Norway (2003), in Norwegian. Available from http://heim.ifi.uio.no/~creol.

[8] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for* $C^\sharp$, in: B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, Lecture Notes in Computer Science **2374** (2002), pp. 415–440.

[9] Brinch Hansen, P., *Java's insecure parallelism*, ACM SIGPLAN Notices **34** (1999), pp. 38–45.

[10] Caromel, D. and Y. Roudier, *Reactive programming in Eiffel//*, in: J.-P. Briot, J. M. Geib and A. Yonezawa, editors, *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, Lecture Notes in Computer Science **1107**, Springer-Verlag, Berlin, 1996 pp. 125–147.

[11] Cenciarelli, P., A. Knapp, B. Reus and M. Wirsing, *An event-based structural operational semantics of multi-threaded Java*, in: J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science **1523**, Springer-Verlag, 1999 pp. 157–200.

[12] Clavel, M., "Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications," CSLI Publications, Stanford, 2000.

[13] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science **285** (2002), pp. 187–243.

[14] Clavel, M., F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. Talcott, "Maude 2.0 Manual," Computer Science Laboratory, SRI International (2003).

[15] Clavel, M. and J. Meseguer, *Reflection in conditional rewriting logic*, Theoretical Computer Science **285** (2002), pp. 245–288.

[16] Cugola, G. and C. Ghezzi, *CJava: Introducing concurrent objects in Java*, in: M. E. Orlowska and R. Zicari, editors, *4th International Conference on Object Oriented Information Systems (OOIS'97)* (1997), pp. 504–514.

[17] de Oliveira Braga, C., "Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics," Ph.D. thesis, Pontifícia Universidade Católica do Rio de Janeiro (2001).

[18] Dijkstra, E. W., *Guarded commands, nondeterminacy and formal derivation of programs*, Communications of the ACM **18** (1975), pp. 453–457.

[19] Gosling, J., B. Joy, G. L. Steele and G. Bracha, "The Java language specification," Java series, Addison-Wesley, Reading, Mass., 2000, 2nd edition.

[20] International Telecommunication Union, *Open Distributed Processing - Reference Model parts 1–4*, Technical report, ISO/IEC, Geneva (1995).

[21] Johnsen, E. B. and O. Owe, *A compositional formalism for object viewpoints*, in: B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)* (2002), pp. 45–60.

[22] Johnsen, E. B. and O. Owe, *Object-oriented specification and open distributed systems*, in: O. Owe, S. Krogdahl and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, Lecture Notes in Computer Science **2635**, Springer-Verlag, 2004 pp. 137–164.

[23] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[24] Meseguer, J., *Rewriting logic as a semantic framework for concurrency: A progress report*, in: U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science **1119** (1996), pp. 331–372.

[25] Najm, E. and J.-B. Stefani, *A formal semantics for the ODP computational model*, Computer Networks and ISDN Systems **27** (1995), pp. 1305–1329.

[26] Nierstrasz, O., *A tour of Hybrid – A language for programming with active objects*, in: D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, Prentice Hall, 1992 pp. 167–182.

[27] Press, W. H., S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, "Numerical Recipes in C: The Art of Scientific Computing," Cambridge University Press, Cambridge, UK, 2002, second edition.

[28] Yokote, Y. and M. Tokoro, *Concurrent programming in ConcurrentSmalltalk*, in: A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, The MIT Press, Cambridge, Mass., 1987 pp. 129–158.

[29] Yonezawa, A., "ABCL: An Object-Oriented Concurrent System," Series in Computer Systems, The MIT Press, 1990.

Toward Reflective Application Testing in Open Environments

Eyvind W. Axelsen, Einar Broch Johnsen, and Olaf Owe
Department of Informatics, University of Oslo

Abstract

Many distributed applications can be understood in terms of components interacting in an open environment such as the Internet. Open environments are subject to change in uncontrollable ways, as other applications may arrive, change, or disappear. In order to test the behavior of components in such environments, it is necessary to build a testing environment which reflects this highly unpredictable behavior. To avoid over-specification of environment components, we use the observable communication history to abstractly reflect the state of communicating components. Rewriting logic has been used to model many different systems of concurrency and communication in an executable manner. In this paper, we show how rewriting logic models can be extended with observable communication histories in a transparent way and suggest using this extension to capture a form of assumption guarantee specification based testing of components in open environments.

1 Introduction

The aim of this paper is to suggest an application of rewriting logic [18] to test the behavior of software units in *open environments* such as the Internet. An open environment is an environment in which other unknown software units exist, and no specific knowledge about these units may be assumed. A software unit in this setting may be a distributed application in an open distributed system, but also an off-the-shelf component which should behave properly in a variety of environments. To keep the exposition simple, we will model software units by *objects* communicating by means of *message passing* in this paper, keeping in mind that the approach may in principle be extended to "real components" or distributed applications.

It is a major challenge to predict the behavior of objects evolving in open environments, in order to ensure and maintain behavioral properties such as safety, availability, quality of service, various forms of fault tolerance, etc. Formal approaches to this challenge include methods for verification, by means of e.g. assertion systems such as Hoare logic, type checking, and model checking. A disadvantage of these approaches is that they generally depend on knowledge of the implementation details of the systems they consider. In contrast, approaches based on testing create an artificial environment in which the object can be subjected to controlled test runs. In contrast to verification methods, testing cannot generally ensure that components are well-behaved at all times, but may still give interesting insights into the component's behavior. This paper takes a testing approach to object analysis. The goal of the paper is to show how open environments can be mimicked by underspecified formal descriptions based on *observable behavior* in order to test object behavior.

Testing is done in an executable platform defined using *rewriting logic* and the Maude system [5, 18]. Rewriting logic can naturally express and combine many different models of communication and concurrency. Further, rewriting logic is *reflective* [3, 6] in a mathematically precise manner: it is possible to reason formally about reflective rewriting inside rewriting logic itself, and to execute reflective specifications at the Maude *meta-level*. The use of reflection is essential to our approach, allowing for guided search and system monitoring in a modular, composable, and hierarchical way. Reflection may be used to define execution strategies for an executable object model, for example a *nondeterministic* execution strategy is proposed in [16]. Reflective specifications support a layered architecture where several specifications may be given at each level. Reflection can be used to extend a system model with e.g. logging facilities [23]. In this paper, we transparently extend an executable system model with its history of observable communications [7, 12] at the meta-level, and define execution strategies at the meta-level which are influenced by requirements on the history. This paper gives an overview, technical details of the implementation of this work in Maude may be found in [1].

Paper overview: Section 2 briefly reviews a notion of behavioral interface and introduces the example of the paper. Section 3 introduces rewriting logic and Maude. Section 4 provides an executable Maude specification of the running example. Section 5 explains how histories are introduced into the Maude model. Section 6 considers testing of observable behavior. Section 7 provides an executable *abstract* Maude specification of the running example. Section 8 combines two meta-level strategies to create an open testing environment. Section 9 concludes the paper.

2 Specifications of Observable Behavior

In the open distributed setting, objects in the environment may come from third-party manufacturers, and their implementation details may be unavailable for various reasons. Reasoning about the overall system behavior should be based on abstract specifications of system components. Specification in terms of observable behavior seems particularly attractive, assuming that components have behavioral interfaces that describe their use. A component may have many interfaces corresponding to different behavioral requirements.

Interaction histories. In a distributed environment, object behavior may be given in terms of an assumption guarantee specification [17]. The assumption is a requirement on the behavior of the objects in the environment. As customary in the assumption-guarantee paradigm, the guaranteed invariant need only hold when the environment respects the assumption. In our setting, the paradigm is adjusted to deal with input and output aspects of communicating systems. An object's observable history, i.e. the trace of all communication events between the object and its environment, represents an abstract view of its state, available for reasoning about past and present behavior. Consequently, an object's behavior may be determined by its communication history up to present time, and a specification of its behavior as a predicate on finite traces. The approach emphasizes mathematically intuitive concepts such as generator inductive function definitions and finite sequences, avoiding fix-point semantics and infinite traces [14].

Behavioral interfaces. In order to specify object behavior in a generic way, we introduce interfaces which may be associated with objects. Interfaces that contain semantic requirements can be understood as *behavioral interfaces*. An interface can be

implemented by different classes and a class can implement different interfaces. If a class implements an interface, all the objects of the class must behave according to the semantic requirements of the interface, which describe observable behavior in terms of possible communication histories. For further technical details and discussion the reader is referred to [13, 14]. We shall here proceed by an example.

Example. Consider the well-known dining philosophers example [9] with N philosophers seated around a table. They are thinking deeply, but may occasionally need to eat from a common resource. Each philosopher is equipped with one chopstick, but in order to eat two chopsticks are needed. Hence, a philosopher may request and return its right-hand neighbor's chopstick, and lend its stick to its left-hand neighbor in response to a request. These are the possible philosopher operations. We assume that philosophers are initially thinking, but at some point they may request their neighbor's chopstick, and lend their chopstick to their left-hand neighbor. A philosopher which controls two chopsticks may eat, return the requested chopstick, and resume thinking.

We specify an interface *Phil*, describing observable philosopher behavior. Let X and Y be variables ranging over possible philosophers. Semantically, we represent an interaction by a triple $\langle X,Y,M\rangle$, where M is a method. Remark that *think* and *eat* are internal methods, i.e. $X = Y$, while *requestStick*, *returnStick* and *lendStick* are external, i.e. $X \neq Y$. In the latter case, say that X represents the initiator of the request. The behavior of a philosopher is formulated in terms of acceptable communication histories, which observationally reflect the internal state of a philosopher. Clearly, if the history ends with a request for the neighbor's chopstick, the philosopher is *hungry*, and if the history ends with the return of the chopstick, the philosopher is *fed*. The philosopher may only *eat* when it is hungry and only *think* when it is fed, so these actions do not change the state. The behavioral interface capturing philosopher behavior is defined as follows:

interface *Phil*
begin
  opr *think*
  opr *eat*
  opr *requestStick*
  opr *returnStick*
  opr *lendStick*
 inv $AccBeh(this,\mathcal{H})$
where $AccBeh(X,\varepsilon) \equiv \mathsf{true}$

$AccBeh(X,\mathcal{H} \vdash \langle X,Y,think\rangle) \equiv \mathcal{H} == \varepsilon \vee (fed?(X,\mathcal{H}) \wedge AccBeh(X,\mathcal{H}))$
$AccBeh(X,\mathcal{H} \vdash \langle X,Y,eat\rangle) \equiv hungry?(X,\mathcal{H}) \wedge AccBeh(X,\mathcal{H})$
$AccBeh(X,\mathcal{H} \vdash \langle X,Y,requestStick\rangle) \equiv fed?(X,\mathcal{H}) \wedge AccBeh(X,\mathcal{H})$
$AccBeh(X,\mathcal{H} \vdash \langle X,Y,returnStick\rangle) \equiv hungry?(X,\mathcal{H}) \wedge AccBeh(X,\mathcal{H})$
$AccBeh(X,\mathcal{H} \vdash \langle X,Y,lendStick\rangle) \equiv hasStick?(X,\mathcal{H}) \wedge AccBeh(X,\mathcal{H})$
$AccBeh(X,\mathcal{H} \vdash \langle X',Y,M\rangle\rangle)) \equiv AccBeh(X,\mathcal{H})$ **if** $(X \neq X')$

$hungry?(X,\mathcal{H} \vdash \langle X',Y,M\rangle) \equiv$ **if** $(X == X')$ **then** $(M == requestStick \vee M == eat)$
  else $hungry?(X,\mathcal{H})$ **fi**
$fed?(X,\mathcal{H} \vdash \langle X',Y,M\rangle) \equiv$ **if** $(X == X')$ **then** $(M == returnStick \vee M == think$
  $\vee M == lendStick)$ **else** $fed?(X,\mathcal{H})$ **fi**
$hasStick?(X,\mathcal{H}) \equiv (\mathcal{H}/\{\langle X,Y,lendStick\rangle, \langle Y,X,returnStick\rangle\}$ **ew** $returnStick)$

end

In the predicate definitions, the free variables in each equation have an implicit universal quantifier, reminiscent of for instance ML, and each line corresponds to a possible

generator case. The empty sequence is denoted ε, and **ew** denotes "ends with". The restriction of a history h to a set S of messages is denoted h/S. Note that the specification is underspecified with regard to philosopher behavior. In particular, we do not know if a philosopher will lend its chopstick to its neighbor when hungry.

3 Rewriting Logic and Maude

This section gives a brief introduction to rewriting logic [18] and Maude [5]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to fragments of a state configuration. If rewrite rules may be applied to non-overlapping fragments of the configuration, the transitions may be performed in parallel. Consequently, rewriting logic (RL) is a logic which easily captures concurrent change. A number of concurrency models have been successfully represented in RL [5, 18], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [19] and real-time systems [20]. RL is also used to define the operational semantics of the Creol language [15, 16], and additionally offers its own model of object orientation [5].

Informally, a state configuration in RL is a multiset of terms of given types. These types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [11] style. When modeling computational systems, configurations may include the local system states, where different parts of the system are modeled by terms of the different types defined in the equational logic. An RL object is a term of the type $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [5].

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the defining equations of E. Each rule describes how a part of a configuration can evolve in one transition step. If several rules can be applied to distinct subconfigurations, they can be executed in a concurrent rewrite step. Consequently, concurrency is implicit in RL.

Conditional rewrite rules are allowed, where the condition can be formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\textbf{crl}\ [label] :\ subconfiguration \longrightarrow subconfiguration\ \textbf{if}\ condition\ .$$

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [8].

Reflection and the Maude Meta-Level. Rewriting logic is reflective in the sense that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal*, meaning that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) in $\mathcal{U}$ [4].

Let C and C' be configurations and $\mathcal{R}$ be a set of rewrite rules. We write $\mathcal{R} \vdash C \rightarrow C'$ to express that C may be rewritten to C' in the rewrite theory $\mathcal{R}$. Informally, a configuration

C and the set $\mathcal{R}$ of rewrite rules of a Maude specification may be represented as terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the meta-level. Using this notation, we have the following equivalence:

$$\mathcal{R} \vdash C \to C' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{C} \rangle \to \langle \overline{\mathcal{R}}, \overline{C'} \rangle$$

In other words, if a term C can be rewritten to a term C' in a rewrite theory $\mathcal{R}$, then the meta-representation of C in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C} \rangle$ can be rewritten to the meta-representation of C' in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C'} \rangle$, in the universal rewrite theory $\mathcal{U}$, and vice versa.

Meta-level rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by a sequential interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Details on the theory and use of reflection in rewriting logic and Maude may be found in [3, 4, 6].

4 A Maude Model of the Dining Philosophers

The dining philosophers of Section 2 are now implemented in Maude. In order to define the *synchronization constraints*, we introduce internal attributes in the philosopher objects to determine their ability to perform the different possible actions. Let C and D be variables ranging over the sort *Cid* of *concrete* philosopher identifiers, a subsort of *Phil*. A concrete philosopher with internal structure may be defined as follows:

$$\langle C : Cid \mid state : _, myStick : _, nbrStick : _, leftNbr : _, rightNbr : _ \rangle$$

The philosophers interact asynchronously by passing messages to each other, as well as by sending synchronous messages to themselves representing internal actions.

A message consists of an envelope with a sender and a receiver and the actual message (or action) represented by a *quoted identifier*. In Maude, messages are conventionally defined as follows:

$$msg_\ from_\ to_$$

We are now able to define synchronization constraints directly in terms of a philosopher's internal state. A *hungry* philosopher may attempt to gain control of two chopsticks in order to eat. A *fed* philosopher may think. Hence, the *state* may be either *hungry* or *fed*. When a philosopher is hungry, it will attempt to acquire chopsticks. When it controls two chopsticks, it may eat, release the chopsticks, and resume thinking. This can be expressed by the following rewrite rules, ignoring attributes that are not needed for synchronization in the style of Full-Maude [5]:

rl $[think]$: $\langle C : Cid \mid state : fed \rangle \longrightarrow \langle C : Cid \mid state : fed \rangle$ $(msg\ 'think\ from\ C\ to\ C)$.

rl $[eat]$:
$\langle C : Cid \mid state : hungry, myStick : yes, nbrStick : yes \rangle$
$\longrightarrow \langle C : Cid \mid state : hungry, myStick : yes, nbrStick : yes \rangle$ $(msg\ 'eat\ from\ C\ to\ C)$.

rl $[requestStick]$:
$\langle C : Cid \mid state : fed, myStick : yes, nbrStick : no, rightNbr : D \rangle$
$\longrightarrow \langle C : Cid \mid state : hungry, myStick : yes, nbrStick : requested, rightNbr : D \rangle$
$(msg\ 'requestStick\ from\ C\ to\ D)$.

rl $[returnStick]$:
$\langle C : Cid \mid state : hungry, myStick : yes, nbrStick : yes, rightNbr : D \rangle$
$\longrightarrow \langle C : Cid \mid state : fed, myStick : yes, nbrStick : no, rightNbr : D \rangle$
$(msg\ 'returnStick\ from\ C\ to\ D)$.

rl $[lendStick]$:
$\langle C : Cid \mid state : fed, myStick : yes \rangle$ $(msg\ 'requestStick\ from\ D\ to\ C)$
$\longrightarrow \langle C : Cid \mid state : fed, myStick : no \rangle$ $(msg\ 'lendStick\ from\ C\ to\ D)$.

rl [*recieveRequestedStick*] :

$\langle C : Cid \,|\, nbrStick : requested, rightNbr : D\rangle$ $(msg\ 'lendStick\ from\ D\ to\ C)$

$\longrightarrow \langle C : Cid \,|\, nbrStick : yes, rightNbr : D\rangle$

rl [*recieveReturnedStick*] :

$\langle C : Cid \,|\, myStick : no, rightNbr : D\rangle$ $(msg\ 'returnStick\ from\ D\ to\ C)$

$\longrightarrow \langle C : Cid \,|\, myStick : yes, rightNbr : D\rangle$

Note that in this specification, a hungry philosopher will not render its chopstick.

5 Extending the Model with Histories

In the executable model, the messages corresponding to the philosophers' actions can be recorded in a *communication history*. In this section we will look at how to utilize Maude's meta-level capabilities during execution of the specified model to record the history in a *transparent* way, i.e. leaving the original specification unchanged.

In order to execute a specification at the meta-level, we develop a custom *strategy*, i.e. rewrite rules which apply to the meta-representation of the model. This strategy makes use of an object *Engine* that stores the information that we need to keep track of to control consecutive meta-level rewrites. The engine object is defined as follows:

$$\langle E : Engine \,|\, curTerm : _, curModule : _, labels : _, failedRules : _\rangle$$

This object contains several attributes, whose values are set at run-time; *curTerm* contains the meta-representation of the current configuration, *curModule* is the meta-representation of the name of the object-level module in which the rewrites will be performed, *labels* is a list of rule labels from the module *curModule*, and *failedRules* contains a list of rule labels for rules that could not be applied to *curTerm*.

The communication history is kept in a Maude object, *History*, which is distinct from the objects of the object-level model and is defined as follows:

$$\langle H : History \,|\, h : _\rangle$$

The only attribute of this object, *h*, will contain the actual recorded communication history in the form of a message list during runtime.

By defining a custom strategy we gain control over when a meta-level rewrite is performed, and hence we are able to inspect the current state in-between rewrites. This is the key point that enables us to record a communication history while executing a specification, since we can now check whether the application of a given rewrite rule to a given configuration results in a new message being sent, by comparing the meta-level representations of the configuration before and after the rule application, respectively.

Our strategy is implemented by a conditional rewrite rule *exec*, defined below. The actual term rewriting is performed by Maude's built-in *descent function* *metaXapply*($\overline{\mathcal{R}}$, $\bar{t}$, $\bar{l}$, σ, n, b, m). This (partial) function provides fine-grained control over rewriting at the meta-level, allowing us to specify which rewrite rule ($\bar{l}$) is to be applied to which term ($\bar{t}$), and at which position (m) within the term the rule is to be applied if there are several possibilities. For a more detailed description of this function, see [5]

The function *metaXapply* returns a four-tuple consisting of the resulting term, the type of the term, a substitution and a context, and we use a function *getTerm* to extract the rewritten term from this tuple. Note that whitespace in Maude denotes list concatenation: If *L* is a label and *LABELS* is a list of labels, then *L LABELS* is a non-empty list of labels.

$$
\begin{array}{l}
\textbf{crl}\ [exec] :\\
\quad \langle E : Engine \mid curTerm : T, curModule : MOD, labels : L\ LABELS, failedRules : FR \rangle\\
\quad \langle H : History \mid h : ML \rangle\\
\longrightarrow\\
\quad \textbf{if}\ metaXapply([MOD], T, L, none, 0, unbounded, 0) =/= failure\ \textbf{then}\\
\qquad \langle E : Engine \mid curTerm : getTerm(metaXapply([MOD], T, L, none, 0, unbounded, 0)),\\
\qquad\quad curModule : MOD, labels : LABELS\ L, failedRules : nil \rangle\\
\qquad \langle H : History \mid h : ML + getNewMessages(T, getTerm(metaXapply([MOD],\\
\qquad\quad T, L, none, 0, unbounded, 0)), MOD, ML) \rangle\\
\quad \textbf{else}\\
\qquad \langle E : Engine \mid curTerm : T, curModule : MOD, labels : LABELS\ L, failedRules : FR \rangle\\
\qquad \langle H : History \mid h : ML \rangle\\
\quad \textbf{fi}\\
\textbf{if}\ length(FR) < length(L\ LABELS).
\end{array}
$$

This strategy applies rules from the *labels* list in a round-robin fashion to the meta-level configuration in *curTerm*. (Remark that we may define other strategies for rule selection than round-robin. In [16], we have shown how to extend a similar strategy such that the rules are selected randomly using a pseudo-random number generator.) In the event that no rule is applicable, the execution will terminate.

The auxiliary function *getNewMessages* compares the term T with the term resulting from applying (with *metaXapply*) the rule labeled L to T. If there are new communication messages in the new configuration, the attribute h of the history object is extended with the new messages. If there are several new messages, these are caused by concurrent actions and we can add them to the history in an arbitrary order.

Using the strategy defined above, we may execute a Maude specification, such as the one for the dining philosophers problem introduced in Section 4, and record any messages that are being sent, without changing anything in the original specification. Hence, the analysis capabilities obtained by recording the history may be added to the model when needed and removed after sufficient analysis.

6 A Strategy for Testing Observable Behavior

The communication history that is built at runtime when applying the rewrite strategy introduced in Section 5, can be used as input to a test *oracle*, blocking execution if a given Maude specification violates a given behavioral specification. This can be achieved by extending the strategy with functionality for checking whether a given rule application will lead to an illegal state, as specified by a given predicate. Taking the observational approach, we consider predicates on communication histories. In order to obtain a compositional system, the predicate on the global history will be the conjunction of a number of behavioral interfaces, associated with different objects. Behavioral specifications for specific object-level objects are represented by trace predicates on the global history, restricted to an appropriate subset of possible communication events. The overall picture is illustrated by Figure 1: Let $\mathcal{R}$ be the object-level set of rewrite rules and $\mathcal{C}$ a system state. A meta-level strategy $\mathcal{S}$ controls how the choice of rule is made for application at the object level. The strategy is here parameterized by a predicate on communication histories. For the testing strategy of this section, a *fail-stop* strategy is defined, which blocks further execution once the system attempts to violate the predicate P on the global history.

	Rule set:	Configuration:
Meta-level rewrite system:	$\mathcal{S}_{\textit{fail-stop}}(P(h))$	$\overline{\mathcal{R}}, \overline{C}, \langle H : \mathit{History} \mid h : _ \rangle$
	$\downarrow$ Control	$\uparrow$ History logger
Object level rewrite system:	$\mathcal{R}$	C

Figure 1: Reflective testing of observable behavior.

In order to implement *fail-stop* testing in Maude, we first define a constant H of a new sort *History*, which we will use as a placeholder for the actual communication history $\mathcal{H}$ (which will be recorded during execution, and hence is not available at the time of specification) in the specification of our predicates. Furthermore, we define a sort *Pred*, the sort of predicates on communication histories. For the running example, acceptable behavior for a system of philosophers behaving according to the behavioral interface defined in Section 2, can then be expressed by the Maude operator

$$\textbf{op}\ \mathit{AccBeh} :\ \mathit{History} \rightarrow \mathit{Pred}\ .$$

During execution, the predicate needs to be checked between each rewrite step. For this purpose, we introduce the function $\mathit{CheckPredicate} : \mathit{Pred} \times \mathit{MsgList} \rightarrow \mathit{Bool}$. This function will be called by the strategy from Section 5, and will parse the predicate specification, call auxiliary predicate checking functions and return a boolean value indicating whether the message list is in compliance with the predicate or not.

Returning to the *AccBeh* predicate, the case in which *think* is the last message in the history can be specified equationally in Maude as follows:

eq *AccBeh((msg'think from X to Y) + ML) = (ML==nil)* **or** *(fed?(X, ML)* **and** *AccBeh(ML))* .

eq *fed?(X, nil) = true* .
eq *fed?(X, (msg M from X to Y) + ML) = (M == 'returnStick*
  or *M == 'think* **or** *M == 'lendStick)* .
ceq *fed?(X, (msg M from Y to Z) + ML) = fed?(X, ML)* **if** *X =/= Y* .

Here, X, Y, and Z are variables of sort *Phil*, M is a message of sort *Msg*, and *ML* is the communication history in the form of a message list as recorded by our strategy during execution. The last equation for *fed?* is *conditional*, and can only be applied if $X \neq Y$. Note that since the predicate in the Maude specification is a *global* predicate that spans all objects, there is no need to pass the object identifier as a separate parameter to *AccBeh*

If the communication history after a given rewrite is not in compliance with the predicate, the execution will terminate and a "failure object" will be inserted into the configuration to indicate what went wrong. For more details on the implementation of the predicate check, see [1].

7 An Executable Abstract Prototype of Dining Philosophers

We now define a prototype philosopher model in Maude. The idea is to postpone design decisions concerning the concrete internal structure and rather define the behavior of the philosopher in terms of restrictions on the history of its observable actions. Let A and B be variables ranging over object identifiers of sort *Aid* for *abstract* philosophers. Define

a sort *AbsPhil* with terms $\langle A : Aid |\ leftNbr : _,\ rightNbr : _\rangle$. Except for philosopher identities and their left- and right-hand side neighbors, we completely ignore any internal structure in the philosopher objects. Consequently, synchronization constraints cannot be expressed in terms of the internal state. The possible actions of the abstract philosophers are captured by the following rewrite rules:

$$\textbf{rl}\ [think] :\ \langle A : Aid |\ \rangle \longrightarrow \langle A : Aid |\ \rangle\ (msg\ 'think\ from\ A\ to\ A)\ .$$

$$\textbf{rl}\ [eat] :\ \langle A : Aid |\ \rangle \longrightarrow \langle A : Aid |\ \rangle\ (msg\ 'eat\ from\ A\ to\ A)\ .$$

$$\begin{array}{l}\textbf{rl}\ [requestStick] : \\ \quad \langle A : Aid |\ rightNbr : B\rangle \\ \longrightarrow \langle A : Aid |\ rightNbr : B\rangle\ (msg\ 'requestStick\ from\ A\ to\ B)\ .\end{array}$$

$$\begin{array}{l}\textbf{rl}\ [returnStick] : \\ \quad \langle A : Aid |\ rightNbr : B\rangle \\ \longrightarrow \langle A : Aid |\ rightNbr : B\rangle\ (msg\ 'returnStick\ from\ A\ to\ B)\ .\end{array}$$

$$\begin{array}{l}\textbf{rl}\ [lendStick] : \\ \quad \langle A : Aid |\ leftNbr : B\rangle \longrightarrow \\ \quad \langle A : Aid |\ leftNbr : B\rangle\ (msg\ 'lendStick\ from\ A\ to\ B)\ .\end{array}$$

These rules do not express any synchronization constraints on the interactions, only which philosophers may interact. Also note, that rules for receiving messages are no longer needed, since no internal state change takes place in the abstract philosopher objects. Instead, a simple consumption rule can be used to remove messages from the configuration:

$$\textbf{rl}\ [consumeMsg] :\ \langle A : Aid |\ \rangle (msg\ M\ from\ B\ to\ A) \longrightarrow \langle A : Aid |\ \rangle\ .$$

8 Simulating Open Environments by Behavioral Interfaces

In an open environment, objects may be created and destroyed dynamically during execution. With our abstract philosopher specification, this can be modeled by the following rewrite rules:

$$\begin{array}{l}\textbf{rl}\ [create] : \\ \quad \langle A : Phil \mid rightNbr : B\rangle\ \langle B : Phil \mid leftNbr : A\rangle \\ \longrightarrow \langle A : Phil \mid rightNbr : A+B\rangle\ \langle B : Phil \mid leftNbr : A+B\rangle \\ \quad \langle A+B : Aid |\ leftNbr : A, rightNbr : A+B\rangle \\ \ .\end{array}$$

$$\begin{array}{l}\textbf{rl}\ [destroy] : \\ \quad \langle A : Phil \mid rightNbr : B\rangle\ \langle B : Aid |\ leftNbr : A, rightNbr : C\rangle\ \langle C : Phil \mid leftNbr : B\rangle \\ \longrightarrow \langle A : Phil \mid rightNbr : C\rangle\ \langle C : Phil \mid leftNbr : A\rangle\ .\end{array}$$

In the *create* rule, the new abstract philosopher object is inserted in-between two existing (abstract or concrete) philosopher objects. The new philosopher will have the concatenation of the existing objects' identifiers as its identifier. In the *destroy* rule, an abstract philosopher object in-between two other philosopher objects is deleted, and the remaining philosopher objects set their *leftNbr* and *rightNbr* properties accordingly.

Using our abstract dining philosopher specification with the rules introduced above, we can simulate an open environment of which the behavior is exclusively defined by a behavioral specification in the form of predicates at the meta-level. As mentioned in Section 5, the meta-level can be used to define a strategy which stops the execution of a specification if a given rule application violates the predicate. However, when defining

	Rule set:	Configuration:
Meta-level:	$\mathcal{S}_{force}(P_1(h/\alpha_1)) \wedge \mathcal{S}_{fail\text{-}stop}(P_2(h/\alpha_2))$	$\overline{\mathcal{R}_1} \cup \overline{\mathcal{R}_2}, (\overline{C_1}\ \overline{C_2}), \langle H : History \mid h : _ \rangle$
	$\downarrow$ Control	$\uparrow$ History logger
Object level:	$\mathcal{R}_1 \cup \mathcal{R}_2$	$C_1\ C_2$

Figure 2: Reflective testing of observable behavior in open environments.

an abstract environment where every rule is applicable at any time (because there is no synchronization code in the objects), this is not optimal. Instead, we want to *force* the abstract specification to behave in compliance with the predicate. This can be achieved by a similar strategy, using the mechanisms introduced in Section 6. However, where the *fail-stop* strategy halts the execution when the application of an enabled rule will break the predicate, the new *force* strategy will try another enabled rule from the *labels* list of the *Engine* object instead. Execution will first terminate when no rule can be applied without violating the predicate.

The abstract environment specification can now be used as a "testbed" for a number of actual programmed components, like the philosophers from Section 2. Let $\mathcal{R}_1$ be an object-level set of rewrite rules which may be applied to a system configuration C_1, the system state of abstract objects (the open environment), and let $\mathcal{R}_2$ be the object-level set of rewrite rules which may be applied to the concrete objects in a system configuration C_2 (the given components, with synchronization constraints on the internal state). Let α_1 and α_2 be messages associated with the objects of C_1 and C_2, respectively. Messages may be exchanged freely between all objects, so the two sets are not disjoint. Let P_1 be a predicate observationally specifying the environment. The meta-level strategy $\mathcal{S}_{force}$ restricts rule application from $\mathcal{R}_1$ to acceptable environment behavior. This provides an abstract, open environment which may behave in any way that does not violate the specification P_1. We here combine two meta-level strategies which react differently to the violation of predicates: *force* will restrict rule application so that the communication history conforms to the predicate, and *fail-stop* will halt the execution and produce an error object if the predicate is attempted violated. By specifying one predicate that spans only messages from the programmed object, and one that spans all objects, and by checking the former in *fail-stop* mode and the latter in *force* mode, we can test whether our programmed component execute correctly provided that the environment does so. This scenario is illustrated by Figure 2.

To illustrate this scheme by the running example of this paper, we may consider a system of dining philosophers, where some philosophers are implemented, i.e. belong to the concrete philosophers defined in Section 4. We want to test that these concrete objects behave according to the requirements of the *Phil* interface by means of the *fail-stop* strategy. To create an environments in which they interact with other, abstract philosophers, we introduce the abstract philosophers of Section 7. In order to make the abstract philosophers behave according to the same *Phil* interface, we use the *force* strategy. In Figure 2, the *Phil* interface will be used as both predicates P_1 and P_2.

9 Conclusion and Future Work

This paper shows how abstract specifications of dynamic environments may be captured very naturally in a rewriting logic model extended with behavioral interfaces. Due to the reflective character of rewriting logic, supported by Maude, it is possible to define execution strategies at the meta-level. In this paper, we have used this facility to test whether an executable model is well behaved with respect to a number of requirements on the observable behavior of the model, defined as behavioral interfaces.

Further, we show how meta-level strategies may be used to execute a prototype model defined by its observable behavior, without deciding on its implementation details. Combining these meta-level strategies, we obtain abstract testing environments for models of components or distributed applications, in which the environment is unspecified but subjected to certain minimal observational requirements. Previous approaches to history-based testing, e.g. [10, 21], and automata-based approaches, e.g. [2, 22], require specific test cases to be defined. In contrast, our approach uses *random testing* and assumption guarantee specifications to define *open environments*. Further, the environment and the test oracle are defined within the same formalism. Methods to generate additional restrictions on assumptions to obtain reasonable coverage remain to be addressed.

Some experiments with socket extensions to Maude suggest that it is possible to employ Maude processes as demonstrated in this paper to act as a testing environment and to simulate an open environment for actual components communicating by means of a predefined set of messages with the Maude process via sockets. However, future work in this vein remains.

References

[1] E. W. Axelsen. A meta-level framework for recording and utilizing communication histories in Maude. Master's thesis, Dept. of Informatics, Univ. of Oslo, Aug. 2004. To appear.

[2] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proc. European Dependable Computing Conference (EDCC2), October 1996*, LNCS 1150, pages 303–320. Springer, 1996.

[3] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, Stanford, California, 2000.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marí-Oliet, and J. Meseguer. Metalevel computation in Maude. In *2nd Intl. Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*. Elsevier, 1998.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[6] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, Aug. 2002.

[7] O.-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, pages 57–114. Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, Dec. 1977.

[8] C. de Oliveira Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[11] J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*. Addison-Wesley, 1986.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[13] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer, Mar. 2002.

[14] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, LNCS 2635, pages 137–164. Springer, 2004.

[15] E. B. Johnsen, O. Owe, and M. Arnestad. Combining active and reactive behavior in concurrent objects. In *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir, Nov. 2003.

[16] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*. To appear in ENTCS. Elsevier, Mar. 2004.

[17] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, UK, June l981.

[18] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[19] E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.

[20] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[21] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In A. Petrenko and A. Ulrich, editors, *3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES 2003)*, LNCS 2931, pages 1–14. Springer, 2004.

[22] L. Van Aertryck, M. Benveniste, and D. Le Metayer. CASTING : A formally based software test generation method. In *1st IEEE Intl. Conf. on Formal Engineering Methods (ICFEM'97)*, pages 101–110, 1997. IEEE Computer Society Press.

[23] N. Venkatasubramanian and C. L. Talcott. Reasoning about meta level activities in open distributed systems. In *Proc. 14th Annual ACM Symp. on Principles of Distributed Computing (PODC '95)*, pages 144–152, Aug. 1995. ACM Press.