Maude: Specification and Programming in Rewriting Logic[1]

Manuel Clavel, Francisco Durán, Steven Eker,
Patrick Lincoln, Narciso Martí-Oliet,
José Meseguer, and José Quesada

Computer Science Laboratory
SRI International

March 8, 1999

[1]Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312 and NASA Contract NAS2-98073, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grants CCR-9505960 and CCR-9633363.

Contents

Chapter 1

Introduction

Maude is a high-performance language and system supporting both equational and rewriting logic computation for a wide range of applications. Maude has been influenced in important ways by OBJ3 [27]. In particular, Maude's equational logic sublanguage essentially contains OBJ3 as a sublanguage. The main differences from OBJ3 at the equational level are a much greater performance, and a richer equational logic, namely, membership equational logic [41], that extends OBJ3's order-sorted equational logic [26].

The key novelty of Maude is that—besides efficiently supporting equational computation and algebraic specification in the OBJ style—it also supports *rewriting logic* computation. Rewriting logic [37] is a logic of concurrent change that can naturally deal with state and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency [40]. In particular, it supports very well concurrent *object-oriented* computation. This is reflected in Maude's design by providing special syntax for *object-oriented modules*. Since the computational and logical interpretations of rewriting logic are like two sides of the same coin, the same reasons making it a good semantic framework at the computational level make it also a good *logical framework* at the logical level, that is, a *metalogic* in which many other logics can be naturally represented and implemented [33, 32]. Consequently, some of the most interesting applications of Maude are *metalanguage* applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation.

The rewriting logic research program, although still very young, has shown good signs of vitality, including two international workshops [20, 28], over a hundred research papers (see the references in [42, 28]), and three language implementation efforts, namely ELAN [29, 3] in France, CafeOBJ [24, 23] in Japan, and Maude. Therefore, Maude should be seen as our contribution to the broader collective effort of building good language implementations for rewriting logic. In this regard, a key distinguishing feature of Maude is its systematic and efficient use of *reflection*—exploiting the fact that rewriting logic is reflective [15, 10]—a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

The present paper documents Maude 1.0, and explains Maude's basic concepts in a leisurely and mostly informal style, illustrating those concepts with examples. Early language design for Maude appeared in [35, 46, 38]. A first implementation of Maude, supporting reflection, was presented and demonstrated at the First International Workshop on Rewriting Logic in September 1996 [13].

A beta version has been available since March 1998 [9].

1.1 The Logical Basis of Maude

Maude's *functional modules* are theories in *membership equational logic* [41, 4], a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : s$, stating that a term t has sort s. Such a logic extends order-sorted equational logic [26], and supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains. Maude's functional modules are assumed to be Church-Rosser; they are executed by the Maude engine according to the rewriting techniques and operational semantics developed in [4].

Membership equational logic is a sublogic of *rewriting logic* [37]. A rewrite theory is a pair (T, R) with T a membership equational theory, and R a collection of labeled and possibly conditional *rewrite rules* involving terms in the signature of T. Maude's *system modules* are rewrite theories in exactly this sense. The rewrite rules $r : t \longrightarrow t'$ in R are *not* equations. Computationally, they are interpreted as local *transition rules* in a possibly concurrent system. Logically, they are interpreted as *inference rules* in a logical system. As already mentioned, this makes rewriting logic both a general *semantic framework* to specify concurrent systems and languages [40], and a general *logical framework* to represent and execute different logics [33].

Rewriting in (T, R) happens *modulo* the equational axioms in T. Maude supports rewriting modulo most of the different combinations of associativity, commutativity, identity, and idempotency axioms. The rules in R need not be Church-Rosser and need not be terminating. Many different rewriting paths are then possible; therefore, the choice of appropriate *strategies* is crucial for executing rewrite theories.

In Maude, such strategies are not an extra-logical part of the language. They are instead *internal strategies* defined by rewrite theories at the metalevel. This is because rewriting logic is *reflective* [10] in the precise sense of having a *universal theory* U that can represent any finitely presented rewrite theory T (including U itself) and any terms t, t' in T as terms $\overline{T}$ and $\overline{t}, \overline{t'}$ in U, so that we have the following equivalence:

$$T \vdash t \longrightarrow t' \quad \Leftrightarrow \quad U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle.$$

Since U is representable in itself, we can then achieve a "reflective tower" with an arbitrary number of levels of reflection. Maude efficiently supports this reflective tower through its `META-LEVEL` module, which makes possible not only the declarative definition and execution of user-definable rewriting strategies, but also many other applications, including an extensible *module algebra* of parameterized module operations that is defined and executed within the logic.

This extensibility by reflection is exploited in Maude's design and implementation, so that the basic functionalities of the language, Core Maude, are extended by reflection to Full Maude. Core Maude supports module hierarchies consisting of (unparameterized) functional and system modules and provides the `META-LEVEL` module. Full Maude is an extension of Core Maude written in Core Maude itself that supports a module algebra of parameterized modules, views, and module expressions in the OBJ style [27] as well as *object-oriented modules* with convenient syntax for object-oriented applications.

1.2 Core Maude

The Maude system is built around the Core Maude interpreter, which accepts module hierarchies of (unparameterized) functional and system modules with user-definable mixfix syntax. It is implemented in C++ and consists of two parts: the rewrite engine and the mixfix front end.

The rewrite engine is highly modular and does not contain any Maude-specific code. Two key components are the "core" module and the "interface" module. The core module contains classes for objects which are not specific to an equational theory, such as equations, rules, sorts, and connected sort components. The "interface" module contains abstract base classes for objects that may have a different representation in different equational theories, such as symbols, term nodes, dag nodes, and matching automata. New equational theories can be "plugged in" by deriving from the classes in the "interface" module. To date, all combinations of associativity, commutativity, left and right identity, and idempotence have been implemented apart from those that contain both associativity and idempotence. New built-in symbols with special rewriting (equation or rule) semantics may be easily added.

The engine is designed to provide the look and feel of an interpreter with hooks for source level tracing/debugging and user interrupt handling. These goals prevent a number of optimizations that one would normally implement in a compiler, such as transforming the user's term rewriting system, or keeping pending evaluations on a stack and only building reduced terms. The actual implementation is a semi-compiler where the term rewriting system is compiled to a system of tables and automata, which is then interpreted. Typical performance with the current version is 800K-840K free-theory[1] rewrites per second and 27K-111K associative-commutative (AC) rewrites per second on standard hardware (300 MHz Pentium II). The figure for AC rewriting is highly dependent on the complexity of the AC patterns (AC matching is NP-complete) and the size of the AC subjects. These results were obtained using fairly simple linear and nonlinear patterns and large (hundreds of nested AC operators) subjects.

The mixfix front end consists of a bison/flex based parser for Maude's surface syntax, a grammar generator (which generates the context-free grammar (CFG) for the mixfix parts of Maude over the user's signature), a context-free parser, a mixfix pretty printer, a fully reentrant debugger, the built-in functions for quoted identifiers, and the `META-LEVEL` module, together with a considerable amount of "glue" code holding everything together. Many of the C++ classes are derived from those in the rewrite engine. The Maude parser (MSCP) is implemented using SCP as the formal kernel [54]. The techniques used include β-extended CFGs (that is, CFGs extended with "bubbles" (strings of identifiers) and precedence/gathering patterns). MSCP provides a basis for flexible syntax definition, and an efficient treatment of what might be called *syntactic reflection*, which is very useful for parsing inputs in extensions of Core Maude such as Full Maude, and in other languages with user-definable syntax that can likewise be implemented in Maude. The point is that we often need to parse the top level syntax, for example of a module, and then extract from it the grammar in which to parse the user-definable expressions in that module.

The functional module `META-LEVEL` efficiently implements key functionality of the universal theory U. In `META-LEVEL` Maude terms are reified as elements of a data type `Term`, and Maude modules are reified as terms in a data type

[1] We say that the rewriting is done in the free-theory when rewriting with terms whose operators have no equational attributes.

`Module`. The processes of reducing a term to normal form in a functional module and of rewriting a term in a system module using Maude's default interpreter are respectively reified by functions `meta-reduce` and `meta-rewrite`. Similarly, the process of applying a rule of a system module to a subject term is reified by a function `meta-apply`. Furthermore, parsing and pretty printing of a term in a signature, as well as key sort operations, are also reified by corresponding metalevel functions.

1.3 Full Maude

Using reflection, Core Maude can be extended to a much richer language with an extensible *module algebra* of module operations that can make Maude modules highly reusable. The basic idea is that `META-LEVEL` can be extended with new data types—extending its `Module` sort to richer sorts for structured and parameterized modules—and with new module operations—such as instantiation of parameterized modules by views, flattening of module hierarchies into single modules, desugaring of object-oriented modules into system modules, and so on. In particular, this supports an OBJ style of *parameterized programming* [27], with highly generic and reusable modules. All such new types and operations can be defined in Core Maude. This, together with the explicit access to modules as terms provided by reflection, makes the corresponding module algebra completely open, and easily extensible by new module operations and transformations.

Using the meta-parsing and meta-pretty printing functions in `META-LEVEL` and a simple `LOOP-MODE` module providing input/output, we can in addition develop in Core Maude a suitable user interface for Full Maude. At present, Full Maude supports all of Core Maude plus *object-oriented* modules, *parameterized* modules, *theories* with loose semantics to state formal requirements in parameters, *views* to bind parameter theories to their instances, and *module expressions* instantiating and composing parameterized modules.

1.4 Applications

All applications typical of equational programming and algebraic specification can be conveniently and efficiently supported through Maude's sublanguage of functional modules. In fact, the paper [41] argues that Maude's equational logic, namely, membership equational logic, is so expressive—yet efficiently implementable—as to offer very good advantages as a logical framework for a very wide range of algebraic specification languages based on both total and partial equational logic formalisms.

However, many other Maude applications go beyond equational logic. System modules support general rewriting logic applications. The important area of concurrent and distributed object-based system specification and prototyping is supported by object-oriented modules. And reflection makes possible many novel metaprogramming and metalanguage applications. In particular, reflection is extremely valuable in many applications using rewriting logic as a *logical and semantic framework*. Thanks to the sustained efforts of many researchers, particularly in the ELAN, Pisa, Stanford, and Maude teams, there is by now very extensive evidence supporting the claim that rewriting logic is indeed a very flexible and simple semantic framework [37, 40, 42, 6], and logical framework [32, 29, 62, 30, 2, 55, 8, 16, 10, 12]. Moreover, object-oriented design languages, architectural description languages, and languages for distributed components also have a natural semantics in rewriting logic [63, 34, 57, 47, 48] (see Section 2.8.2 for more discussion on the use of reflection in logical and semantic framework applications, and Appendix E for an application of Maude to the interoperation of software architectures).

The largest Maude application developed so far is Full Maude itself [19] (about 7,000 lines of Maude code). Two other substantial applications are an inductive theorem prover and a Church-Rosser checker for equational theories, that are part of a formal environment for Maude and for the CafeOBJ language [12]. In addition, several language interpreters and strategy languages, a supercompiler, several object-oriented specifications—including cryptographic protocols and network applications—and a variety of executable translations mapping logics, architectural description languages and models of computation into the rewriting logic reflective framework have been developed by different authors (see references in [20, 42, 28]). We hope that the present release will encourage and support many other applications.

Acknowledgments

Our previous work with Joseph Goguen and the other members of the OBJ team has influenced the design of Maude in important ways; in fact, OBJ3 is essentially a functional sublanguage of Maude. We thank Timothy Winkler for his valuable contributions to the development of the Maude ideas. We also thank Carolyn Talcott for many discussions on Maude and for her valuable suggestions on strategy aspects. We are grateful to Jean-Pierre Jouannaud and Adel Bouhoula for their collaboration on the proof theory and theorem proving techniques of membership equational logic, and to Peter Mosses for his very detailed and helpful comments on this paper and his kind help and advice with its LaTeX and HTML versions. Ralph Wachter deserves special thanks for encouraging us in the development of the Maude ideas from the initial stages to the present. We are grateful for very helpful discussions and exchanges with Christiano Braga, Roberto Bruni, Grit Denker, Kokichi Futatsugi, Claude and Hélène Kirchner, Alexander Knapp, Ulrike Lechner, Christian Lengauer, Ugo Montanari, Pierre-Etienne Moreau, Uri Nodelman, Peter Ölveczky, Isabel Pita, Mark-Oliver Stehr, Valentin Turchin, Martin Wirsing, and many other colleagues.

Chapter 2

Core Maude

After introducing functional and system modules we discuss module hierarchies. Several predefined modules such as Booleans, machine integers, quoted identifiers, and so on are also described. The reflective aspects of Maude, and the related topic of internal rewriting strategies—that is, strategies that can be defined with rewrite rules at the metalevel—are explained in detail. Parsing issues, as well as the input/output facilities provided by the `LOOP-MODE` module are also treated in detail. We finish the section with a discussion of system issues and debugging.

2.1 Functional Modules

Functional modules define data types and functions on them by means of equational theories whose equations are Church-Rosser and terminating. A mathematical model of the data and the functions is provided by the *initial algebra* defined by the theory, whose elements consist of equivalence classes of ground terms modulo the equations. Evaluation of any expression to its reduced form using the equations as rewrite rules assigns to each equivalence class a unique canonical representative. Therefore, in a more concrete way we can equivalently think of the initial algebra as consisting of those canonical representatives, that is, of the values to which the functional expressions evaluate by algebraic simplification using the equations.

As in the OBJ language [27] that Maude extends, functional modules can be unparameterized, or they can be parameterized with *functional theories* as their parameters. Core Maude only allows unparameterized modules, although, as further explained in Section 2.3 and also illustrated in some of the following examples, such unparameterized modules can import other modules to form module hierarchies. Parameterized modules are supported in Full Maude, as discussed in Section 3.5.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called *membership equational logic* [41, 5]; we discuss this and give more details about the semantics of functional modules in Section 4.1. For the moment, it suffices to say that, in addition to supporting sorts, subsort relations, and overloading of function symbols, functional modules also support *membership axioms*, a generalization of sort constraints [43] in which a term is asserted to have a certain sort if a condition consisting of a conjunction of equations and of unconditional membership tests is satisfied. Such membership axioms can be used to define partial functions, that become defined when their arguments satisfy certain equational and membership conditions.

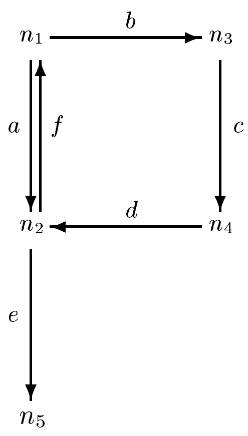


Figure 2.1: An Automaton.

We can illustrate these ideas, as well as Maude's support for mixfix user-definable syntax, with a module `PATH` that forms paths over a graph. Consider the graph in Figure 2.1. This graph describes an automaton whose *states* are the nodes of the graph, and whose *transitions* are the labeled edges. A *behavior* of the automaton is a *path* in the graph, that is, a concatenation of transitions such that the target state of one transition becomes the source state of the next transition. Of course, not all random concatenations of edges are legal paths, that is, not all strings of edges are behaviors of the automaton. The `PATH` module below axiomatizes the automaton and characterizes in a computable way its paths by means of a path concatenation operation, a length function, and source and target functions, together with appropriate axioms in membership equational logic.

```
fmod PATH is
  protecting MACHINE-INT .

  sorts Edge Path Path? Node .
  subsorts Edge < Path < Path? .

  ops n1 n2 n3 n4 n5 : -> Node .
  ops a b c d e f : -> Edge .
  op _;_ : Path? Path? -> Path? [assoc] .
  ops source target : Path -> Node .
  op length : Path -> MachineInt .

  var E : Edge .
  var P : Path .

  cmb (E ; P) : Path if target(E) == source(P) .

  ceq source(E ; P) = source(E) if E ; P : Path .
  ceq target(P ; E) = target(E) if P ; E : Path .
  eq length(E) = 1 .
```

```
  ceq length(E ; P) = 1 + length(P) if E ; P : Path .

  eq source(a) = n1 .
  eq target(a) = n2 .
  eq source(b) = n1 .
  eq target(b) = n3 .
  eq source(c) = n3 .
  eq target(c) = n4 .
  eq source(d) = n4 .
  eq target(d) = n2 .
  eq source(e) = n2 .
  eq target(e) = n5 .
  eq source(f) = n2 .
  eq target(f) = n1 .
endfm
```

The module is introduced with the functional module syntax `fmod ... endfm` and has a name, `PATH`. It imports a predefined module of machine integers with the declaration `protecting MACHINE-INT` (for more on predefined modules see Section 2.4, and for more on module importation and module hierarchies see Section 2.3). The sorts and subsort relations of this module are introduced by a sort declaration and a subsort declaration. Sorts—we could have called them *types* instead—are used to classify data. A subsort relation between two sorts is interpreted as a set-theoretic inclusion, that is, it means that the data of the subsort is included in that of the supersort. For example, the subsort declaration

```
subsorts Edge < Path < Path? .
```

declares that edges are a subsort of paths—that is, the set of edges is contained in the set of paths—and paths are a subsort of a supersort `Path?` of what we might call "confused paths." This supersort is needed because in general the path concatenation operator `_;_` may build nonsensical concatenations that are not paths. This operator is declared with the "infix" syntax

```
op _;_ : Path? Path? -> Path? [assoc] .
```

where the declaration indicates that it is a binary operator with `Path?` as the sort of its two arguments and also of its result. Before the colon, the user-definable "mixfix" syntactic form of the operator is given. In this case it is an infix operator with the two underbars indicating the places where the first and second arguments should be placed, namely, on both sides of the semicolon. The "attribute declaration" `assoc` states that `_;_` is *associative*. The Maude engine then uses this information when matching the equations and membership axioms in the module, that are then matched "modulo associativity," that is, regardless of how parentheses are left- or right-associated in a concatenation expression. In general, when an operator is associative the user does not have to write such parentheses around expressions involving several instances of such an operator. For example, `b ; c ; d` is a perfectly acceptable and unambiguous expression because of associativity.

Except for the conditional membership axiom for path concatenation, and the use of sort tests in the conditions of some equations, that we explain below, the rest of the module should be straightforward. Some nodes and edges are declared, plus source, target, and length functions, all of them with standard *prefix* notation, that is also allowed as a simpler choice of user-definable syntactic form. Then equations are given, defining the semantics of the operations. Each equation is introduced by the keyword `eq`, or `ceq` for conditional equations, and having variables of appropriate sorts previously declared with `var` declarations. The equations are then used from left to right by the Maude engine to simplify each expression to its canonical form, that is, to evaluate each expression to its corresponding value.

In general, an operator can be declared with the keyword `op`[1] followed by its *syntactic form*, followed by a colon, followed by the list[2] of sorts for its arguments (called the operator's *arity*), followed by `->`, followed by the sort of its result (called the operator's *coarity*). The operator can have some *attributes*, such as the `assoc` attribute for path concatenation, which indicate some equational axioms satisfied by the operator and used for term matching, or some syntactic information for parsing purposes, or some other information. All such attributes are declared within a single pair of enclosing square brackets after the sort of the result and before the ending period.

The syntactic form of the operator is a string of characters[3]. If no underbar character occurs in the string—as in the case of the `source`, `target`, and `length` functions—then the operator is declared in *prefix* form. If underbar characters occur in the string, then their number must coincide with the number of sorts declared as arguments of the operator. The operator is then in *mixfix* form, with the n-th underbar indicating the place where arguments of the n-th sort must be placed in expressions formed with that operator. There may or may not be any other characters before or after any of the underbars. If no other characters appear, we say that the operator has been declared with *empty syntax*. For example, we could have instead declared the path concatenation operator with empty syntax as

```
op __ : Path? Path? -> Path? .
```

and then `b c d` would be a `Path` expression (see Section 2.1.1 for more discussion on the mixfix syntax of operators, and Section 2.7 for a general discussion of mixfix parsing issues).

The ruling out of nonsensical concatenations is achieved by the *conditional membership axiom*[4]

```
cmb (E ; P) : Path if target(E) == source(P) .
```

[1] It is possible to simultaneously declare several operators having the same arity and coarity by using instead the keyword `ops` and giving the nonempty list of their corresponding syntactic forms after the `ops` keyword, as done for the nodes and edges declared in our example.

[2] If this list is empty, as for the edges and nodes declared in our example, the operator is called a *constant*.

[3] Such a string may have blank spaces and may consist of several identifiers in the Maude sense; see Section 2.1.1 for more details on the syntactic conventions.

[4] Unconditional membership axioms are introduced with the keyword `mb`. For example, in a module `NAT` of natural numbers with Peano notation we can define subsorts `Zero` (for the zero element) and `3*Nat` for numbers that are multiples of three by declaring

```
fmod NAT is
   sorts Zero 3*Nat Nat .
   subsorts Zero < 3*Nat < Nat .
   op 0 : -> Zero .
   op s_ : Nat -> Nat .
   var M : 3*Nat .
   mb s s s M : 3*Nat .
endfm
```

stating that an edge concatenated with a path is also a path if the target node of the edge coincides with the source node of the path. This has the effect of defining path concatenation as a partial function on paths, although it is total on the supersort `Path?` of confused paths. In fact, the domain of definition of path concatenation as a partial function on path pairs is the set of path pairs (`P`,`Q`) satisfying the equational condition

```
target(P) == source(Q) .
```

Note, however, that the corresponding conditional membership axiom

```
cmb (P ; Q) : Path if target(P) == source(Q) .
```

is not explicitly asserted in the module. It is instead an *inductive consequence* of all the axioms given in the module, including the simpler membership axiom that is indeed asserted. That is, it holds in the *initial algebra* specified by the functional module, that provides the mathematical semantics for the module as explained in Section 4.1. Such inductive properties can be proved using an inductive theorem prover in the style of the one proposed in [12]. Of course, the above membership axiom could instead have been declared as an axiom in the module, but we have chosen to use the more restricted membership axiom because it has a more efficient execution, and because it allows us to illustrate the distinction between the axioms given explicitly in a module and their inductive consequences.

All variables in righthand sides of equations should also appear in the corresponding lefthand sides. The *conditions* in conditional membership axioms—respectively, in conditional equations—should only involve variables appearing in the corresponding membership predicate—respectively, in the corresponding lefhand side. As the above example shows, the user can use the Boolean-valued, built-in equality predicate `_==_` and sort predicates such as `_: Path`, and in general Boolean combinations of such predicates or of other user-defined Boolean-valued expressions, in conditions of equations and membership axioms[5]. See Sections 2.4.1 and 4.1 for more details on the built-in equality and inequality predicates, and for a discussion of why negations and Boolean combinations of built-in equality and membership predicates in conditions—which would seem to go beyond Horn logic—are unproblematic under appropriate Church-Rosser and terminating assumptions about the specification.

Note that the functions `source`, `target`, and `length` are only defined on legal paths, so that on nonsensical paths they will return an unevaluated expression in an *error supersort*. For the first two functions the error supersort is `Error(Node)` above the sort of nodes, and for the third it is `Error(MachineInt)` above the sort of machine integers. Such expressions are very informative error messages.

[5] It is possible to give instead a single equation of the form `exp = exp'` as the condition. In fact, giving just a Boolean expression `exp` as the condition is equivalent to giving the equation `exp = true`. Note that there is no real loss of generality in restricting conditions to be either a single equation or a Boolean expression, since we can for example express a condition involving a conjunction of equations and membership axioms of the form

$$t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \wedge u_1 : s_1 \wedge \ldots \wedge u_m : s_m$$

by the single equation

$$(t_1 \texttt{ == } t'_1 \texttt{ and } \ldots \texttt{ and } t_n \texttt{ == } t'_n \texttt{ and } u_1 : s_1 \texttt{ and } \ldots \texttt{ and } u_m : s_m) \texttt{ = true},$$

or just by the Boolean expression

$$t_1 \texttt{ == } t'_1 \texttt{ and } \ldots \texttt{ and } t_n \texttt{ == } t'_n \texttt{ and } u_1 : s_1 \texttt{ and } \ldots \texttt{ and } u_m : s_m.$$

The Maude system automatically adds these error supersorts above each of the connected components of the poset of sorts declared by the user, using the set of maximal sorts in each connected component to qualify the corresponding error sort; such error sorts are called *kinds* in the theory of membership algebras (see Section 4). In this example there is a third connected component in the subsort ordering poset, namely, the connected component involving the sorts `Edge`, `Path`, and `Path?`, and therefore a third error supersort, `Error(Path?)`, is also added. Note that, even though `Path?` was introduced by the user with the purpose of catching errors, Maude always adds a new error supersort above each connected component. This is because, conceptually, an error supersort is really not a sort, but a *kind*. The point is that sorts are user-defined, while kinds are implicitly associated with connected components of sorts and are automatically added by Maude in the form of “error supersorts”. The Maude system also lifts automatically to the error supersorts all the operators involving sorts of the corresponding connected components to form *error expressions*. Such error expressions allow us to give expressions to be evaluated the benefit of the doubt: if when they are simplified they have a legal sort, then they are ok; otherwise, the fully simplified error expression is returned as an error message.

As illustrated by a few sample evaluations and their results, expressions formed with the operators declared in the module can be evaluated with the `reduce` command—`red` in abbreviated form. In the reduction process the equations are used from left to right as rules of simplification, and the membership axioms are also used to lower the sort of each expression as much as possible. When the expression has the lowest possible sort and cannot be simplified anymore using the equations, it is returned together with its lowest sort as the result.

```
Maude> red (b ; c ; d) .
result Path: (b ; c ; d)

Maude> red length(b ; c ; d) .
result NzMachineInt: 3

Maude> red (a ; b ; c) .
result Path?: (a ; b ; c)

Maude> red source(a ; b ; c) .
result Error(Node): source(a ; b ; c)

Maude> red target(a ; b ; c) .
result Error(Node): target(a ; b ; c)

Maude> red length(a ; b ; c) .
result Error(MachineInt): length(a ; b ; c)
```

2.1.1 Identifiers, Order-Sorted Signatures, and Overloading

We first explain the syntactic conventions about Maude *identifiers*, which must be followed when declaring module, sort, and operator names. Then we explain the notions of order-sorted signature and overloading that are key for understanding the syntax of expressions in functional and system modules.

In Core Maude, the name of a module or a sort must be an *identifier*. For example, `PATH`, `Edge`, and `Path` are identifiers. In general, an identifier in Maude is any finite string of ASCII characters such that:

- It does not contain any white space. For example, the sequence '`abc def`' is not one identifier, but two.
- The characters '`{`', '`}`', '`(`', '`)`', '`[`', '`]`' and '`,`' are *special*, in that they break a sequence of characters into several identifiers. For example, the sequence '`ab{c,d}ef`' counts as *seven* identifiers, namely, '`ab`', '`{`', '`c`', '`,`', '`d`', '`}`', and '`ef`'.
- The backquote character '`` ` ``' is only used as an *escape character* to indicate that a blank space or the special characters do not break the sequence. Consequently, backquotes can *only* appear immediately *before* any of the special characters, or *between* two nonempty strings of characters—with neither the ending of the first string nor the beginning of the second string being another backquote—for exactly these purposes. For example, ``1`ab`{c`,d`}ef`` is a single identifier. Maude's pretty printer will display such an identifier in the form '`1 ab{c,d}ef`'.

The conventions for the syntactic form of operators allow great flexibility in their user-definable syntax. An operator declared using a *single identifier* has automatically a *prefix* form, in which it can be displayed before its arguments enclosed in parentheses; but if such an operator contains underscore characters '`_`' then it must contain exactly as many underscores as the number of sorts in its arity, and in that case it has also a *mixfix* form. For example, `_+_(2,3)` and `2 + 3` are the prefix and mixfix ways of displaying the same arithmetic expression[6].

An operator can also be declared using *several identifiers*. This can be due to the presence of special characters, or to blank spaces, or both. Consider for example the operator declaration

```
op [_] and then [_] : Command Command -> Command .
```

that may allow a natural language style in the syntax of a programming language. It uses eight identifiers in the Maude sense, but declares a single binary operator, with the underscores indicating the place of the arguments in the mixfix notation. Internally, Maude also associates to this operator a corresponding *single identifier* variant by using backquotes. This is the form in which we could have equivalently defined the operator using a single identifier, namely,

```
op `[_`]and`then`[_`] : Command Command -> Command .
```

Of course, both variants are equivalent and have the same mixfix display, but the version without backquotes is obviously more convenient.

The declaraton of an operator requires an extra pair of parentheses if we already use parentheses as part of the syntax of the operator. Suppose we had in a programming language another binary operator `(_ only after _)`. We have to declare it as follows.

```
op ((_ only after _)) : Command Command -> Command .
```

[6]By default, Maude's pretty printer will display operators in mixfix form whenever possible, but it can be turned to prefix mode by the `set print mixfix off` command (see Appendix A).

Since an operator may be declared using several identifiers, in an `ops` declaration involving several operators each operator declaration can be enclosed in parentheses if necessary, to indicate where the syntax of each operator begins and ends. Then, we could have declared both operators together as follows.

```
op ([_] and then [_]) ((_ only after _)) :
     Command Command -> Command .
```

We now turn to order-sorted signatures. As the `GRAPH` example shows, the sorts declared in a functional module (and the same will hold for system modules) can be related by a subsort inclusion ordering. At the level of the corresponding algebras this subsort ordering is interpreted as set-theoretic *inclusion* of the data in a subsort into the data in a supersort. For example, every `Edge` is a `Path`, and every `Path` is a `Path?`. In general, we can declare arbitrary long chains of subsort inclusions, not only between individual sorts, but between sets of sorts. For example, if sorts `A`, `B`, and `C` are each of them subsorts of sorts `D` and `E`, and these in turn are subsorts of sorts `F`, `G`, and `H`, we can specify all these inclusions with a single declaration

```
subsorts A B C < D E < F G H .
```

Another feature of order-sorted signatures is that the function symbols declared in the signature can be *overloaded*, that is, we can have several operator declarations for the same operator with different arities and coarities. Consider for example the module

```
fmod NUMBERS is
  sorts Nat NzNat Nat3 .
  subsort NzNat < Nat .

  op zero : -> Nat .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat .
  op _+_ : NzNat NzNat -> NzNat .
  ops 0 1 2 : -> Nat3 .
  op _+_ : Nat3 Nat3 -> Nat3 [comm] .

  vars N M : Nat .
  var N3 : Nat3 .

  eq p s N = N .
  eq N + zero = N .
  eq N + s M = s (N + M) .
  eq (N3 + 0) = N3 .
  eq 1 + 1 = 2 .
  eq 1 + 2 = 0 .
  eq 2 + 2 = 1 .
endfm
```

declaring the natural numbers in Peano notation with a subsort `NzNat` of nonzero natural numbers and with successor, predecessor and addition functions, and declaring also the integers modulo 3 with their addition as a commutative (`comm`) operator.

The addition operator has three declarations and is therefore overloaded. However, there are two different kinds of overloading present in the example. The signature of the example is an *order-sorted* signature [26] in which overloaded operators related in the subsort ordering, such as the two additions for natural numbers and for nonzero natural numbers, are supposed to have the exact same behavior, in the sense that the bigger operator *restricts* to the smaller one on the subsorts; such operators are called *subsort overloaded*. Addition in the number hierarchy of naturals, integers, rationals, and so on, provides a very familiar example of subsort overloading, of which the present overloading of natural number addition is a special case (see Section 2.3).

By contrast, the sorts `Nat` and `NzNat` on the one hand, and the sort `Nat3` on the other form two different *connected components* in the subsort ordering and therefore natural number addition and addition modulo-3 are semantically unrelated. This form of overloading is called *ad-hoc overloading*. Both subsort and ad-hoc overloading of operators are allowed in Maude. However, to avoid ambiguous expressions we require that if the sorts in the arities of two operators with the same syntactic form are pairwise in the same connected components, then the sorts in the coarities must likewise be in the same connected component.

In particular, this rules out ad-hoc overloaded constants. Therefore, we have declared two different constants `zero` and `0` for the corresponding zero elements. However, this requirement can be relaxed, and it is often natural to do so. For example, the constants of a parameterized module can appear in many different connected components for different instances of the module, and it may be cumbersome to qualify them all. To allow this relaxation, constants—and, more generally, terms—can be qualified by their sort, by enclosing them in parentheses followed by a dot and the sort name. In this way, we could have instead declared `0` as an ad-hoc overloaded constant for naturals and for integers modulo-3, and could then disambiguate the expression `0 + 0` by writing, for example, `0 + (0).Nat` and `0 + (0).Nat3`, or `(0 + 0).Nat` and `(0 + 0).Nat3`.

Note that in an order-sorted signature a term can have several sorts. For example, the term `s s 0` in the `NUMBERS` module has sorts `NzNat` and `Nat`. An order-sorted signature is called *preregular* [26] when the set of sorts that can be assigned to a term according to the signature has always a *least element*. The order-sorted signatures in functional and system modules are assumed to be preregular.

Note that, as already mentioned, Maude will extend the signature given by the user in a module by adding the error supersorts above each connected component of sorts, and by adding an additional subsort overloaded operator with all its arities and coarities in the corresponding error supersorts for each family of subsort overloaded operators in the original signature for the purpose of dealing with error terms. Other operators such as equality predicates, sort predicates, and if-then-else, as well as the above-mentioned sort qualification operators are also added. See Section 2.7.2 for more details about this extended signature.

2.1.2 A Set Hierarchy Example

Another functional module example is provided by the following `SET-HIERARCHY` functional module, that defines the set hierarchy of all finite sets whose most basic elements are machine integers. Comments on the meaning of operations and equations are included in the text. Such comments must begin with `***` or `---`.

```
fmod SET-HIERARCHY is
  protecting MACHINE-INT .

  sorts Set Elt Magma .
  subsorts Set < Elt < Magma .
  subsorts MachineInt < Elt .

  op mt : -> Set .                  *** empty set
  op _,_ : Magma Magma -> Magma [assoc comm] .
  op {_} : Magma -> Set .
                                    *** set constructor
  ops _U_ _I_ : Set Set -> Set [assoc comm] .
                                    *** union, intersection
  op _\_ : Set Set -> Set .         *** difference
  op _in_ : Elt Set -> Bool .
  op P : Set -> Set .               *** power set
  op augment : Set Set -> Set .
  op |_| : Set -> MachineInt .      *** cardinality

  vars L M : Magma .
  vars E F : Elt .
  vars S T : Set .

*** equations between constructors to eliminate
*** duplicate elements
  eq { L , L , M } = { L , M } .
  eq { L , L } = { L } .

*** set union
  eq S U mt = S .
  eq { L } U { M } = { L , M } .

*** set membership
  eq E in mt = false .
  eq E in { F } = (E == F) .
  eq E in { F , L }
    = if E == F then true else E in { L } fi .

*** set intersection
  eq mt I S = mt .
  eq { E } I S = if E in S then { E } else mt fi .
  eq { E , L } I S =  ({ E } I S) U ({ L } I S) .

*** set difference
  eq mt \ S = mt .
  eq { E } \ S = if E in S then mt else { E } fi .
  eq { E , L } \ S = ({ E } \ S) U ({ L } \ S) .

*** power set (defined using auxiliary function "augment")
  eq P(mt) = { mt } .
  eq P({ E }) = { mt , { E } } .
  eq P({ E , L }) = P({ L }) U augment(P({ L }), { E }) .
```

```
    eq augment(mt, T) = mt .
    eq augment({ S } , T) = { S U T } .
    eq augment({ S , L } , T)
      = { S U T } U augment({ L }, T) .

  *** cardinality
    eq | mt | = 0 .
    eq | { E } | = 1 .
    eq | { E , L } |
      = | { L } | + if E in { L } then 0 else 1 fi .

  endfm
```

A finite set is represented using the standard notation {`E1,...,En`} as an (associative and commutative) collection of elements `E1,...,En` (here called a `Magma`) that is then enclosed in curly brackets by applying the constructor {_} , that builds a set out of a magma. Since `Set` is a subsort of `Magma`, sets can contain other sets as elements, and therefore we get the entire hierarchy of finite sets. The meaning of each operation symbol is explained either in its declaration or in the comments that precede the equations for that symbol. Notice that now several operators such as element concatenation, set union, and set intersection are declared to be associative and commutative with the `assoc` and `comm` attributes. The Maude engine then performs multiset matching and rewriting on those symbols; that is, neither association of parentheses nor the order of elements matter at all when finding a match. In general, the Maude engine can rewrite *modulo* most of the different combinations of associativity, commutativity, identity (left-, right-, or two-sided) and idempotency for different operators in the given specification. This of course gives the effect of rewriting the *equivalence classes* modulo such axioms of the terms in question, instead or rewriting just the terms themselves.

Note that the equational axioms declared as attributes of operators should *not* be written explicitly as equations in the specification. There are two reasons for this. Firstly, this is redundant, since they have already been declared as attributes. Secondly, although declaring such equations either only explicitly as equations, or twice—one time as attributes, and another as explicit equations—does not affect the *mathematical* semantics of the specification, that is, the initial algebra that the specification denotes (see Section 4.1) it does however drastically alter the specification's *operational semantics*. Indeed, Maude uses the equations from left to right as simplification rules, matching the equations *modulo* the axioms declared as attributes in operators. The equations in a Maude specification are assumed to be Church-Rosser and terminating modulo such axioms. But they may fail to have such properties if the axioms are instead added as ordinary equations. For example, a commutativity equation for set union such as

```
  eq S U T = T U S .
```

would make the above specification nonterminating.

Here are several sample reductions of set expressions.

```
  Maude> red P({ 1 , 2 , 3 }) \ P({ 1 , 2 }) .
  result Set: {{3}, {1, 3}, {2, 3}, {1, 2, 3}}

  Maude> red | P(P({ 1 , 2 , 3 })) | .
```

```
result NzMachineInt: 256

Maude> red | (P(P({1 , 2 , 3 })) \ P(P({ 1 , 2 }))) | .
result NzMachineInt: 240
```

2.1.3 Operator Evaluation Strategies

If a collection of equations is Church-Rosser and terminating, given an expression, no matter how the equations are used from left to right as simplification rules we will always reach the same final result. However, even though the final result may be the same, some orders of evaluation can be considerably more efficient than others. It may therefore be useful to have some way of controling the way in which equations are applied by means of strategies.

In general, given an expression $f(t_1, \ldots, t_n)$ we can try to evaluate it to its reduced form in different ways, such as:

- first obtaining the reduced form of all the t_is and then applying equations for f at the top of the term, what is called a *bottom-up*, or *eager* strategy;
- evaluating only some of the arguments, and then trying to evaluate at the top with equations for f; for example, an `if_then_else_fi` operator will typically be evaluated by evaluating first the first argument, and then the `if_then_else_fi` operator at the top;
- trying to evaluate the top of the term first, and then, if this fails, either not evaluating the subterms at all, or trying to evaluate only some of them, that is, some kind of *lazy* evaluation strategy.

Typically, a functional language is either eager, or lazy with some strictness analysis added for efficiency, and the user has to live with whatever the language provides. Maude adopts OBJ3's [27] flexible method of user-specified evaluation strategies on an operator-by-operator basis, adding some improvements to the OBJ3 approach to ensure a correct implementation [21]. For an n-ary operator f such strategies can be specified as a list of numbers from 0 to n ending with 0. For example, the default eager strategy given in Maude to all operators, unless another strategy is explicitly declared by the user, is (1 ... n 0), and the one given to the `if_then_else_fi` is (1 0 2 3 0), whereas a lazy "cons" list constructor may have strategy (0).

The syntax to declare an operator `f` with strategy (i_1 ... i_k 0) is

```
op f : S1 ... Sn -> S [strat (i1 ... ik 0)] .
```

Of course, if some of the argument positions are never mentioned in some of the operator strategies, the notion of *reduced expression* becomes now *relative* to the given strategies and may not coincide with the standard notion. This may be just what we want, since we may be able to achieve termination to a reduced expression relative to some strategies in cases when the equations may be nonterminating in the standard sense. For example, the factorial equation

```
fact(N) = if N == 0 then 1 else N * fact(N - 1) fi .
```

is nonterminating in the standard sense, but it is terminating up to the above strategy for `if_then_else_fi`. More generally, strategies may allow us to compute with infinite data structures which are evaluated on demand, such as the following slight reformulation of the Sieve of Eratosthenes example—which finds all prime numbers using lazy lists—in Appendix C.5 of [27].

```
fmod SIEVE is
  protecting MACHINE-INT .
  sort IntList .
  subsort MachineInt < IntList .
  op nil : -> IntList .
  op _._ : IntList IntList -> IntList
      [assoc id: nil strat (0)] .
  op force : IntList IntList -> IntList [strat (1 2 0)] .
  op show_upto_ : IntList MachineInt -> IntList .
  op filter_with_ : IntList MachineInt -> IntList .
  op ints-from_ : MachineInt -> IntList .
  op sieve_ : IntList -> IntList .
  op primes : -> IntList .

  var P I E : MachineInt .
  var S L : IntList .

  eq force(L, S) = L . S .
  eq show nil upto I = nil .
  eq show E . S upto I
    = if I == 0 then nil
      else force(E, show S upto (I - 1))
      fi .
  eq filter nil with P = nil .
  eq filter I . S with P
    = if (I % P) == 0 then filter S with P
      else I . filter S with P
      fi .
  eq ints-from I = I . ints-from (I + 1) .
  eq sieve nil = nil .
  eq sieve (I . S) = I . sieve (filter S with I) .
  eq primes = sieve ints-from 2 .
endfm

Maude> reduce show primes upto 10 .
result IntList: 2 . 3 . 5 . 7 . 11 . 13 . 17 . 19 . 23 . 29
```

The paper [21] documents in much more detail the operational semantics and the implementation techniques for Maude's operator evaluation strategies. In particular, it analyzes carefully a number of subtle anomalies in the OBJ3 implementation that are avoided in Maude, and uses a Maude specification of a rewrite theory (a *system module*, see Section 2.2) to formally specify the term graph rewriting done by the implementation to execute such strategies.

Of course, operator evaluation strategies, while quite useful, are by design restricted in their scope of applicability to *functional modules*[7]. As we shall see in Section 2.2, *system modules*, specifying rewrite theories—that are not functional, and need not be Church-Rosser or terminating—require much more general notions of strategy. Such general strategies are provided by Maude using reflection by means of *internal strategy languages*, in which strategies are defined by rewrite rules at the metalevel (see Section 2.6).

[7]More precisely, the scope of applicability of operator evaluation strategies are restricted to functional modules and to the *equational* part of system modules.

2.2 Rewriting Logic and System Modules

The type of rewriting typical of functional modules terminates with a single value as its outcome. In such modules, each step of rewriting is a step of *replacement of equals by equals*, until we find the equivalent, fully evaluated value. In general, however, a set of rewrite rules need not be terminating, and need not be Church-Rosser. That is, not only can we have infinite chains of rewriting, but we may also have highly divergent rewriting paths, that could never cross each other by further rewriting.

The essential idea of rewriting logic [37] is that the *semantics* of rewriting can be drastically changed in a very fruitful way. We no longer interpret a term t as a functional expression, but as a *state* of a system; we no longer interpret a rewrite rule $t \longrightarrow t'$ as an equality, but as a *local state transition*, stating that if a portion of a system's state exhibits the pattern described by t, then that portion of the system can change to the corresponding instance of t'. Furthermore, such a local state change can take place independently from, and therefore concurrently with, any other nonoverlapping local state changes.

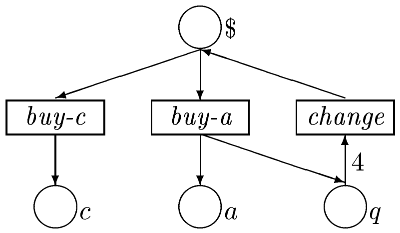
Of course, rewriting will happen *modulo* whatever equational structural axioms the state of the system satisfies. For example, the top level of a distributed system's state does often have the structure of a *multiset*, so that we can regard the system as composed together by an associative and commutative state constructor.

We can represent a *rewrite theory* as a four-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic, that specifies states of the system as an abstract data type, L is a set of labels, to label the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional [37].

Rewriting logic is therefore a logic of concurrent state change. The logic's four rules of deduction—namely, reflexivity, transitivity, congruence, and replacement (see Section 4.2.1)—allow us to infer all the complex concurrent state changes that a system may exhibit, given a set of rewrite rules that describe its elementary local changes. It then becomes natural to realize that many reactive systems so specified should never terminate, and that a system may evolve in highly nondeterministic ways through paths that will never cross each other.

The most general Maude modules are *system modules*. They specify the initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ [37]. This initial model is a transition system whose states are equivalence classes $[t]$ of ground terms modulo the equations E in $\mathcal{R}$, and whose transitions are proofs $\alpha : [t] \longrightarrow [t']$ in rewriting logic—that is, concurrent computations in the system so described. Such proofs are equated modulo a natural notion of proof equivalence that computationally corresponds to the "true concurrency" of the computations (for a detailed construction of $\mathcal{T}_{\mathcal{R}}$ see Section 4.2.2).

As a first example of a system module, we specify a simple concurrent system, namely a vending machine, as a Petri net. Petri nets have a straightforward rewriting logic semantics as initial models of their associated rewrite theories [37]; therefore, this example illustrates a general method to give executable formal specifications in Maude to Petri nets, a method that can also be naturally extended to high-level algebraic Petri nets [56]. Our Petri net represents a vending machine to buy cakes and apples; a cake costs a dollar and an apple three quarters. Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar. We can represent graphically such a machine in the conventional way as follows.



The so-called *places* of this net are cakes, apples, quarters, and dollars, denoted in the picture by circles labeled, respectively, by c, a, q, and \$. In each of these places several *tokens* can be placed. We can therefore think of the places as *slots* of our machine, in which units of a certain kind appear or disappear. Tokens in one or several of these places can then be consumed by *transitions*, denoted by rectangular boxes labeled by the transition's name, with incoming and outgoing arcs indicating which tokens are consumed and produced by the transition; we can think of such transitions as the *buttons* of our vending machine. Such transitions consume tokens from the places in their incoming arcs and produce new tokens in the places of the outgoing arcs. If several tokens must be either consumed or produced in a place, then the corresponding arc indicates the exact number. For example, the *change* transition requires four quarters to produce a dollar. The vending machine is *concurrent* because, provided enough tokens are available, we can *simultaneously push* several buttons and then can simultaneously get the combined results. For example, if we place a dollar and four quarters in the corresponding slots and push the *change* and *buy-c* buttons at once, we can simultaneously get a dollar changed and a cake as the result.

The distributed states of the machine, namely the collections of tokens available in the different places, are called *markings*. They can be naturally regarded as a *multiset* of places. Using juxtaposition notation, we can for example regard the state with one dollar and four quarters as the multiset $\$\ q\ q\ q\ q$. Meseguer and Montanari [44] observed that we can then view the Petri net as an ordinary graph, in which the transitions are the edges, and the nodes are multisets of places. Therefore, as a graph, this net has the following arcs:

$$\begin{array}{ll} \mathit{buy\text{-}c}: & \$ \longrightarrow c \\ \mathit{buy\text{-}a}: & \$ \longrightarrow a\ q \\ \mathit{change}: & q\ q\ q\ q \longrightarrow \$ \end{array}$$

The expression of this Petri net in rewriting logic is now obvious. We can view each of the labeled arcs of the Petri net as a rewrite rule in a rewrite theory having a binary *associative* and *commutative* operator `__` (multiset union), so that rewriting happens modulo such axioms, that is, it is multiset rewriting. We can gather together the "places" $\$, q, a, c$ into a sort `Place` and view the states of the net, that is, the markings, as elements of a supersort `Marking` containing `Place` and endowed with a multiset union operator with empty syntax. The corresponding Maude module then becomes

```
mod PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  op __ : Marking Marking -> Marking [assoc comm] .
  ops $ q a c : -> Place .

  rl [buy-c] : $ => c .
```

```
  rl [buy-a] : $ => a  q .
  rl [change] : q  q  q  q => $ .
endm
```

The rewrite theory (Ω, E, L, R) corresponding to a system module has a signature Ω given by the sorts, subsort relations, and operator declarations, a set E of equations, that is assumed to be decomposed as a union $E = A \cup E'$, with A a set of axioms to rewrite modulo among those supported by Maude, and E' a set of Church-Rosser and terminating equations modulo A. In the above example A consists of the associativity and commutativity axioms, E' is empty, the label set L contains the labels of the three transitions, and the set of rules R contains the above three rules, each introduced by the keyword `rl`. One can also define *conditional rules*[8], introduced by the `crl` keyword.

A key result about the representation of Petri nets as rewrite theories is that, given two markings M and M' on a net, the second is reachable from the first by a concurrent net computation if and only if the sequent $M \longrightarrow M'$ is provable in rewriting logic using the rewrite theory associated to the net [37]. Of course, net computations need not be confluent—therefore, they can be highly nondeterministic—and need not be terminating (they just happen to be terminating in this example). Therefore, the issue of *executing* rewriting logic specifications, such as those for Petri nets, or for system modules in general, is considerably more subtle than executing expressions in a functional module, for which the termination and Church-Rosser properties guarantee a unique final result regardless of the order in which equations are applied as simplification rules.

As we explain in Section 2.6, using reflection the rewriting inference process can be controlled with great flexibility in Maude by means of *strategies* that can be defined by rewrite rules at the metalevel. However, the Maude interpreter provides a *default strategy* for executing expressions in system modules. The default strategy applies the rules in a top-down fair way, and is provided by the *rewrite* command, with keyword `rewrite` or, in abbreviated form, `rew`. Since we assume that the equations E in a system module are decomposed as a union $E = A \cup E'$ with A a set of equational axioms declared as attributes of some operators to rewrite modulo, and E' a set of Church-Rosser and terminating equations modulo A, before the application of each rewrite rule the expression is simplified to its canonical form using the equations[9]; that is, it is simplified by

---

[8]Rules or conditional rules can have *extra variables* in their righthand sides that do not occur in their lefthand sides. For example, the identity rule in the one-sided sequent calculus for linear logic [25] as presented in [32] is of the form

```
rl [id] :  empty => P not P .
```

where `P` is a variable of sort `Atom`, `_,_` is the constructor for multisets of propositions and `not_` is the negation operator. However, since for applying such rules we need extra information about how the extra variables should be instantiated, rules with extra variables cannot be applied when using the default `rewrite` command (explained later in this section) to rewrite terms with Maude's default interpreter. They must be executed at the metalevel, using the `meta-apply` operator in the `META-LEVEL` module (see Section 2.5.5). In the present version of Maude, the condition of a conditional rule must satisfy the same requirements as those for the condition of a conditional equation explained in Section 2.1, including the requirement that all variables in the condition must appear in the rule's lefthand side. In a future version we plan to support more general conditions, involving extra variables and containing not only equalities and membership axiomss, but also rewrite conditions requiring that a term can rewrite to another term.

[9]By always reducing a term to canonical form using the equations before applying a rule, we could potentially miss some rewrites, in the sense that a rule could have been applied before simplifying a term, but cannot be applied after simplification. The property ensuring

applying the equations E' modulo A. Then, a rule is applied to such a simplified expression modulo the axioms A according to the default strategy. Since in our Petri net example E' is empty, this equational simplification process before each rule application becomes in this case the identity.

An expression given to Maude with the `rew` command will be rewritten with the default strategy until no more rules can be applied. Since such a computation in general may not terminate, Maude allows the user to specify the maximum number, enclosed in brackets, of rule applications allowed when executing the `rew` command. We give below several sample executions for our Petri net module with and without a bound for the `rew` command.

```
Maude> rew $ $ $ $ $ q q q q q .
result Marking: a a a c c c c

Maude> rew [1] $ $ $ $ $ q q q q q .
result Marking: $ $ $ $ $ $ q

Maude> rew [2] $ $ $ $ $ q q q q q .
result Marking: $ $ $ $ $ q q a

Maude> rew [3] $ $ $ $ $ q q q q q .
result Marking: $ $ $ $ q q a c

Maude> rew [4] $ $ $ $ $ q q q q q .
result Marking: $ $ $ q q a c c
```

Another simple, yet interesting, system module is the following `ND-INT` module, that provides nondeterministic machine integers and *nondeterministic choice*.

```
mod ND-INT is
  protecting MACHINE-INT .
  sort NdInt .
  subsort MachineInt < NdInt .
  op _?_ : NdInt NdInt -> NdInt [assoc comm] .
  var N : MachineInt .
  var ND : NdInt .
  eq N ? N = N .
  rl [choice]: N ? ND => N .
endm
```

In this example we regard a finite set of integers as a *nondeterministic integer* of sort `NdInt`, that is, as an integer that could be any of those in the set. Of course, as indicated by the subsort declaration `MachineInt < NdInt`, singleton sets are just machine integers, that is, they can be viewed as those nondeterministic integers from which any nondeterminism has already been eliminated. Union of nondeterministic integers, denoted `_?_` is associative and commutative and obeys also an idempotency equation. Nondeterministic choice is then provided by the `choice` rule.

Note that, since in membership equational logic the operators in the module are lifted to the kinds, that is, to the error supersorts, we can give expressions

that we do not miss such rewrites is called *coherence* (see [61] and Section 4.3). Coherence (or at least "weak coherence") is assumed to hold for system modules. It plays a role analogous to that played by the Church-Rosser property for functional modules.

the benefit of the doubt and therefore we can perform arithmetic operations on such nondeterministic integers as exemplified by the following Maude executions

```
Maude> rew (1 ? 5 ? 2 ? 1 ? 5) + (3 ? 11 ? 7 ? 3 ? 11) .
result NzMachineInt: 4

Maude> rew (1 ? 5 ? 2 ? 1 ? 5) * (3 ? 11 ? 7 ? 3 ? 11) .
result NzMachineInt: 3

Maude> rew [1] (1 ? 5 ? 2 ? 1 ? 5) * (3 ? 11 ? 7 ? 3 ? 11) .
result Error(NdInt): 1 * (3 ? 7 ? 11)

Maude> rew [2] (1 ? 5 ? 2 ? 1 ? 5) * (3 ? 11 ? 7 ? 3 ? 11) .
result NzMachineInt: 3
```

Note that the idempotency equation is applied before and after applying the `choice` rule. Note also that this example shows very clearly why equational logic cannot be used as the semantics of the `choice` rule, that is, why we absolutely need a rewriting logic interpretation. Indeed, if we were to consider `choice` as an equation, we would for instance have

```
2 = 2 ? 5 = 5
```

an obvious absurdity.

As yet another example of a system module we introduce below the module `SORTING` for sorting vectors of integers. In this module, such vectors are represented as sets of pairs of integers, with the first component of each pair corresponding to the vector position and the second to the number in that position.

```
mod SORTING is
  protecting MACHINE-INT .

  sorts Pair PairSet .
  subsort Pair < PairSet .

  op <_;_> : MachineInt MachineInt -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .

  vars I J X Y : MachineInt .

  crl [sort] : < J ; X > < I ; Y > => < J ; Y > < I ; X >
        if (J < I) and (X > Y) .
endm
```

Note that, by default, Maude automatically imports the predefined `BOOL` module into any other module. Therefore, the `_and_` function is available and can be used in the condition of the `sort` rule.

The top level of the state is in fact a set, namely, an element of sort `PairSet` built up by the associative and commutative operator `__`. That is, the states are sets P of pairs of integers. For the sake of the example, let us suppose that for any pair `<` i `;` x `>` in an input set P, $i < \mathrm{card}(P)$, and we cannot have two different pairs `<` i `;` x `>` and `<` j `;` y `>` in P such that $i = j$. That is, any input set P is indeed a vector of integers; of course, these requirements on P could have been explicitly specified by declaring a subsort and giving membership axioms imposing the above conditions, but this is inessential for our present purposes.

There is just one conditional rule, labeled `sort`, that modifies a vector of integers in order to put it into sorted order. The system thus described is highly concurrent, because the `sort` rule can be applied concurrently to many couples of pairs in the set representing the vector. Any complex concurrent rewriting of the set of pairs will then correspond to a proof in rewriting logic.

Using the `rew` command, we can use Maude's default interpreter for executing expressions in system modules. The Maude engine then applies the rules in a fair top-down fashion to sort a vector of integers, e.g.,

```
Maude> rew < 1 ; 3 > < 2 ; 2 > < 3 ; 1 > .
result PairSet: < 1 ; 1 > < 2 ; 2 > < 3 ; 3 >
```

Using the default interpreter we have virtually no control over the application of the rules in the module. In particular, in this example we have virtually no control over the way in which the rule `sort` is applied. Although not a problem in this case, because this specification happens to be confluent and terminating, in general we may want to control the way in which the rules are applied. Of course, if the specification is nonconfluent or nonterminating it is not only that we might want to have this control but that we need it. As already mentioned, this can be done with strategies. Section 2.6 explains how, using strategies, the rewriting inference process can be controlled in Maude for this example and for a highly nondeterministic example specifying the rules of a game. For this example such strategies correspond to specifying different *sorting algorithms* guiding where the `sort` rule should be applied at each point in the computation.

Among the many concurrent systems that we can specify as system modules in Maude, concurrent object-oriented systems are an important subclass. Maude has special syntactic conventions for specifications in this subclass, called *object-oriented modules* [38]. However, object-oriented modules are entirely reducible to ordinary system modules by a desugaring process that strips away the, very convenient, syntactic sugar. Object-oriented modules are not supported in Core Maude; they are instead supported in Full Maude, the extension of Maude written in Maude itself in which we also support parameterized modules and module expressions. They are discussed in detail in Section 3.2.

2.3 Module Hierarchies

Specifications and code should be structured in *modules* of relatively small size to facilitate understandability of large systems, increase reusability of components, and localize the effects of system changes. In Maude, the fullest support of these goals is achieved in Full Maude, which has a rich and extensible *module algebra* supporting, in particular, parameterized programming techniques in the OBJ3 style [27]. However, Core Maude provides already some useful basic support for modularity by allowing the definition of module *hierarchies*, that is, acyclic graphs of module importations. Mathematically, we can think of such hierarchies as partial orders of *theory inclusions*, that is, the theory of the importing module contains the theories of its submodules as subtheories.

Recall that a *rewrite theory* is a four-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic. As already mentioned, and further explained in Section 4, a system module is a rewrite theory with *initial semantics*.

Note that we can use the inclusion of membership equational logic into rewriting logic to view a functional module specifying an equational theory (Ω, E) as a degenerate case of a rewrite theory, namely the rewrite theory $(\Omega, E, \emptyset, \emptyset)$. In fact the initial algebra of (Ω, E) and the initial model of $(\Omega, E, \emptyset, \emptyset)$ coincide [37]. Therefore, in essence we can view all modules as rewrite theories.

The most general form of module inclusion is provided by the `including` keyword, followed by the nonempty list of imported modules and finished by a period. The `protecting` keyword is a more restricted form of inclusion, in the sense that it makes a *semantic assertion* about the relationship between the initial models of the two theories. Let $\mathcal{R} = (\Omega, E, L, R)$ be the rewrite theory specified by a system module, and let $\mathcal{R}' = (\Omega', E', L', R')$ be the theory of a supermodule, so that we have a theory inclusion $\mathcal{R} \subseteq \mathcal{R}'$. Then, we can view each model $\mathcal{M}'$ of $\mathcal{R}'$ as a model $\mathcal{M}'|_{\mathcal{R}}$ of $\mathcal{R}$, simply by disregarding the extra sorts, operations, equations, membership axioms, and rules in $\mathcal{R}' - \mathcal{R}$. Since, as further explained in Section 4, the rewrite theories $\mathcal{R}$ and $\mathcal{R}'$ have respective initial models $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}'}$, by initiality of $\mathcal{T}_{\mathcal{R}}$ we always have a unique $\mathcal{R}$-homomorphims

$$h : \mathcal{T}_{\mathcal{R}} \longrightarrow \mathcal{T}_{\mathcal{R}'}|_{\mathcal{R}}.$$

The `protecting` importation asserts that for each sort s in the signature Ω of $\mathcal{R}'$ the function h_s is an isomorphism of categories[10]. Intuitively, this means that the initial model of the supermodule does not add any "junk" or any "confusion" to the initial model of the submodule.

Of course, the `protecting` assertion cannot be checked by Maude at runtime. It requires inductive theorem proving. Using the proof techniques in [4] together with an inductive theorem prover for membership equational logic and a Church-Rosser checker such as those described in [12], this can be done for functional modules; and it seems natural to expect that these techniques and tools will extend to similar ones for rewrite theories.

By contrast, the `including` assertion does not make such requirements on h. It does, however, make *some* requirements. Namely, if the subtheory $\mathcal{R}$ does itself contain a proper subtheory $\mathcal{R}_0$ that it imports in `protecting` mode, then the inclusion $\mathcal{R}_0 \subseteq \mathcal{R}'$ does still have to be protecting. If we do not want it to be, we have to say so by explicitly listing the module defining $\mathcal{R}_0$ in the list of modules imported in `including` mode.

We give below an example of a module hierarchy, namely, the number hierarchy from the naturals to the rationals in a somewhat different form than in Appendix C.7 of [27]. This hierarchy happens to be a linear order of theory inclusions, with `BOOL` implicitly at the bottom. In general, any partial order of inclusions can be defined in the same way. Note that all the importations are `protecting` importations. The `including` importation, although possible in Core Maude, is more natural in the context of a module *renaming* operation. Indeed, if the semantics of a module is going to be modified by a supermodule it is better to make a *copy* of such a module and import the copy. At present, Core Maude does not support renaming; it is supported by Full Maude (see Section 3.5.4). Renaming will probably be added to Core Maude in a future version.

[10] In the models of a rewrite theory the sorts are interpreted as categories, thus the requirement; for functional theories the requirement becomes that each such h_s is bijective. Note that the expected condition would have been to require h to be an $\mathcal{R}$-*isomorphism*. However, due to the presence of error elements at the kind level, the isomorphism condition would be too strong, since in general, when enlarging a signature, there will be new error terms that cannot be proved equal to old ones. See [4] for a detailed discussion of, and proof techniques for, protecting extensions in membership equational logic.

```
fmod NAT is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat [comm] .
  op _*_ : NzNat NzNat -> NzNat [comm] .
  op _>_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : NzNat NzNat -> NzNat [comm] .
  vars N M : Nat .
  vars N' M' : NzNat .
  eq p s N = N .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq N * 0 = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
  eq d(0, N) = N .
  eq d(s N, s M) = d(N, M) .
  ceq quot(N, M') = s quot(d(N, M'), M') if N > M' .
  eq quot(M', M') = s 0 .
  ceq quot(N, M') = 0 if M' > N .
  eq gcd(N', N') = N' .
  ceq gcd(N', M') =  gcd(d(N', M'), M') if N' > M' .
endfm

fmod INT is
  sorts Int NzInt .
  protecting NAT .
  subsort Nat < Int .
  subsorts NzNat < NzInt < Int .
  op -_ : Int -> Int .
  op -_ : NzInt -> NzInt .
  op _+_ : Int Int -> Int [comm] .
  op _*_ : Int Int -> Int [comm] .
  op _*_ : NzInt NzInt -> NzInt [comm] .
  op quot : Int NzInt -> Int .
  op gcd : NzInt NzInt -> NzNat [comm] .
  vars I J : Int .
  vars I' J' : NzInt .
  vars N' M' : NzNat .
  eq - - I = I .
  eq - 0 = 0 .
  eq I + 0 = I .
  eq M' + (- M') = 0 .
  ceq M' + (- N') = - d(N', M') if N' > M' .
```

```
  ceq M' + (- N') = d(N', M') if M' > N' .
  eq (- I) + (- J) = - (I + J) .
  eq I * 0 = 0 .
  eq I * (- J) = - (I * J) .
  eq quot(0, I') = 0 .
  eq quot(- I', J') = - quot(I', J') .
  eq quot(I', - J') = - quot(I', J') .
  eq gcd(- I', J') = gcd(I', J') .
endfm

fmod RAT is
  sorts Rat NzRat .
  protecting INT .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat .
  op -_  : NzRat -> NzRat .
  op _+_ : Rat Rat -> Rat [comm] .
  op _*_ : Rat Rat -> Rat [comm] .
  op _*_ : NzRat NzRat -> NzRat [comm] .
  vars I' J' : NzInt . vars R S : Rat .
  vars R' S' : NzRat .
  eq R / (R' / S') = (R * S') / R' .
  eq (R / R') / S' = R / (R' * S') .
  ceq J' / I'
    = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
      if gcd(J', I') > s 0 .
  eq R / s 0 = R .
  eq 0 / R' = 0 .
  eq R / (- R') = (- R) / R' .
  eq - (R / R') = (- R) / R' .
  eq R + (S / R') = ((R * R') + S) / R' .
  eq R * (S / R') = (R * S) / R' .
endfm
```

2.4 Some Predefined Modules

Maude has a standard library of predefined modules that, by default, are entered into the system at the beginning of each session, so that any of these predefined modules can be imported by any other module defined by the user. Also, by default the predefined functional module `BOOL` is automatically imported as a submodule of any user-defined module, unless such importation is explicitly disabled. We discuss below some of the basic predefined modules in the standard library; some of them have a syntax similar to that of their counterparts in OBJ3's standard prelude [27]. The `META-LEVEL` module is discussed in Section 2.5, and the `LOOP-MODE` module in Section 2.8. The entire standard library of predefined modules is included as Appendix D.

2.4.1 Truth and Booleans

There are three predefined modules involving truth values, namely, TRUTH-VALUE, TRUTH, and BOOL. The most basic one is TRUTH-VALUE, which has the following definition.

```
fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [special (id-hook SystemTrue)] .
  op false : -> Bool [special (id-hook SystemFalse)] .
endfm
```

That is, the module just declares two constants, true and false. The key thing to note is the special attribute associated to each of the operator declarations for these constants. This states that the constants are treated as *built-in* operators, so that instead of having the standard treatment of any user-defined operator they are instead associated to appropriate C++ code by "hooks" as stated next to the special attribute. This is important, because certain basic constructs of the language such as conditions in a conditional equation, membership axiom, or rule, and also sort predicates associated to membership assertions evaluate to these built-in truth values.

In general, many operators in predefined modules are special operators. In what follows, to lighten the exposition, we will omit the details about such hooks in special operators writing [special (...)] instead. The full definitions can be found in Appendix D.

The module TRUTH adds a number of important operators to TRUTH-VALUE.

```
fmod TRUTH is
  protecting TRUTH-VALUE .

  op if_then_else_fi : Bool Universal Universal -> Universal
        [special ( ... )] .
  op _==_ : Universal Universal -> Bool
        [prec 51 special ( ... )] .
  op _=/=_ : Universal Universal -> Bool
        [prec 51 special ( ... )] .
endfm
```

The prec attribute in the last two operators gives a precedence to the operator for parsing purposes (see Section 2.7.4). The operators are, respectively, if_then_else_fi, and the built-in equality and inequality predicates. These operators are special in a number of ways. Firstly, they are automatically added to every module. Secondly, they are *polymorphic*, so that, for each module, they can be considered to be normal operators that are ad-hoc overloaded for each connected component in the module. In fact, Universal is not a normal sort, but should instead be understood as a polymorphic sort whose concrete effect is the instantiation of the corresponding operators in each connected component. These operators have the same semantics as their OBJ3 counterparts except that if if_then_else_fi fails to rewrite at the top, it then evaluates its *then* and *else* arguments. In particular, the equality and inequality predicates are evaluated by reducing two ground terms to their normal form and comparing the results for equality, modulo the equational axioms in the attributes of the operators in the module.

Note that the equality and inequality predicates that the module TRUTH adds to each connected component of a user-defined module in a built-in and efficient way could in principle have been defined in a more cumbersome and inefficient way by the user. In fact, assuming as we usually do that the equations and membership axioms in the user module are Church-Rosser and terminating modulo the axioms in operator attributes, the corresponding initial algebra is a *computable* algebraic data type, for which equality and inequality are also computable functions. Therefore, by a well-known metaresult of Bergstra and Tucker [1], such equality and inequality predicates can themselves be equationally defined by Church-Rosser and terminating equations. It is of course very convenient, and much more efficient, to unburden the user from having to give those explicit equational definitions of equality and inequality, by providing them in a built-in way.

Note also that, by the above metaargument, the use of inequality predicates in equations, membership axioms, or conditions, does not involve any real introduction of *negation* in the underlying membership equational logic, which remains a Horn logic. What we are really doing is adding more Boolean-valued functions to the module, but such functions, although provided in a built-in way for convenience and efficiency, could have been equationally defined without any use of negation.

The module `BOOL` imports `TRUTH` and adds the usual conjunction, disjunction, exclusive or, negation, and implication operators. Such operators are defined entirely equationally.

```
fmod BOOL is
  protecting TRUTH .

  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .

  vars A B C : Bool .

  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
```

By default, the `BOOL` module is included as a submodule of any other module defined by the user. This is accomplished by the command

```
set include BOOL on .
```

which can mention any module we wish to include—in this case `BOOL`—and is set when the standard library is entered. However, this default inclusion can be disabled. For example, if the user wished to have the polymorphic equality and `if_then_else_fi` operators automatically added to modules, but wanted to exclude the usual Boolean connectives for the built-in truth values, he/she could write

```
set include BOOL off .
set include TRUTH on .
```

The last module involving truth values is the IDENTICAL module. It is not included by default in other modules. That is, it has to be imported explicitly if it is needed. When imported into a module, it adds to each of its connected components polymorphic operators for *syntactic* equality and inequality. That is, two ground terms are compared for syntactic equality—modulo the equational axioms in the attributes of the operators in the module—without performing any reduction of the terms by the equations in the module.

Note that what this module provides in a built-in way would require a considerably more cumbersome, and subtle, explicit definition at the user level. In fact, given that equality in a functional module is always *semantic* equality using the equations, to explicitly define the above operators the entire signature of the module would have to be *duplicated* in a disjoint copy, for which *no equations* would be given, except for the equational axioms in operator attributes. Then, the above syntactic operators would reduce to the standard semantic equality and inequality operators on *that* equationless disjoint copy of the signature.

```
fmod IDENTICAL is
  op _===_ : Universal Universal -> Bool
        [prec 51 strat (0) special ( ... )] .

  op _=/==_ : Universal Universal -> Bool
        [prec 51 strat (0) special ( ... )] .
endfm
```

2.4.2 The Machine Integers

The machine integers are defined below. The constants in this module represent the C++ data type *int*, as elements of a sort MachineInt, with a subsort NzMachineInt of nonzero machine integers. The first two operator declarations illustrate the way of linking the built-in integer constants to the sorts MachineInt and NzMachineInt. The other operations declared in the module represent their C++ counterparts with the same notation; their meaning is indicated in the associated comments. The division and remainder operations produce an unreduced error term if their second argument is zero. The machine integers provide a fast arithmetic data type for general purpose programming and for metalevel hooks into the rewriting engine.

```
fmod MACHINE-INT is
  sorts MachineInt NzMachineInt .
  subsort NzMachineInt < MachineInt .
  op <MachineInts> : -> NzMachineInt [special ( ... )] .
  op <MachineInts> : -> MachineInt [special ( ... )] .
  op -_ : MachineInt -> MachineInt
      [prec 15 special ( ... )] .     *** minus
  op -_ : NzMachineInt -> NzMachineInt
      [prec 15 special ( ... )] .     *** minus
  op ~_ : MachineInt -> MachineInt
      [prec 15 special ( ... )] .     *** bitwise complement
  op _+_ : MachineInt MachineInt -> MachineInt
      [prec 33 gather (E e) special ( ... )] . *** addition
  op _-_ : MachineInt MachineInt -> MachineInt
```

```
        [prec 33 gather (E e) special ( ... )] . *** difference
    op _*_ : MachineInt MachineInt -> MachineInt
        [prec 31 gather (E e) special ( ... )] . *** multiplication
    op _*_ : NzMachineInt NzMachineInt -> NzMachineInt
        [prec 31 gather (E e) special ( ... )] . *** multiplication
    op _/_ : MachineInt NzMachineInt -> MachineInt
        [prec 31 gather (E e) special ( ... )] . *** division
    op _%_ : MachineInt NzMachineInt -> MachineInt
        [prec 31 gather (E e) special ( ... )] . *** remainder
    op _&_ : MachineInt MachineInt -> MachineInt
        [prec 53 gather (E e) special ( ... )] . *** bitwise and
    op _|_ : MachineInt MachineInt -> MachineInt
        [prec 57 gather (E e) special ( ... )] . *** bitwise or
    op _|_ : NzMachineInt NzMachineInt -> NzMachineInt
        [prec 57 gather (E e) special ( ... )] . *** bitwise or
    op _^_ : MachineInt MachineInt -> MachineInt
        [prec 55 gather (E e) special ( ... )] . *** bitwise xor
    op _>>_ : MachineInt MachineInt -> MachineInt
        [prec 35 gather (E e) special ( ... )] . *** left shift
    op _<<_ : MachineInt MachineInt -> MachineInt
        [prec 35 gather (E e) special ( ... )] . *** right shift
    op _<_ : MachineInt MachineInt -> Bool
        [prec 37 special ( ... )] .     *** less than
    op _<=_ : MachineInt MachineInt -> Bool
        [prec 37 special ( ... )] .     *** less or equal than
    op _>_ : MachineInt MachineInt -> Bool
        [prec 37 special ( ... )] .     *** greater than
    op _>=_ : MachineInt MachineInt -> Bool
        [prec 37 special ( ... )] .     *** greater or equal than
  endfm
```

2.4.3 Quoted Identifiers

Quoted identifiers have the following signature:

```
  fmod QID is
    protecting MACHINE-INT .
    sort Qid .
    op <Qids> : -> Qid [special ( ... )] .
    op conc : Qid Qid -> Qid [special ( ... )] .
    op index : Qid MachineInt -> Qid [special ( ... )] .
    op strip : Qid -> Qid [special ( ... )] .
  endfm
```

Typical constants of sort `Qid` are quoted identifiers such as `'a`, `'aa`, `'f`(x`)`, `'-1`, `''123abc`, and `'A_quoted_identifier`. Every quoted identifier is a legal Maude identifier. That is, it satisfies the conventions for Maude identifiers explained in Section 2.1.1 and, in addition, it begins with the quote character. Of course, in syntax declarations for sorts, variables, etc., of a module that includes `QID` we should avoid using quoted names, since they are now used for constants of sort `Qid`. In fact, that is the whole point of using a module of quoted identifiers instead of a module of general identifiers, since that could create massive ambiguities.

Quoted identifiers are defined in a built-in way by the first operator declaration. The operation `conc` concatenates two quoted identifiers, omitting the quote of the second. The operation `index` appends the result of the machine integer expression at the end of the quoted identifier. The operation `strip` strips off the first character after the quote. Their semantics can be inferred from the following examples:

```
conc('a, 'b) = 'ab
conc('a, '42) = 'a42
index('a, 2 * 21) = 'a42
conc('a, index(' , 1 - 43)) = 'a-42
strip('abcd) = 'bcd
```

For different purposes it is useful to have not only quoted identifiers, but also a data type of lists of quoted identifiers. In particular, the remaining two predefined modules in the standard library, namely `META-LEVEL` and `LOOP-MODE`—discussed in Sections 2.5 and 2.8, respectively—both import the following `QID-LIST` module.

```
fmod QID-LIST is
  protecting QID .
  sort QidList .
  subsort Qid < QidList .
  op nil : -> QidList .
  op __ : QidList QidList -> QidList [assoc id: nil] .
endfm
```

2.5 Reflection and the `META-LEVEL`

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [14, 15, 10]. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way.

A naive implementation of reflection can be very expensive both in time and memory use. Therefore, a good implementation must provide efficient ways of performing reflective computations. This section explains how this is achieved in Maude through its predefined `META-LEVEL` module. We first discuss the semantics of metalevel computations, and how their efficiency can be dramatically increased by conservatively extending the universal theory $\mathcal{U}$ to a metalevel theory $\mathcal{M}$ with *descent* functions and rules that allow lowering deductions at higher levels of reflection to much more efficient deductions at lower levels. Then, we explain how terms and modules are meta-represented in `META-LEVEL`, and how these semantic principles are supported in important special cases by the `META-LEVEL` module in a built-in way.

The important topic of internal strategy languages, that use reflection in an essential way, is discussed separately, in Section 2.6. Besides strategies, reflection makes possible many advanced metaprogramming applications. One important such application is Full Maude which, as discussed later in this document, makes essential use of reflection to provide Maude with a rich and extensible *module algebra*; this is a special instance of a general class of applications in which, using reflection, we can use Maude as a *metalanguage* to reify other languages and logics within rewriting logic (see Section 2.8.2). The paper [11] summarizes, and gives references for, a number of other important applications, including uses of rewriting logic as a *logical framework* and the development of theorem proving tools.

2.5.1 Reflection and Metalevel Computation

Rewriting logic is reflective [14, 10] in a precise mathematical way, namely, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $\langle\overline{\mathcal{R}}, \overline{t}\rangle$, in such a way that we have the following equivalence

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

Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$\mathcal{R} \vdash t \to t' \ \Leftrightarrow\ \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \to \langle\overline{\mathcal{R}}, \overline{t'}\rangle \ \Leftrightarrow\ \mathcal{U} \vdash \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t}\rangle}\rangle \to \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t'}\rangle}\rangle \ldots$$

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible—so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

To achieve a systematic descent into equivalent rewriting computations at lower levels, the key idea is to exploit the equivalence ($\dagger$). A detailed proof of this equivalence has been given for the case of unconditional and unsorted theories [10]. The extension to the case of interest for Maude—namely to conditional rewrite theories with membership equational logic [41, 5] as the underlying equational logic—although nontrivial, is essentially unproblematic. We therefore assume a universal theory $\mathcal{U}$ for this more general class of finitely presented rewrite theories. In particular, the signature $\Sigma_{\mathcal{U}}$ of $\mathcal{U}$ has sorts *Term*, *Module*, and *Kind*, whose respective elements $\overline{t} : \mathit{Term}$, $\overline{\mathcal{R}} : \mathit{Module}$, and $\overline{K} : \mathit{Kind}$ represent terms, rewrite theories, and kinds[11] in a signature, respectively. We assume that there is also an equationally defined Boolean predicate $\mathit{parse} : \mathit{Module} \times \mathit{Kind} \times \mathit{Term} \longrightarrow \mathit{Bool}$ so that $\mathit{parse}(\overline{\mathcal{R}}, \overline{K}, \overline{t}) = \mathit{true}$ if t is an $\mathcal{R}$-term of kind K, and $\mathit{parse}(\overline{\mathcal{R}}, \overline{K}, \overline{t}) = \mathit{false}$ otherwise.

We can exploit the equivalence ($\dagger$) by introducing the notion of *descent function*, that is, a function that, given metalevel representations for a rewrite theory $\mathcal{R}$ and a term t in it, rewrites such a term in $\mathcal{R}$ according to a given strategy and returns the meta-representation of the resulting term. Such functions can be simply expressed in terms of a general *sequential interpreter function* I for rewriting logic. This is a *partial* function that takes three arguments: a finitely presented rewrite theory $\mathcal{R}$, a term t, and a deterministic strategy S. In case of termination it returns either the term t' to which t was rewritten according

[11] In a membership equational logic signature, terms always have a *kind*; they may or may not have a *sort* of that kind. As already mentioned, in Maude kinds are represented as *error sorts*, that are added by the system at the top of each connected component of sorts defined by the user.

to S, or an error message that is not a term in $\mathcal{R}$. The function is undefined in case the strategy does not terminate. For any finitely presented rewrite theory $\mathcal{R}$, terms t, t' in it, and admissible deterministic strategy S, any such interpreter function must of course satisfy the correctness requirement

$$(\flat) \quad I(\mathcal{R}, t, S) = t' \quad \Rightarrow \quad \mathcal{R} \vdash t \longrightarrow t'.$$

The point is that, regardless of the particular details of I, we can always equationally axiomatize any such effective interpreter function by means of a Church-Rosser, but in general nonterminating, finitary equational theory $\mathcal{I}$. This can be done in a signature that we can assume contains $\Sigma_{\mathcal{U}}$ as a subsignature. By extending our universal theory $\mathcal{U}$ with the new sorts, operations, and equations of $\mathcal{I}$, we obtain an extended rewrite theory $\mathcal{U} \cup \mathcal{I}$. A *descent function* is then a function

$$d : \mathit{Module} \times \mathit{Term} \times \mathit{Parameters} \longrightarrow \mathit{Term}$$

such that there is a deterministic strategy expression S_d with a single free variable of sort *Parameters* satisfying the equality

$$d(\overline{\mathcal{R}}, \overline{t}, p) = \overline{I(\mathcal{R}, t, S_d(p))}.$$

Such descent functions are of course easily definable equationally as definitional extensions of the theory $\mathcal{U} \cup \mathcal{I}$. Note that, since we have only added some new equations, the only rewrite rules in $\mathcal{U} \cup \mathcal{I}$ are exactly those in $\mathcal{U}$. But, given a descent function d, we can now exploit the equivalence (†) by adding to $\mathcal{U} \cup \mathcal{I}$ a *descent rule*

$$\begin{array}{l} d : \langle M, x \rangle \longrightarrow \langle M, y \rangle \\ \quad \mathit{if}\ \mathit{parse}(M, K, x) = \mathit{true} \wedge \mathit{parse}(M, K, y) = \mathit{true} \wedge d(M, x, p) = y. \end{array}$$

where $M : \mathit{Module}$, $x, y : \mathit{Term}$, $K : \mathit{Kind}$, and $p : \mathit{Parameters}$. The equivalence (†) can be exploited for efficiency reasons with such a rule, because the sequential interpreter I can be a built-in function such as the Maude interpreter; therefore, instantiating M with $\overline{\mathcal{R}}$, we can use efficient deduction in $\mathcal{R}$ to perform deduction in $\mathcal{U}$. Let $\mathcal{M}$ denote a rewrite theory of the form $\mathcal{M} = \mathcal{U} \cup \mathcal{I} \cup \mathcal{D}$, where $\mathcal{D}$ is the addition of several descent functions and of their associated descent rules. We shall call $\mathcal{M}$ a *metalevel theory.*

The addition of descent rules to $\mathcal{U}$ is of course *conservative*, in the sense of not adding any rewrites that could not be performed, albeit less efficiently, in $\mathcal{U}$ itself, since for any descent rule d we have[12],

$$\begin{array}{rcl} \mathcal{M} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \stackrel{d}{\longrightarrow} \langle \overline{\mathcal{R}}, \overline{t'} \rangle & \Rightarrow & I(\mathcal{R}, t, S_d(p)) = t' \\ & \stackrel{\flat}{\Rightarrow} & \mathcal{R} \vdash t \longrightarrow t' \\ & \stackrel{\dagger}{\Leftrightarrow} & \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle. \end{array}$$

Note that, by applying several descent functions, we can descend several levels in the reflective tower. Assume that $\mathcal{M}$ includes descent functions d and d', and

[12] Of course, to ensure conservativity we also should assume that the new equations in $\mathcal{M}$ do not disturb the equality of terms or the rewriting relation in $\mathcal{U}$. Since the equations are assumed to be Church-Rosser, conservativity of equality can be easily achieved by assuming that the tops of lefthand sides of new equations are new function symbols. Preservation of the rewriting relation can be achieved by forbidding nonvariable overlaps between lefthand sides of new equations and of rules, as done in [10].

let $\mathcal{R}$ and t be an arbitrary rewrite theory and a term in it; then we have, for example,

$$\begin{aligned} d(\overline{\mathcal{M}}, \overline{d'(\overline{\mathcal{R}}, \bar{t}, p')}, p) &= \overline{I(\mathcal{M}, d'(\overline{\mathcal{R}}, \bar{t}, p'), S_d(p))} \\ &= \overline{I(\mathcal{M}, \overline{I(\mathcal{R}, t, S_{d'}(p'))}, S_d(p))}. \end{aligned}$$

That is, a meta-metalevel computation can be efficiently carried out at the object level. An example of this kind of combined descent is given in Section 3.3. More generally, we should view descent functions as *basic strategies*, that can be used as fundamental building blocks to define *internal strategy languages* [16, 10], in which they can be combined with each other and with more complex strategies at several levels of reflection to perform efficiently sophisticated metalevel computations (see section 2.6).

2.5.2 The Module `META-LEVEL`

In Maude, key functionality of a metalevel theory $\mathcal{M}$ with several descent functions has been efficiently implemented in a functional module `META-LEVEL`, by using as the interpreter function I Maude's own interpreter. Furthermore, several other useful functions of the universal theory $\mathcal{U}$ are also built-in for efficiency reasons.

We summarize below the key functionality provided by `META-LEVEL`. We recall that Maude's *functional modules* are equational theories that are assumed to be Church-Rosser and terminating modulo some axioms for which matching algorithms are available in the implementation, and that *system modules* are rewrite theories whose equational part satisfies the same requirements as a functional module, and where the equations and the rules are assumed to be *weakly coherent* [61, 60] modulo the axioms (see Section 4.3). In `META-LEVEL`:

- Maude terms are reified as elements of a data type `Term` of terms;
- Maude modules are reified as terms in a data type `Module` of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a descent function `meta-reduce`;
- the process of applying a rule of a system module to a subject term is reified by a descent function `meta-apply`;
- the process of rewriting a term in a system module using Maude's default interpreter is reified by a descent function `meta-rewrite`; and
- parsing and pretty printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also reified by corresponding metalevel functions.

`META-LEVEL` imports the module `QID-LIST` (lists of quoted identifiers) from the standard library of predefined modules, which contains in turn the modules `QID`, `MACHINE-INT`, and `BOOL`. We first introduce the syntax used in `META-LEVEL` for representing terms; then we explain how modules are represented; and finally we discuss the different built-in functions, namely, the descent functions, and the parsing and sort functions.

2.5.3 Representing Terms

Terms are reified as elements of the data type `Term` of terms, with the following signature

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op _:_ : Term Qid -> Term .
op _::_ : Term Qid -> Term .
op error* : -> Term .
```

The first declaration, making `Qid` a subsort of `Term`, is used to represent variables by the corresponding quoted identifiers. Thus, the variable `N` is represented by `'N`. The operator `{_}_` is used for representing constants essentially as pairs, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant `0` in the module `NAT` in Section 2.5.4 below is represented as `{'0}'Nat`. The operator `_[_]` corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form, and the second argument the list of its subterms, where list concatenation is denoted `_,_`. For example, the term `s s 0 + s 0` of sort `Nat` in module `NAT` is meta-represented as

```
'_+_['s_['s_[{'0}'Nat]], 's_[{'0}'Nat]].
```

Since terms in the module `META-LEVEL` can be meta-represented just as terms in any other module, as already mentioned when discussing the universal theory, the representation of terms can be iterated. For example, the meta-meta-representation $\overline{\overline{\texttt{s 0}}}$ of the term `s 0` in `NAT` is the term

```
'_[_]['{''s}'Qid,'{_}_['{''0}'Qid,{''Nat}'Qid]].
```

For the most part, the meta-representation of terms involving built-in operators proceeds as for any other term. For example, the application of the equality predicate `_==_` in the term `s s 0 == s 0` is represented as the term

```
'_==_['s_['s_[{'0}'Nat]], 's_[{'0}'Nat]].
```

But since we can think of membership predicates $t : s$ not as unary predicates, one for each sort s, but as a binary predicate with the second argument varying over sorts, it is natural to meta-represent them in a uniform way by means of a binary constructor `_:_` with the first argument the representation of the term, and the second the representation of the sort as a quoted identifier. For example, the membership predicate `s 0 : Nat` is represented as the term

```
's_[{'0}'Nat] : 'Nat.
```

Similarly, there is also a binary constructor `_::_` for meta-representing the "lazy" membership predicate that does not evaluate the term in question at all, but uses only the syntactic declarations in the module's order-sorted signature and the membership axioms to decide whether the least sort (see, e.g., [26]) of the unreduced term is smaller or equal to a given sort. The last declaration for the data type of terms is a constant `error*` to be used as an error element.

2.5.4 Representing Modules

Functional and system modules are meta-represented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now meta-represented as we have already explained; (2) in the meta-representation of modules we follow a *fixed order* in introducing the different kinds of declarations for sorts, subsort relations, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operator `_;_`.

To motivate the general syntax for representing modules, we illustrate it with a simple example—namely, a module `NAT` for natural numbers with zero and successor and with commutative addition and multiplication operators.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
  eq 0 * N = 0 .
  eq s N * M = M + (N * M) .
endfm
```

The syntax for the top-level operators representing functional and system modules is as follows

```
sorts FModule Module .
subsort FModule < Module .

op fmod_is_______endfm : Qid ImportList SortDecl
      SubsortDeclSet OpDeclSet
      VarDeclSet MembAxSet EquationSet -> FModule .

op mod_is________endm : Qid ImportList SortDecl
      SubsortDeclSet OpDeclSet
      VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

The representation $\overline{\texttt{NAT}}$ of `NAT` in `META-LEVEL` is the term

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
  var 'N : 'Nat .
  var 'M : 'Nat .
```

```
    none
    eq '_+_[{'0}'Nat, 'N] = 'N .
    eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
    eq '_*_[{'0}'Nat, 'N] = {'0}'Nat .
    eq '_*_['s_['N], 'M] = '_+_['N, '_*_['N, 'M]] .
  endfm
```

of sort `FModule`. Since `NAT` has no list of imported submodules and no membership axioms, those fields are filled respectively by the constants `nil` of sort `ImportList`, and `none` of sort `MembAxSet`. Similarly, since the zero and successor operators have no attributes, they have the empty set of attributes `none`.

The full definition of `META-LEVEL` is given in Appendix D. Part of the syntax for functional and system modules is listed below. It extends in a natural way the fragment illustrated in the above example and should, for the most part, be self-explanatory by comparison with the Core Maude syntax, which is mirrored quite closely. Note that we have to represent the set of attributes of an operator. For this, sorts `Attr` and `AttrSet` are used. Such attributes may be equational axioms to rewrite modulo, syntactic attributes for parsing purposes, or attributes to link operators to built-in functions (see Sections 2.4.1 and 2.4.2).

```
  sorts FModule Module ModuleExpression Import ImportList
    MachineIntList QidSet Sort SortDecl SubsortDecl
    SubsortDeclSet Attr AttrSet OpDecl OpDeclSet VarDecl
    VarDeclSet Term TermList Equation EquationSet Rule
    RuleSet MembAx MembAxSet Hook HookList .
  subsort FModule < Module .
  subsort Import < ImportList .
  subsort Qid < ModuleExpression .
  subsort Qid < QidList .
  subsort Qid < QidSet .
  subsort Qid < Sort .
  subsort MachineInt < MachineIntList .
  subsort SubsortDecl < SubsortDeclSet .
  subsort Attr < AttrSet .
  subsort OpDecl < OpDeclSet .
  subsort VarDecl < VarDeclSet .
  subsort Equation < EquationSet .
  subsort Rule < RuleSet .
  subsort MembAx < MembAxSet .
  subsort Hook < HookList .

  op none : -> QidSet .
  op _;_ : QidSet QidSet -> QidSet [assoc comm id: none] .

  op nil : -> MachineIntList .
  op __ : MachineIntList MachineIntList -> MachineIntList
        [assoc id: nil] .

  op nil : -> ImportList .
  op __ : ImportList ImportList -> ImportList [assoc id: nil] .
  op including_. : ModuleExpression -> Import .

  op sorts_. : QidSet -> SortDecl .
```

```
  op subsort_<_. : Qid Qid -> SubsortDecl .
  op none : -> SubsortDeclSet .
  op __ : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
          [assoc comm id: none] .

  op op_:_->_[_]. : Qid QidList Qid AttrSet -> OpDecl .
  op none : -> OpDeclSet .
  op __ : OpDeclSet OpDeclSet -> OpDeclSet
          [assoc comm id: none] .

  op none : -> AttrSet .
  op __ : AttrSet AttrSet -> AttrSet [assoc comm id: none] .
  ops assoc comm idem : -> Attr .
  ops id left-id right-id : Term -> Attr .
  op strat : MachineIntList -> Attr .
  op prec : MachineInt -> Attr .
  op gather : QidList -> Attr .

  op special : HookList -> Attr .
  op __ : HookList HookList -> HookList [assoc] .
  op id-hook : Qid QidList -> Hook .
  op op-hook : Qid Qid QidList Qid -> Hook .
  op term-hook : Qid Term -> Hook .

  op var_:_. : Qid Qid -> VarDecl .
  op none : -> VarDeclSet .
  op __ : VarDeclSet VarDeclSet -> VarDeclSet
          [assoc comm id: none] .

  op mb_:_. : Term Qid -> MembAx .
  op cmb_:_if_=_. : Term Qid Term Term -> MembAx .
  op none : -> MembAxSet .
  op __ : MembAxSet MembAxSet -> MembAxSet
          [assoc comm id: none] .

  op eq_=_. : Term Term -> Equation .
  op ceq_=_if_=_. : Term Term Term Term -> Equation .
  op none : -> EquationSet .
  op __ : EquationSet EquationSet -> EquationSet
          [assoc comm id: none] .

  op rl[_]:_=>_. : Qid Term Term -> Rule .
  op crl[_]:_=>_if_=_. : Qid Term Term Term Term -> Rule .
  op none : -> RuleSet .
  op __ : RuleSet RuleSet -> RuleSet [assoc comm id: none] .
```

Note that—just as in the case of terms—terms of sort `Module` can be meta-represented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels. A good example is the inductive theorem prover described in [12], where modules are meta-represented as terms in the inference rules for induction, but they have to be meta-meta-represented as terms of sort `Term` when used in strategies that control the application of the inductive inference rules. We illustrate the meta-meta-representation of modules with a simple example, namely a module `TRUTH-VALUES` of truth values

```
fmod TRUTH-VALUES is
  sorts Truth .
  ops t f : -> Truth .
endfm
```

whose meta-meta-representation $\overline{\overline{\overline{\texttt{TRUTH-VALUES}}}}$ is the following term of sort `Term`

```
'fmod_is_______endfm[
   {''TRUTH-VALUES}'Qid,
   {'nil}'ImportList,
   'sorts_.[{''Truth}'Qid],
   {'none}'SubsortDeclSet,
   '__['op_:_->_`[_`].[
          {''t}'Qid, {'nil}'QidList, {''Truth}'Qid,
          {'none}'AttrSet],
       'op_:_->_`[_`].[
          {''f}'Qid, {'nil}'QidList, {''Truth}'Qid,
          {'none}'AttrSet]],
   {'none}'VarDeclSet,
   {'none}'MembAxSet,
   {'none}'EquationSet]
```

2.5.5 Descent Functions

`META-LEVEL` has three built-in descent functions, `meta-reduce`, `meta-apply`, and `meta-rewrite`, that provide three useful and efficient ways of reducing metalevel computations to object-level ones.

The operation `meta-reduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t, of a membership predicate $t : s$, or of a lazy membership predicate $t :: s$, in that module. It has syntax

```
op meta-reduce : Module Term -> Term [special ( ... )] .
```

When the second argument is the representation $\overline{t}$ of a term t in $\mathcal{R}$, the function `meta-reduce` returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, e.g.,

$$\texttt{Maude> red meta-reduce(}\overline{\texttt{NAT}}\texttt{, }\overline{\texttt{s 0 + 0}}\texttt{) .}$$
$$\texttt{result Term: }\overline{\texttt{s 0}}$$

Note that in order to simplify the presentation we use the meta-notations $\overline{t}$ and $\overline{Id}$ for t a term and Id the name of a module. As explained in Section 3.4, in Full Maude we can use the `up` command to get the meta-representation denoted by the overline notation. In Core Maude, however, such a meta-representation has to be explicitly given, that is, the example above must be written as follows.

```
Maude> red meta-reduce(
              fmod 'NAT is
                 nil
                 sorts 'Zero ; 'Nat .
                 subsort 'Zero < 'Nat .
                 op '0 : nil -> 'Zero [none] .
```

```
                  op 's_ : 'Nat -> 'Nat [none] .
                  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
                  op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
                  var 'N : 'Nat .
                  var 'M : 'Nat .
                  none
                  eq '_+_[{'0}'Nat, 'N] = 'N .
                  eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
                  eq '_*_[{'0}'Nat, 'N] = {'0}'Nat .
                  eq '_*_['s_['N], 'M] = '_+_['N, '_*_['N, 'M]] .
                endfm,
                '_+_['s_[{'0}'Nat], {'0}'Nat]) .
  result Term: 's_[{'0}'Zero]
```

This is particularly cumbersome for the meta-representation of modules, which can be quite big. However, as illustrated by the examples in Section 2.6, one easy solution is to define a new module importing `META-LEVEL` in which we introduce a new constant of sort `Module` or `FModule` to name the module in question—in this example, the constant `NAT`—and then give an equation identifying such a constant with the meta-representation of the given module.

Similarly, when the second argument of `meta-reduce` is the representation of a membership predicate $t : s$ (or a lazy membership predicate $t :: s$) the term t is fully reduced using the equations in $\mathcal{R}$ and the least sort of the reduced term is computed (respectively, the least sort of the term t according to the order-sorted signature and the membership axioms of the module $\mathcal{R}$ is computed) and then the representation of the Boolean value of the corresponding predicate is returned.

The interpreter function for $\texttt{meta-reduce}(\overline{\mathcal{R}},\overline{t})$ rewrites the term t to normal form using only the equations in $\mathcal{R}$, and does so according to the operator evaluation strategies (see Section 2.1.3 and [21]) declared for each operator in the signature of $\mathcal{R}$—which by default is bottom-up for operators with no such strategies declared. In other words, the interpreter strategy for this function coincides with that of the `red` command in Maude, that is,

$$\texttt{meta-reduce}(\overline{\mathcal{R}},\overline{t}) = \overline{I_{Maude}(\mathcal{R},t,\texttt{red})}.$$

The operation `meta-rewrite` has syntax

```
  op meta-rewrite : Module Term MachineInt -> Term
        [special ( ... )] .
```

It is entirely analogous to `meta-reduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude's default strategy. Its first two arguments are the representations of a module $\mathcal{R}$ and of a term t, and its third argument is a positive machine integer n. Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's default interpreter, which applies the rules in a fair, top-down fashion. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end. Again, `meta-rewrite` is a paradigmatic example of a descent function; its corresponding interpreter strategy is that of the `rewrite` command in Maude [9], that is,

$$\texttt{meta-rewrite}(\overline{\mathcal{R}},\overline{t},n) = \overline{I_{Maude}(\mathcal{R},t,\texttt{rewrite}\,[n])}.$$

The operation `meta-apply` has syntax:

```
op meta-apply :
      Module Term Qid Substitution MachineInt -> ResultPair
      [special ( ... )] .
```

The first four arguments are representations in META-LEVEL of a module $\mathcal{R}$, a term t in $\mathcal{R}$, a label l of some rules in $\mathcal{R}$, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n. meta-apply then returns a pair of sort ResultPair consisting of a term and a substitution. The syntax for substitutions and for results is

```
sorts Assignment Substitution ResultPair .
subsort Assignment < Substitution .

op _<-_ : Qid Term -> Assignment .
op none : -> Substitution .
op _;_ : Substitution Substitution -> Substitution
         [assoc comm id: none] .
op {_,_} : Term Substitution -> ResultPair .
```

The operation meta-apply is evaluated as follows:

1. the term t is first fully reduced using the equations in $\mathcal{R}$;
2. the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;
3. the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise {error*, none} is returned;
4. the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in $\mathcal{R}$;
5. the pair formed using the constructor {_,_} whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

The interpreter strategy associated to meta-apply$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ is not that of a user-level command in the Maude interpreter. It is instead a built-in strategy internal to the interpreter that attempts one rewrite at the top as explained above.

2.5.6 Parsing, Pretty Printing, and Sort Functions

Besides the descent functions already discussed, META-LEVEL provides several other functions that naturally belong to the universal theory and could have been equationally axiomatized in such a theory. However, for efficiency reasons they are provided as built-in functions. These functions allow parsing and pretty printing a term in a module at the metalevel, and performing efficiently a number of useful operations on the sorts declared in a module's signature.

The function meta-parse has syntax

```
op meta-parse : Module QidList -> Term [special ( ... )] .
```

It takes as arguments the representation of a module and the representation of a list of tokens as a list of quoted identifiers. It returns the meta-representation of the parsed term of that list of tokens for the signature of the module, which is assumed to be unambiguous. If such a parsed term does not exist, the error constant `error*` is returned instead. For example, given the module `NAT` presented in Section 2.5.4 and the input `'s '0 '+ '0`, we get

```
Maude> red meta-parse(NAT, ('s '0 '+ '0)) .
result Term: '_+_['s_['0'Zero],'0'Zero]
```

The function `meta-pretty-print` has syntax

```
op meta-pretty-print : Module Term -> QidList
      [special ( ... )] .
```

It takes as arguments the representation of a module M and the representation of a term t. It returns a list of quoted identifiers that encode the string of tokens produced by pretty printing t in the syntax given by M. In the event of an error an empty list is returned. Thus, given the module `NAT` presented in Section 2.5.4, we have, e.g.,

```
Maude> red meta-pretty-print(NAT,
           '_*_['_+_['s_['0'Zero], '0'Zero], 's_['0'Zero]]) .
result QidList: '`( 's '0 '+ '0 '`) '* 's '0
```

Pretty printing a term involves more than just naively using the mixfix syntax for operators. Precedence and gathering information (see Section 2.7.4) and the relative positions of underscores in an operator and its parent in the term must be considered to determine whether parentheses need to be inserted around any given subterm to avoid ambiguity. Also, if there is ad-hoc overloading in the module, additional checks must be done to determine if and where sort disambiguation syntax is needed.

The operations on sorts include `sameComponent`, `leastSort`, `lesserSorts`, `sortLeq`, and `glbSorts`. They provide commonly needed functions on the poset of sorts of a module in a built-in way at the metalevel. Their syntax is as follows

```
subsort Qid < Sort .

op leastSort : Module Term -> Sort [special ( ... )] .
op sortLeq : Module Sort Sort -> Bool [special ( ... )] .
op sameComponent : Module Sort Sort -> Bool [special ( ... )] .
op lesserSorts : Module Sort -> QidSet [special ( ... )] .
op glbSorts : Module Sort Qid -> QidSet [special ( ... )] .
```

At the metalevel, the sorts given by the user in his/her module are represented as quoted identifiers, that is, terms of sort `Qid`. However, the module `META-LEVEL` has also a sort `Sort` defined to be a supersort of `Qid`. Sorts not defined by the user, as for example the "error supersorts" added by the system to complete each connected component, are in this sort `Sort`. The syntax of an error supersort uses the set of maximal sorts of its connected component and is as follows

```
op errorSort : QidSet -> Sort .
```

The function `leastSort` takes as arguments the representations of a module and a term and computes the (representation of the) least sort of that term in the module. This function can return an error sort not defined by the user. For example, we can compute the least sort of the term $\overline{\texttt{N + s M}}$ in the previous module `NAT` as follows.

```
Maude> red least-sort(NAT, N + s M) .
result Qid: Nat
```

Given a module M with subsort relation $\leq_M$, and sorts $s, s' \in S$, where S is the set of sorts in M, the Boolean function `sortLeq(`$\overline{M}$`,`$\overline{s}$`,`$\overline{s'}$`)` is true if and only if $s \leq_M s'$. Note that the sorts passed to the function are of sort `Sort`.

```
Maude> red sortLeq(NAT, Nat, NzNat) .
result Bool: false
```

Given a module M with subsort relation $\leq_M$, and sorts $s, s' \in S$, where S is the set of sorts in M, the Boolean function `sameComponent(`$\overline{M}$`, `$\overline{s}$`,`$\overline{s'}$`)` is true if and only if s and s' belong to the same connected component in the subsort ordering $\leq_M$. Note that the sorts passed to the function are of sort `Sort`.

```
Maude> red sortLeq(NAT, Nat, Bool) .
result Bool: false
```

where it should be noted the sort `Bool`, although not explicitly present in the signature of `NAT`, is nevertheless present in its *extended signature*, as explained in Sections 2.1.1 and 2.7.2.

Given a module M and a sort s, the function `lesserSorts` takes their metalevel representations as arguments, and returns (the representation of the) set of sorts strictly smaller than s in M. For example:

```
Maude> red lesserSorts(NAT, Nat) .
result Qid: NzNat
```

Note that in this case it returns only one sort (of sort `Qid`), but in general it returns a set of sorts. Since no error sorts can appear in such a set, that is, the sorts are all in fact quoted identifiers, the function `lesserSorts` returns a `QidSet`.

Finally, the function `glbSorts` takes the representations of two sorts and a module as input and computes the representation of the set of maximal lower bounds of the two sorts[13]. Note the asymmetry in the declaration of this function, having as arguments, together with the module, a sort of sort `Sort` and another of sort `Qid`. This asymmetry might be eliminated in a future version of the system. As an example for the use of this function, let us see how to compute the greatest lower bound for sorts `Nat` and `NzNat` in the module `NAT` presented above.

```
Maude> red glbSorts(NAT, Nat, NzNat) .
result Qid: NzNat
```

As for `lesserSorts`, in general `glbSorts` returns a set of sorts.

2.6 Internal Strategies

System modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. We need to have good ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate *strategies*. In Maude, thanks to its reflective capabilities,

[13] Of course, when the set of maximal lower bounds of two sorts is a singleton $\{s\}$, then s will be the greatest lower bound of the two sorts, thus the notation `glbSorts`. In subsequent discussions, when we speak of the "greatest lower bound" we will always in fact mean the more general notion of the set of maximal lower bounds.

strategies can be made *internal* to the system. That is, they can be defined by rewrite rules in a normal module in Maude, and can be reasoned about as with rules in any other module.

In fact, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. The idea is to use the operations `meta-reduce`, `meta-apply`, and `meta-rewrite` as basic strategy expressions, and then to extend the module `META-LEVEL` by additional strategy expressions and corresponding semantic rules. Here we follow the methodology for defining and proving correct internal strategy languages for reflective logics introduced in [15, 10].

To illustrate this idea, let us reconsider the module `SORTING` for sorting vectors of integers introduced in Section 2.2. We will use this module as a running example to explain the way in which the application of rules can be controlled.

As mentioned before, strategy languages can be defined within Maude in user-definable extensions of the module `META-LEVEL`. As an example, we introduce the following module `STRATEGY`. We first introduce the basic syntax; the module's equations are then discussed and illustrated with examples in the rest of the section.

```
fmod STRATEGY is
  protecting META-LEVEL .
  sorts MetaVar Binding BindingList
        Strategy StrategyExpression .
  subsort MetaVar < Term .

  ops I J : -> MetaVar .
  op binding : MetaVar Term -> Binding .
  op nilBindingList : -> BindingList .
  op bindingList : Binding BindingList -> BindingList .

  op rewInWith :
        Module Term BindingList Strategy -> StrategyExpression .
  op set : MetaVar Term -> Strategy .
  op rewInWithAux :
        StrategyExpression Strategy -> StrategyExpression .
  op idle : -> Strategy .
  op failure : -> StrategyExpression .
  op and : Strategy Strategy -> Strategy .
  op apply : Qid -> Strategy .
  op applyWithSubst : Qid Substitution -> Strategy .
  op iterate : Strategy -> Strategy .
  op while : Term Strategy -> Strategy .
  op orelse : Strategy Strategy -> Strategy .

  op extTerm : ResultPair -> Term .
  op extSubst : ResultPair -> Substitution .
  op update : BindingList Binding -> BindingList .
  op applyBindingListSubst :
        Module Substitution BindingList -> Substitution .
  op substituteMetaVars : TermList BindingList -> TermList .
```

```
op SORTING : -> Module .

var M : Module .
vars V V' F G L : Qid .
vars T T' : Term .
var TL : TermList .
var SB : Substitution .
vars B B' : Binding .
vars BL BL' : BindingList .
var MV MV' : MetaVar .
vars ST ST' : Strategy .

eq SORTING
  = (mod 'SORTING is
       including 'MACHINE-INT .
       sorts 'Pair ; 'PairSet .
       subsort 'Pair < 'PairSet .
       op '<_;_> : 'MachineInt 'MachineInt -> 'Pair
           [none] .
       op 'empty : nil -> 'PairSet [none] .
       op '__ : 'PairSet 'PairSet -> 'PairSet
             [assoc comm id({'empty}'PairSet)] .
       var 'I : 'MachineInt .
       var 'J : 'MachineInt .
       var 'X : 'MachineInt .
       var 'Y : 'MachineInt .
       none
       none
       crl ['sort]: '__['<_;_>['J, 'X], '<_;_>['I, 'Y]]
          => '__['<_;_>['J, 'Y], '<_;_>['I, 'X]]
             if '_and_['_<_['J, 'I], '_>_['X, 'Y]]
                 = {'true}'Bool .
     endm) .
```

Before we explain some of the strategies that can be defined using the strategy language introduced in STRATEGY, note that the default strategy of the Maude interpreter for system modules can be easily (and efficiently) called using the built-in function meta-rewrite introduced in Section 2.5.5.

```
Maude> rew meta-rewrite(SORTING,
             '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                 '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                 '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
             0) .
result Term: '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
                 '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
                 '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]]
```

In this example, the third argument of meta-rewrite is 0. As explained in Section 2.5.5 this indicates that no bound on the number of rewrites is imposed. We can use this argument to see intermediate steps, or to stop at some point nonterminating rewrites. For example, we can see the resulting term after the application of two rules (twice the same rule in this case) as follows.

```
Maude> rew meta-rewrite(SORTING,
             '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                 '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                 '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
             2) .
result Term: '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
                 '<_;_>[{'2}'NzMachineInt,{'3}'NzMachineInt],
                 '<_;_>[{'3}'NzMachineInt,{'2}'NzMachineInt]]
```

In the module `STRATEGY` the function `rewInWith` computes strategy expressions. The first two arguments of `rewInWith` are the metarepresentations of a module T and a term t in `META-LEVEL`. The fourth argument is the strategy S we want to compute, and the third argument is used to store information that may be relevant for S. Our definition of `rewInWith` is such that, as the computation of a given strategy expression proceeds, t gets rewritten by controlled application of rules in T, the information stored in the third argument may be updated, and the strategy S is rewritten into the remaining strategy to be computed. In case of termination, this is the `idle` strategy and we are done. The strategy expression `failure` is returned when a requested strategy cannot be carried out.

A basic strategy we can express is the application of a rule once at the top of a term (if the top operator has attributes containing axioms such as associativity or associativity and commutativity, matching is done modulo those axioms) with the first possible match found when no constraints are placed on the matching substitution. For this basic strategy, we introduce in our signature the constructor `apply`, whose only argument is an identifier corresponding to the rule label to be applied, and we define the value of `rewInWith` for this strategy, using the built-in operation `meta-apply`, as follows:

```
eq rewInWith(M, T, BL, apply(L))
  = if meta-apply(M, T, L, none, 0)
          == {error*, none}
    then failure
    else rewInWith(M,
            extTerm(meta-apply(M, T, L, none, 0)),
            BL, idle)
    fi .
```

The operations `extTerm` and `extSubst` are selectors extracting the first and second component, respectively, from a pair constructed with {_,_}.

```
eq extSubst({T, SB}) = SB .
eq extTerm({T, SB}) = T .
```

We can see the computation of an `apply`-strategy expression with the following example:

```
Maude> rew rewInWith(SORTING,
              '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                  '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                  '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              nilBindingList,
              apply('sort)).
result StrategyExpression:
rewInWith(SORTING14,
```

```
           '__['<_;_>[{'1}'NzMachineInt,{'2}'NzMachineInt],
               '<_;_>[{'2}'NzMachineInt,{'3}'NzMachineInt],
               '<_;_>[{'3}'NzMachineInt,{'1}'NzMachineInt]],
           nilBindingList,
           idle)
```

The information relevant for the computation of a strategy expression is recorded as a list of bindings of values to metavariables, where the values are of sort `Term` (that is, they are representations of terms) and metavariables are introduced by the user as constants of sort `MetaVar`. The sort `MetaVar` is declared as a subsort of the sort `Term`, so that in any expression in which the representation of a term t can appear, a metavariable—to which the representation of t may be bound—can appear as well.

The computation of the strategy `set` updates the recorded information. This is done by the function `update`. Notice that the terms whose representations are bound to metavariables are kept in fully reduced form, using the built-in operation `meta-reduce`. The representation of the term set to a metavariable may itself contain metavariables, which must be substituted by the representations of the terms they are bound to in the list of bindings present before the updating. This is done by the function `substituteMetaVars`. Recall that the default operational semantics for functional modules, and therefore for the function `meta-reduce`, is eager (i.e., bottom up or call-by-value).

```
eq rewInWith(M, T, BL, set(MV, T'))
  = rewInWith(M, T,
       update(BL,
          binding(MV, meta-reduce(M,
                         substituteMetaVars(T', BL)))),
       idle) .

eq substituteMetaVars(T, nilBindingList)
  = T .
eq substituteMetaVars(MV, bindingList(binding(MV', T'), BL))
  = if MV == MV' then T'
    else substituteMetaVars(MV, BL) fi .
eq substituteMetaVars(F, BL)
  = F .
eq substituteMetaVars({F}S, BL) = {F}S .
eq substituteMetaVars(F[TL], BL)
  = F[substituteMetaVars(TL, BL)] .
eq substituteMetaVars((T, TL), BL)
  = (substituteMetaVars(T, BL), substituteMetaVars(TL, BL)).

eq update(bindingList(binding(MV, T), BL), binding(MV', T'))
  = if MV == MV'
    then bindingList(binding(MV, T'), BL)
    else bindingList(binding(MV, T),
                update(BL, binding(MV', T')))
    fi .
eq update(nilBindingList, B)
  = bindingList(B, nilBindingList) .
```

[14] Of course, the constant `SORTING` gets also rewritten (in this case, to the meta-representation of the module `SORTING`); however, to ease readability we have "hidden" this rewrite in all the examples of this Section.

We can see the computation of a `set`-strategy expression with the following example:

```
Maude> rew rewInWith(SORTING,
            '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
            nilBindingList,
            set(I, {'1}'MachineInt)).
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'MachineInt,{'3}'MachineInt],
              '<_;_>[{'2}'MachineInt,{'2}'MachineInt],
              '<_;_>[{'3}'MachineInt,{'1}'MachineInt]],
          bindingList(binding(I, {'1}'NzMachineInt),
                      nilBindingList),
          idle)
```

The computation of the strategy `applyWithSubst` applies a rule partially instantiated with a set of assignments once at the top of a term (if the top operator has attributes containing axioms such as associativity or associativity and commutativity, matching is done modulo those axioms) using the first successful match consistent with the given partial substitution. The representations of the terms assigned to variables may contain metavariables that must be substituted by the representations of the terms they are bound to in the current list of bindings. This is done by the function `applyBindingListSubst`.

```
eq rewInWith(M, T, BL, applyWithSubst(L, SB))
  = if meta-apply(M, T, L,
                  applyBindingListSubst(M, SB, BL), 0)
         == {error*, none}
    then failure
    else rewInWith(M, extTerm(meta-apply(M, T, L,
                  applyBindingListSubst(M, SB, BL), 0)),
                   BL, idle)
    fi .

eq applyBindingListSubst(M, none, BL)
  = none .
eq applyBindingListSubst(M, ((V <- T); SB), BL)
  = ((V <- meta-reduce(M, substituteMetaVars(T, BL)));
     applyBindingListSubst(M, SB, BL)) .
```

Many interesting strategies are defined as concatenations of more basic strategies, or iterations of a given strategy. Frequently, the strategies must consider possible branchings in their computations, or establish conditions for further computations. To represent these cases, we extend our basic strategy language with the constructors `and`, `orelse`, `iterate`, and `while`.

The equations for the strategies `and`, `orelse`, and `iterate` are defined as follows.

```
eq rewInWith(M, T, BL, and(ST, ST'))
  = if rewInWith(M, T, BL, ST) == failure
    then failure
```

```
    else rewInWithAux(rewInWith(M, T, BL, ST), ST')
    fi .

eq rewInWith(M, T, BL, orelse(ST, ST'))
  = if rewInWith(M, T, BL, ST) == failure
    then rewInWith(M, T, BL, ST')
    else rewInWith(M, T, BL, ST)
    fi .

eq rewInWith(M, T, BL, iterate(ST))
  = if rewInWith(M, T, BL, ST) == failure
    then rewInWith(M, T, BL, idle)
    else rewInWithAux(rewInWith(M, T, BL, ST), iterate(ST))
    fi .
```

where the function `rewInWithAux` is defined by the equation

```
eq rewInWithAux(rewInWith(M, T, BL, idle), ST)
  = rewInWith(M, T, BL, ST) .
```

which forces the computation of a sequence of strategies to proceed step-by-step, in the sense that a strategy will only be considered after the previous one has been fully computed. We can illustrate the computation of the above strategies with the following examples:

```
Maude> rew rewInWith(SORTING,
            '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              nilBindingList,
              and(set(I, {'3}'MachineInt),
                  applyWithSubst('sort, ('I <- I)))) .
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]],
          bindingList(binding(I, {'3}'NzMachineInt),
                      nilBindingList),
          idle)


Maude> rew rewInWith(SORTING,
            '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              bindingList(binding(J, {'2}'MachineInt),
                          nilBindingList),
      orelse(applyWithSubst('sort, ('J <- {'4}'MachineInt)),
             applyWithSubst('sort, ('J <- J)))).
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'3}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'1}'NzMachineInt],
```

```
               '<_;_>[{'3}'NzMachineInt,{'2}'NzMachineInt]],
           bindingList(binding(J, {'2}'MachineInt),
                       nilBindingList),
           idle)

Maude> rew rewInWith(SORTING,
            '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
               nilBindingList,
               iterate(apply('sort))).
result StrategyExpression:
rewInWith(SORTING,
           '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
               '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
               '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]],
           nilBindingList, idle)
```

Finally, the strategy `while` makes the computation of a given strategy conditional to the satisfaction of a condition. This condition should be the representation in `META-LEVEL` of a term of sort `Bool`. As usual, the condition may contain metavariables that must be substituted by the representations of terms they are bound to in the current list of bindings. Notice that the definition of the value of `rewInWith` for the constructor `while` makes the iterative computation of the strategy contained in the second argument of `while` depend on the satisfaction (at the metalevel) of the condition represented in the first argument of `while`.

```
eq rewInWith(M, T, BL, while(T', ST))
  = if meta-reduce(M, substituteMetaVars(T', BL))
       == {'true}'Bool
    then (if rewInWith(M, T, BL, ST) == failure
          then rewInWith(M, T, BL, idle)
          else rewInWithAux(rewInWith(M, T, BL, ST),
                            while(T', ST))
          fi)
    else rewInWith(M, T, BL, idle)
    fi .
```

Now we can extend our basic strategy language to define, as an example, the algorithm for insertion sorting. The strategy `insert`(n) below can be used to sort a vector of integers of length n. The main loop in insertion sorting looks at each element of the vector of integers from the second to the n-th, and inserts it in the appropriate place among its predecessors in the vector.

We introduce two new metavariables `X` and `Y`.

```
op insert : MachineInt -> Strategy .
ops X Y : -> MetaVar .
var N : MachineInt .
eq insert(N)
   = and(set(Y, {'2}'MachineInt),
     while('_<=_[Y, {index(', N)}'MachineInt],
           and(set(X, Y),
           and(while('_>_[X, {'1}'MachineInt],
```

```
                    and(applyWithSubst('sort,
                           (('I <- X);
                           ('J <- '_-_[X, {'1}'MachineInt]))),
                    set(X, '_-_[X, {'1}'MachineInt]))),
          set(Y, '_+_[Y, {'1}'MachineInt]))))) .
```

For example, we can use the strategy `insert` to sort a vector of integers of length 10:

```
Maude> rew rewInWith(SORTING,
             '__['<_;_>[{'1}'MachineInt, {'10}'MachineInt],
                 '<_;_>[{'2}'MachineInt, {'9}'MachineInt],
                 '<_;_>[{'3}'MachineInt, {'8}'MachineInt],
                 '<_;_>[{'4}'MachineInt, {'7}'MachineInt],
                 '<_;_>[{'5}'MachineInt, {'6}'MachineInt],
                 '<_;_>[{'6}'MachineInt, {'5}'MachineInt],
                 '<_;_>[{'7}'MachineInt, {'4}'MachineInt],
                 '<_;_>[{'8}'MachineInt, {'3}'MachineInt],
                 '<_;_>[{'9}'MachineInt, {'2}'MachineInt],
                 '<_;_>[{'10}'MachineInt, {'1}'MachineInt]],
             nilBindingList,
             insert(10)) .
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt],
              '<_;_>[{'4}'NzMachineInt,{'4}'NzMachineInt],
              '<_;_>[{'5}'NzMachineInt,{'5}'NzMachineInt],
              '<_;_>[{'6}'NzMachineInt,{'6}'NzMachineInt],
              '<_;_>[{'7}'NzMachineInt,{'7}'NzMachineInt],
              '<_;_>[{'8}'NzMachineInt,{'8}'NzMachineInt],
              '<_;_>[{'9}'NzMachineInt,{'9}'NzMachineInt],
              '<_;_>[{'10}'NzMachineInt,{'10}'NzMachineInt]],
          bindingList(binding(Y, {'11}'NzMachineInt),
          bindingList(binding(X, {'1}'NzMachineInt),
                      nilBindingList)),
       idle)
```

2.6.1 The Game of Nim

To illustrate the great flexibility we have in defining strategies to control the process of execution of rules, we discuss a second example, namely, a system module `NIM` specifying a version of the game of Nim. There are two players and two bags of pebbles: a "draw" bag to remove pebbles from, and a "limit" bag to limit the number of pebbles that can be removed. We represent each state of the game as a pair of bags, where the first one represents the draw bag and the second one the limit bag. The two players take turns making moves in the game. At each move a player draws a nonempty set of pebbles not exceeding those in the limit bag. The limit bag is then readjusted to contain the least number of pebbles in either the double of what the player just drew, or what was left in the draw bag. The game then continues with the two bags in this new state. This move is axiomatized by the rule `mv`. The player who empties the draw bag wins.

```
mod NIM is
  protecting MACHINE-INT .
  sorts Pebble Bag State .
  subsorts Pebble < Bag .

  op o : -> Pebble .
  op emptyBag : -> Bag .
  op __ : Bag Bag -> Bag [assoc id: emptyBag] .
  op <_;_> : Bag Bag -> State .
  op size : Bag -> MachineInt .
  op readjust : Bag Bag -> Bag .

  vars X Y Z : Bag .

  eq size((o  X))
     = size(X) + 1 .
  eq size(emptyBag)
     = 0 .

  eq readjust(X, Y)
     = if size((X X)) <= size(Y)
       then (X X)
       else Y
       fi .

  crl [mv] : < (X Y) ;  Z >
     => < Y ;  readjust(X, Y) >
        if size(X) <= size(Z) and (X =/= emptyBag) .
endm
```

The initial model described by this module is the transition system containing exactly all the possible game moves allowed by the game. But there are many bad moves that would allow the other player to win. A good player should avoid such bad moves by trying to have a *winning strategy*. With such a strategy, each move made by the player inexorably leads to success, no matter what moves the other player attempts.

As we have already said, there is great freedom for defining many different strategy languages inside Maude. Even if some users decide to adopt a particular strategy language because of its good features, such a language remains fully *extensible*, so that new features and new strategies can be defined on top of them.

We define a winning strategy for the Nim game in the following extension of the module `STRATEGY`.

```
fmod NIM-STRATEGY is
  protecting STRATEGY .

  op moveToWin : -> Strategy .
  op findWinMove : Term Term -> Term .
  op noWinMove : -> Term .
  op NIM : -> Module .

  var T X Y Z : Term .
```

```
var M : Module .

eq NIM
  = (mod 'NIM is
       including 'BOOL .
       including 'MACHINE-INT .
       sorts 'Pebble ; 'Bag ; 'State .
       subsort 'Pebble < 'Bag .
       op 'o : nil -> 'Pebble [none] .
       op 'emptyBag : nil -> 'Bag [none] .
       op '__ : ('Bag 'Bag) -> 'Bag
           [assoc id({'emptyBag}'Bag)] .
       op '<_;_> : ('Bag 'Bag) -> 'State [none] .
       op 'size : 'Bag -> 'MachineInt [none].
       op 'readjust : ('Bag 'Bag) -> 'Bag [none] .
       var 'X : 'Bag .
       var 'Y : 'Bag .
       var 'Z : 'Bag .
       none
       eq 'size['__[{'o}'Pebble, 'X]]
         = '_+_['size['X], {'1}'MachineInt] .
       eq 'size[{'emptyBag}'Bag] = {'0}'MachineInt .
       eq 'readjust['X, 'Y]
         = 'if_then_else_fi[
              '_<=_['size['__['X, 'X]], 'size['Y]],
              '__['X, 'X],
              'Y] .
       crl['mv]: '<_;_>['__['X, 'Y], 'Z]
         => '<_;_>['Y, 'readjust['X, 'Y]]
            if '_and_['_<=_['size['X], 'size['Z]],
                      '_=/=_['X, {'emptyBag}'Bag]]
               = {'true}'Bool .
     endm) .

eq rewInWin(M, T, nilBindingList, moveToWin)
  = if findWinMove(T, {'o}'Pebble) == noWinMove
    then failure
    else rewInWin(M, findWinMove(T, {'o}'Pebble),
            nilBindingList, idle)
    fi .

eq findWinMove('<_;_>[X, Y], Z)
  = if meta-reduce(NIM, '_>=_['size[Y], 'size[Z]])
         == {'true}'Bool
    then (if findWinMove(
                extTerm(
                  meta-apply(NIM, '<_;_>[X, Y],
                     'mv, ('X <- Z), 0)), {'o}'Pebble)
              == noWinMove
          then extTerm(
                 meta-apply(NIM, '<_;_>[X, Y],
                    'mv, ('X <- Z), 0))
```

```
            else findWinMove('<_;_>[X, Y], '__[Z, {'o}'Pebble])
            fi)
      else noWinMove
      fi .

endfm
```

Given a state `< X ; Y >` in the game, the strategy `moveToWin` finds a *winning move* `< X' ; Y' >` if there is one, in the sense that either `< X' ; Y' >` is equal to `< emptyBag ; emptyBag >` or `< X' ; Y' >` is a move that eventually will lead to success, no matter which moves the other players attempts, assuming that in the following moves the player that makes the *winning* move uses the same *winning* strategy.

The strategy `moveToWin` calls the function `findWinMove` with the representation of a bag with only one pebble as its second argument. This argument is used as a tentative move. If the tentative bag `Z` is valid (the number of pebbles in it is smaller than the number of pebbles in the limit bag) then we tentatively make that move; if it is the case that from the new state of the game there is no winning move for the other player, then we make that move; but, if there is a winning move for our opponent, then `findWinMove` is called again with the tentative number of pebbles to remove increased by one. If the size of the tentative bag is greater than the size of the limit bag, then there is no possible winning move and `failure` is returned.

For example, to make a winning move from a state with a draw bag with seven pebbles and a limit bag with three pebbles we use the following strategy expression:

```
Maude> rew rewInWith(NIM,
        '<_;_>['__[{'o}'Pebble, {'o}'Pebble, {'o}'Pebble,
                   {'o}'Pebble, {'o}'Pebble, {'o}'Pebble,
                   {'o}'Pebble],
              '__[{'o}'Pebble, {'o}'Pebble, {'o}'Pebble]],
             nilBindingList, moveToWin) .
result StrategyExpression:
rewInWith(NIM,
    '<_;_>['__[{'o}'Pebble,{'o}'Pebble,{'o}'Pebble,
              {'o}'Pebble,{'o}'Pebble],
           '__[{'o}'Pebble,{'o}'Pebble,{'o}'Pebble,
              {'o}'Pebble]], nilBindingList, idle)
```

There are, of course, states of the game from which no winning move can be made. In these cases, the strategy `moveToWin` will return `failure`. For example:

```
Maude> rew rewInWith(NIM,
        '<_;_>['__[{'o}'Pebble, {'o}'Pebble, {'o}'Pebble],
                   {'o}'Pebble], nilBindingList, moveToWin) .
result StrategyExpression: failure
```

2.6.2 A Meta-Interpreter

As yet another example of user-defined strategies in Maude, we specify in an extension of the module `STRATEGY` a meta-interpreter for modules that only contain rules that are Church-Rosser and terminating (no equations are declared and none of the operators have attributes). For the sake of simplicity, we assume that all the rules are labeled `any`.

```
fmod META-INTERPRETER is
  protecting STRATEGY .
  sorts Position .
  subsorts MachineInt < Position .

  op emptyPos : -> Position .
  op pos : Position Position -> Position [assoc] .
  op nullPos : -> Position .
  op getSubterm : Term Position -> Term .
  op getSubtermAux : TermList Position -> Term .
  op replace : Term Term Position -> Term .
  op replaceAux : TermList Term Position -> Term .
  op nextPosition : Term Position -> Position .
  op nextPositionUp : Term Position -> Position .

  var P : Position .
  var N : MachineInt .
  var F G X Y L S : Qid .
  var T T' T1 T1' : Term .
  var TL TL' : TermList .

  eq pos(emptyPos, P) = P .
```

We first define some auxiliary functions needed to find the positions in a term. Positions are represented at the metalevel as `pos`-lists of natural numbers, and `emptyPos` is the empty position. We denote by $\overline{p}$ the representation of a position p in the module `META-INTERPRETER`.

The function `getSubterm(`$\bar{t}$`, `$\overline{p}$`)` returns the term $\overline{t|_p}$, if p is a valid position in t; otherwise, it returns `error*`.

```
  eq getSubterm(F, N) = error* .
  eq getSubterm({F}S, N) = error* .
  eq getSubterm(F[TL], N) = getSubtermAux(TL, N) .

  eq getSubterm(F, pos(N, P)) = error* .
  eq getSubterm({F}S, pos(N, P)) = error* .
  eq getSubterm(F[TL], pos(N, P))
    = getSubterm(getSubtermAux(TL, N), P) .

  eq getSubtermAux((T, TL), N)
    = if N == 1 then T else getSubtermAux(TL, (N - 1)) fi .
  eq getSubtermAux(T, N)
    = if N == 1 then T else error* fi .
```

The function `nextPosition(`$\bar{t}$`, `$\overline{p}$`)` returns the next position in the tree defined by the term t, according to a top-down leftmost-innermost strategy. If all positions have already been considered, the function `nextPosition` returns `nullPos`.

```
  eq nextPosition(T, P)
    = if getSubterm(T, pos(P, 1)) == error*
      then nextPositionUp(T, P)
      else pos(P, 1)
      fi .
```

```
eq nextPositionUp(T, emptyPos) = nullPos .

eq nextPositionUp(T, N)
  = if getSubterm(T, (N + 1)) == error*
    then nullPos
    else (N + 1)
    fi .

eq nextPositionUp(T, pos(P, N))
  = if getSubterm(T, pos(P, (N + 1))) == error*
    then nextPositionUp(T, P)
    else pos(P, (N + 1))
    fi .
```

The function `replace`($\overline{t}$, $\overline{t'}$, $\overline{p}$) returns the term $\overline{t[t']_p}$.

```
eq replace(T, T', emptyPos) = T' .

eq replace(F, T', N) = error* .
eq replace({F}S, T', N) = error* .
eq replace(F[TL], T', N) = F[replaceAux(TL, T', N)] .

eq replace(F, T', pos(N, P)) = error* .
eq replace({F}S, T', pos(N, P)) = error* .
eq replace(F[TL], T', pos(N, P))
  = F[replaceAux(TL, T', pos(N, P))] .

eq replaceAux((T, TL), T', N)
  = if N == 1
    then (T', TL)
    else (T, replaceAux(TL, T', (N - 1)))
    fi .
eq replaceAux(T, T', N)
  = if N == 1 then T' else error* fi .

eq replaceAux((T, TL), T', pos(N, P))
  = if N == 1
    then (replace(T, T', P), TL)
    else (T, replaceAux(TL, T', pos((N - 1), P)))
    fi .
eq replaceAux(T, T', pos(N, P))
  = if N == 1
    then replace(T, T', P)
    else error*
    fi .
```

Finally, we introduce the strategy `metaInterpreter` that specifies the Maude interpreter for functional modules[15], that is, for any valid module M and any

[15]We talk about an interpreter for Maude functional modules in the sense of reducing the (meta-representation of) a term to its canonical form using the Church-Rosser and terminating equations just as the Maude interpreter would do it. Note, however, that, as already mentioned, the *functional* module—whose operators cannot have any attributes—is represented here as a *system* module in which the Church-Rosser equations are represented as *rules* labeled `any`.

term t in that module, `rewInWith(`$\overline{M}$`, `$\bar{t}$`, nilBindingList, metaInterpreter)` returns `rewInWith(`$\overline{M}$`, `$\overline{t'}$`, nilBindingList, idle)`, where t' is the canonical form of t with respect to M.

```
op metaInterpreter : -> Strategy .
op applyInPRedex : Position -> Strategy .

var M : Module .

eq rewInWith(M, T, nilBindingList, metaInterpreter)
  = rewInWith(M, T, nilBindingList,
       orelse(and(applyInPRedex(emptyPos),
                  metaInterpreter),
              idle)) .
```

The auxiliary strategy `applyInPRedex(`$\overline{p}$`)` specifies an interpreter that only applies once a rule to a term t at position p or at any position "after" p in t (traversing the tree defined by t with a top-down leftmost-innermost strategy).

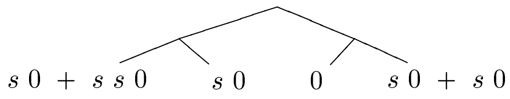
```
eq rewInWith(M, T, nilBindingList, applyInPRedex(P))
  = if P =/= nullPos
    then (if meta-apply(M, getSubterm(T, P), 'any, none, 0)
              == {error*, none}
          then rewInWith(M, T, nilBindingList,
                  applyInPRedex(nextPosition(T, P)))
          else rewInWith(M,
                  replace(T,
                     extTerm(
                        meta-apply(M, getSubterm(T, P),
                           'any, none, 0)),
                     P),
                  nilBindingList, idle)
          fi)
    else failure
    fi .
```

As an example, consider the following module `NAT-TREE`:

```
mod NAT-TREE is
  sorts Nat Tree .
  subsort Nat < Tree .

  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .

  rl [any]: (N + 0) =>  N .
  rl [any]: (0 + N) =>  N .
  rl [any]: (s N + s M) => s s (N + M) .

  op _^_ : Tree Tree -> Tree .
  op rev : Tree -> Tree .
```

```
    vars T T' : Tree .

    rl [any]: rev(N) => N .
    rl [any]: rev(T ^ T') => (rev(T') ^ rev(T)) .
  endm
```

Thus, in the module NAT-TREE, the tree



is represented by the term

```
    (((s 0 + s s 0) ^ s 0) ^ (0 ^ (s 0 + s 0))).
```

We extend the modulo META-INTERPRETER with the equation

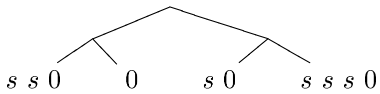
```
  op NAT-TREE : -> Module .
  eq NAT-TREE
    = (mod 'NAT-TREE is
         including 'BOOL .
         sorts ('Nat ; 'Tree) .
         subsort 'Nat < 'Tree .
         op '0 : nil -> 'Nat [none] .
         op 's_ : 'Nat -> 'Nat [none] .
         op '_+_ : ('Nat 'Nat) -> 'Nat [none] .
         op '_^_ : ('Tree 'Tree) -> 'Tree [none] .
         op 'rev : 'Tree -> 'Tree [none] .
         var 'N : 'Nat .
         var 'M : 'Nat .
         var 'T : 'Tree .
         var 'T' : 'Tree .
         none
         none
         rl ['any]: '_+_['N, {'0}'Nat] => 'N .
         rl ['any]: '_+_[{'0}'Nat, 'N] => 'N .
         rl ['any]: '_+_['s_['N], 's_['M]]
           => 's_['s_['_+_['N, 'M]]] .
         rl ['any]: 'rev['N] => 'N .
         rl ['any]: 'rev['_^_['T, 'T']]
           => '_^_['rev['T'], 'rev['T]] .
       endm) .
```

The result of computing the strategy metaInterpreter on the metarepresentation of the operation of reversing the above tree is the following

```
  Maude> rew rewInWith(NAT-TREE,
              'rev['_^_['_^_['_+_['s_[{'0}'Nat],
                                  's_['s_[{'0}'Nat]]],
                             's_[{'0}'Nat]],
                        '_^_[{'0}'Nat,
                             '_+_['s_[{'0}'Nat],
                                  's_[{'0}'Nat]]]]],
```

```
                 nilBindingList,
                 metaInterpreter) .
result StrategyExpression:
rewInWith(NAT-TREE,
   '_^_['_^_['s_['s_[{'0}'Nat]],{'0}'Nat],
        '_^_['s_[{'0}'Nat],'s_['s_['s_[{'0}'Nat]]]]],
   nilBindingList, idle)
```

Thus, the result is the meta-representation of the tree



that is, the meta-representation of the original tree reversed after all its leaves have been evaluated.

2.7 Parsing, Bubbles and Meta-Parsing

This section explains the parsing and meta-parsing functionalities of Maude. Section 2.7.1 presents a general overview of the design of the Maude Parser (MSCP). Section 2.7.2 explains how terms with user-definable mixfix syntax are parsed in a module and illustrates the basic functionalities of the parser with several examples. For convenience and expressiveness the signature of each module is extended with parentheses, Boolean connectives, some built-in polymorphic operators, sort test operators, and so on. Section 2.7.3 explains this extended signature of a module and how terms are parsed in it. Section 2.7.4 concentrates on the strategies for the specification of user-defined models of precedence/gathering patterns. Since a module's user-defined syntax can specify a general context-free grammar that can be ambiguous, parentheses may in general be needed to resolve such ambiguities. By means of the definition of precedence/gathering patterns, the user can control the precedence and the syntactic order of evaluation of operators to remove such ambiguities without recourse to unnecessary parentheses, while keeping the same syntax. Section 2.7.5 describes the rules used by Maude to assign default precedence values and gathering patterns. Finally, Section 2.7.6 explains what we might call *linguistic reflection*, that is, the possibility of parsing a term from which we then extract a grammar to parse some unanalyzed portions of that term—for example, parsing the top-level syntax of a module in a language allowing user-definable syntax, to obtain the grammar in which to parse expressions in that module—, is supported by means of "bubbles" and the metaparsing facility of META-LEVEL.

2.7.1 MSCP Parser Design: An Overview

From a computational point of view, the semantic and logical framework provided by rewriting logic has to be complemented with a *reflective syntactic framework*. Syntactic, or linguistic, reflection allows the *effective* specification, implementation, and semantic definition of a very wide range of logics and languages—including languages such as Core Maude and Full Maude, whose modules can have user-definable syntax—for which rewriting logic acts as a metalanguage.

The intrinsic characteristics of Maude—mainly, its metalanguage functionality, its reflective nature, and its logical and semantic framework applications—pose very strong requirements on the design of a parsing algorithm for the language, since it has to fulfill the following constraints [52]:

- Interpreted parsing: languages are user-definable.
- Full Context-Free Grammars (CFG), and not only LALR models.
- With precedence/gathering patterns that modify the grammatical power of nonterminal symbols.
- Grammars are extended to incorporate *bubbles*. Bubbles are the key notion to implement syntactic reflection. Furthermore, bubble sorts are user-definable.
- Techniques for error detection and error recovery must be supported.
- Efficiency is a main goal, as the parser is the surface of the rest of the system, especially in `META-LEVEL` computations.

The logical kernel of the current version of the parser is based on the SCP parsing algorithm [54, 53]. SCP is a bidirectional, bottom-up and event-driven parser for unrestricted context-free grammars. From an algorithmic point of view, we have proved the soundness and completeness of SCP. From a computational perspective, SCP avoids overparsing [51], allowing an elegant and very efficient manipulation of a wide set of CFGs. The use of multi-virtual trees [50] at the level of representation and the relations of coverage, partial derivability and adjacency as top-down predictions over the basic bottom-up strategy, obtain a high level of efficiency without diminishing the generality of the algorithm.

The logically proved soundness and completeness of SCP guarantees that the Maude version of SCP (MSCP) will generate all the possible grammatical analyses for each term in a given signature. This avoids some completeness problems detected in the OBJ3 parser.

MSCP is able to analyze *β-extended CFGs* (CFGs extended with bubbles and precedence/gathering patterns) [52]. The MSCP parsing algorithm incorporates very sophisticated error detection and error recovery mechanisms based on the notions of partial derivability and adjacency, originally developed in SCP.

Finally, the overall architecture of the MSCP algorithm allows an efficient treatment of syntactic reflection. Besides, this reflective power of the parser supports the parsing and meta-parsing functionality of the `META-LEVEL` module (Section 2.5) as well as a flexible and natural syntax definition model (see Section 2.7.6).

A detailed description of the SCP Parsing Algorithm may be found in [53]. The notion of overparsing is described in [51], while other internal strategies of SCP such as the use of multi-virtual trees and the formal kernel of the algorithm may be found in [50]. The techniques used in the computational layer of SCP, responsible of the efficiency of the algorithm, are described in [54]. Finally, the technical report [52] describes the Maude version (MSCP) of the parser.

2.7.2 Mixfix Parsing of Terms in a Module

We can illustrate the notion of term and the idea of parsing terms in the signature of a module by means of the following `BINARY-NAT` module supporting natural number arithmetic in binary notation. The module includes the usual arithmetic operators `_+_`, `_*_`, and `_^_` of sum, product, and exponentiation on natural numbers in binary notation plus:

- Constants `0` and `1` as constructors of the sort `Bit`.
- The operator `__` to represent elements of the sort `Bits` as sequences of `0`'s and `1`'s.
- The operator `|_|`, to obtain the length of a binary number.
- The operator `normalize`, to compress the representation of a binary number by suppressing the 0's on the left of a number, if any.
- The "greater-than" Boolean predicate `_>_`.
- The `not_` operator, that performs the logical negation of a string of bits.

```
fmod BINARY-NAT is
  protecting MACHINE-INT .
  sorts Bit Bits .
  subsort Bit < Bits .

  ops 0 1 : -> Bit .

  op __ : Bits Bits -> Bits [assoc] .
  op |_| : Bits -> MachineInt .
  op not_ : Bits -> Bits .
  op normalize : Bits -> Bits .
  ops _+_ _*_ : Bits Bits -> Bits [assoc comm] .
  op _^_ : Bits Bits -> Bits .
  op _>_ : Bits Bits -> Bool .
  op _?_:_ : Bool Bits Bits -> Bits .

  vars S T : Bits .
  vars B C : Bit .
  var L : Bool .

  *** Length
  eq | B | = 1 .
  eq | S B | = | S | + 1 .

  *** Not
  eq not (S T) = (not S) (not T) .
  eq not 0 = 1 .
  eq not 1 = 0 .

  *** Normalize suppresses zeros at the left of a binary number
  eq normalize(0 S) = normalize(S) .
  eq normalize(1 S) = 1 S .

  *** Greater than
  eq 0 > S = false .
  eq 1 > (0).Bit = true .
  eq 1 > (1).Bit = false .
```

```
  eq B > (0 S) = B > S .
  eq B > (1 S) = false .
  eq (1 S) > B = true .
  eq (B S) > (C T)
    = if | normalize(B S) | > | normalize(C T) |
      then true
      else if | normalize(B S) | < | normalize(C T) |
           then false
           else (S > T)
           fi
      fi .

  *** Binary addition
  eq 0 + S = S .
  eq 1 + 1 = 1 0 .
  eq 1 + (T 0) = T 1 .
  eq 1 + (T 1) = (T + 1) 0 .
  eq (S B) + (T 0) = (S + T) B .
  eq (S 1) + (T 1) = (S + T + 1) 0 .

  *** Binary multiplication
  eq 0 * T = 0 .
  eq 1 * T = T .
  eq (S B) * T = ((S * T) 0) + (B * T) .

  *** Binary exponentiation
  eq T ^ 0 = 1 .
  eq T ^ 1 = T .
  eq T ^ (S B) = (T ^ S) * (T ^ S) * (T ^ B) .

  *** Mixfix ?: operator
  eq L ? S : T = if L then S else T fi .
endfm
```

Note the use of the sort `Bool`. This sort is not a proper sort of the signature of `BINARY-NAT`. However, `Bool`, together with other information about polymorphic operators, parentheses, subsort-overloaded operators, and so on, belongs to the extended signature of a module, which Maude generates automatically for each module (see Section 2.7.3).

This module illustrates several important aspects of the grammatical power of Maude.

- *Empty Syntax*: With the following declarations we can write natural numbers in binary notation such as 1 0 0 1 or 1 1.

  ```
  sorts Bit Bits .
  subsort Bit < Bits .

  ops 0 1 : -> Bit .

  op nil : -> Bits .
  op __ : Bits Bits -> Bits [assoc id: nil] .
  ```

- *Outfix Syntax*: The length operator `|_|` is an example of outfix syntax specification for operators.

```
Maude> red | 1 0 1 1 0 | .
result NzMachineInt: 5
```

- *Prefix and Postfix Syntax*: `BINARY-NAT` includes the `not_` operator, defined with prefix syntax.

```
Maude> red 0 (not 1) 0 .
result Bits: 0 0 0

Maude> red not (1 0 1) .
result Bits: 0 1 0
```

- *Infix Syntax*: The operators `_+_`, `_*_` and `_>_` illustrate the model for the specification of infix operators in Maude.

```
Maude> red (1 0 0) + (1 1 1) .
result Bits: 1 0 1 1

Maude> red (1 1 1) * (1 1 0) .
result Bits: 1 0 1 0 1 0

Maude> red (0 1 0 1) > (1 1) .
result Bool: true
```

- *Mixfix Syntax*: The operator `_?_:_` is an example of mixfix notation in Maude. In fact, this operator combines both outfix and infix notation.

```
Maude> red ((1 0) > (0 1 1)) ? ((1 0) * 1) : ((1 0) + 1) .
result Bits: 1 1
```

The previous discussion on the user-definable notational power of Maude is a good basis for discussing the process of parsing terms in the context of a signature. This process is divided in two phases. In a first step, Maude collects all the information pertinent to the parsing problem included in a module: set of sorts, subsort relations, operators (paying special attention to the notational pattern of each operator) and variables. All this information is translated into a context-free grammar. In a second step, each time Maude detects a term in the corresponding signature, the MSCP algorithm is used to obtain the grammatical structure of the term according to the context-free grammar previously obtained from the module.

2.7.3 Parsing Terms in the Extended Signature of a Module

In `BINARY-NAT` it is possible to reduce the following terms.

```
Maude> red ((1 0) + (1 0)) * (1 1) .
result Bits: 1 1 0 0
```

```
Maude> red ((1 0) > 1) and ((1 1) > (1 0)) .
result Bool: true

Maude> red (1).Bit * 0 .
result Bit: 0

Maude> red (1 0) + (1 0) + (1 0) .
result Bits: 1 1 0
```

But, parentheses, logical operators such as `and`, sort tests, and qualification operators are not a proper part of the module. These structures belong to the so-called *extended signature of a module*. That is, the process of grammar generation from the user-defined signature adds automatically more information than that strictly contained in the signature. From the user's viewpoint, the main structures added in the extended signature of a module are:

- *Sort Disambiguation*: For each sort `S` in the signature of the module, Maude generates the operator

  ```
  op (_).S : S -> S .
  ```

  This helps in the disambiguation of ad-hoc overloaded constants and terms. For example, in the module `BINARY-NAT`, because of the presence of machine integers, the constants `0` and `1` and the operator `_+_` are ad hoc overloaded (see Section 2.1.1). Thus, terms such as `0`, `1` or `1 + 1` are ambiguous. We can eliminate this ambiguity by using sort disambiguation operators, that is, by qualifying the ambiguous terms with their sorts.

  ```
  Maude> red (1).Bit .
  result Bit: 1

  Maude> red (1 + 1).Bits .
  result Bits: 1 0

  Maude> red 1 + (1).MachineInt .
  result NzMachineInt: 2
  ```

- *Parentheses*: For each sort `S` in the signature of a module, the extended signature of that module contains the following operator.

  ```
  op (_) : S -> S .
  ```

  These operators allow the use of parentheses without having to declare a parentheses operator for each sort.

  ```
  Maude> red not (1 0 1 1) .
  result Bits: 0 1 0 0
  ```

- *Prefix Form of Mixfix Operators* or *Simple Identifier Form*: Each operator declared in mixfix form, may also be used in its single identifier prefix form. For example:

```
Maude> red _>_(1 0, 1) .
result Bool: true

Maude> red _?_:_(1 > 1 0, 1, 0) .
result Bit: 0
```

- *Flattened Associative Argument Lists*: Operators with the attribute `assoc` may be used in Maude in a nonparenthesized form.

  ```
  Maude> red (1 1) + (1 1) + (1 1) + (1 1) .
  result Bits: 1 1 0 0
  ```

  Furthermore, if the associative operator is given in prefix notation, it can take not only two, but arbitrarily many more arguments.

  ```
  Maude> red _+_(1 1, 1 1, 1 1, 1 1) .
  result Bits: 1 1 0 0
  ```

- *Polymorphic Operators and the* `BOOL` *Module*: All the information contained in the predefined modules `TRUTH-VALUE`, `TRUTH` and `BOOL` (see Section 2.4) is included in the extended signature of each module. In particular, appropriate instances of the polymorphic operators contained in `TRUTH` (that is, `if_then_else_fi`, `_==_` and `_=/=_`) are generated for each sort in the module. In addition, for each sort `S` sort predicates `_: S` and `_:: S` are also added.

  Note that this extension of the signature has allowed the specification of operators, variables, and equations such as the following ones in `BINARY-NAT`.

  ```
  op _?_:_ : Bool Bits Bits -> Bits .

  var L : Bool .

  eq B S > C T
    = if | normalize(B S) | > | normalize(C T) |
      then true
      else if | normalize(B S) | < | normalize(C T) |
           then false
           else (S > T)
           fi
      fi .
  ```

- The extended signature includes also the error supersorts, and the overloaded lifting to those supersorts of all operators to support error terms.

2.7.4 Precedence and Gathering

`BINARY-NAT` contains a rich set of notational models for the specification of operators, and illustrates clearly the idea of an extended signature for a module. Nevertheless, the operators in this module will generate, in most cases, complex and subtle inconsistencies, ambiguities and unintuitive results, as shown in the following examples. The concepts of precedence and gathering provide a flexible way of avoiding these ambiguities without having to write unnecessary parentheses.

- Let us consider the following reduction.

```
Maude> red 1 0 + 1 0 .
result Bits: 1 1 0
```

  The expected result is `1 0 0`. The reason for getting the unexpected result is that Maude is really processing the term `1 (0 + 1) 0`, which generates `1 1 0`.

- The following example shows a problematic interaction between the operators `not_` and `__`. Intuitively, we expect that the result of `not 0 1 0` will be `1 0 1`. But Maude parses the term `not 0 1 0` as `(not 0) 1 0`. Therefore, the result is `1 1 0`.

```
Maude> red not 0 1 0 .
result Bits: 1 1 0
```

- One might expect that the two terms reduced in the following example, namely, `(1 0) + (1 0) * (1 0)` and `(1 0) * (1 0) + (1 0)` yield the same result (`1 1 0`). However, the second term is parsed in a form that applies the operators in an unexpected way, yielding a possibly confusing result.

```
Maude> red (1 0) + (1 0) * (1 0) .
result Bits: 1 1 0

Maude> red (1 0) * (1 0) + (1 0) .
result Bits: 1 0 0 0
```

- The term `1 1 > 1 ? 1 : 0` seems unambiguous. Nevertheless, following strictly the signature of `BINARY-NAT`, there are two possible parses:
  - `((1 1) > 1) ? 1 : 0`, with result `1`, and
  - `1 ((1 > 1) ? 1 : 0)`, with result `1 0`.

  Maude detects this ambiguity, and selects randomly one of the parses, which may lead to unexpected results in more complex terms.

```
Maude> red 1 1 > 1 ? 1 : 0 .
WARNING: <standard input>, line 894:
   Ambiguous term, two parses are:
(1 1) > 1 ? 1 : 0
-versus-
1 (1 > 1) ? 1 : 0

Arbitrarily taking the first as correct.
result Bit: 1
```

  At this point we may say a few words about the treatment of ambiguities in Maude. The MSCP parser obtains all parses of a term. However, since the number of ambiguous parses can sometimes be quite large, Maude presents only two of them to show the ambiguity, and lets the user solve the problem. For example, the following expression in fact has three different parses, but only two are given.

```
Maude> red 1 1 1 > 1 ? 1 : 0 .
WARNING: <standard input>, line 895:
   Ambiguous term, two parses are:
(1 1 1) > 1 ? 1 : 0
-versus-
1 ((1 1) > 1) ? 1 : 0

Arbitrarily taking the first as correct.
result Bit: 1
```

The third parse is `1 1 (1 > 1) ? 1 : 0`.

- The following examples illustrate problems which appear as a consequence of an undefined model of precedence between operators. In this case, Maude applies automatically the algorithm for default precedence values and gathering patterns assignment according to the rules presented in Section 2.7.5. But there is an additional source of ambiguity, in this case related to the notion of *syntactic order of evaluation* of operators.

  The precedence of infix operators determines the order in which the operators are to be applied. For example, given two operators δ_1 and δ_2, if δ_1 takes precedence over δ_2, this means that an expression like $E_1 \delta_2 E_2 \delta_1 E_3$ will be evaluated as $E_1 \delta_2 (E_2 \delta_1 E_3)$. The operator `_*_` takes precedence over (has a higher precedence than) the operator `_+_`, and this is the reason why we expect that `(1 0) + (1 0) * (1 0)` should be parsed as `(1 0) + ((1 0) * (1 0))`.

  Let us now consider expressions with the *same* operator appearing several consecutive times. For associative operators, this is not a problem, as it happens in:

```
Maude> red (1 1) + (1 1) + (1 1) + (1 1) .
result Bits: 1 1 0 0
```

  But for nonassociative operators, the so-called *syntactic order of evaluation*[16], that is, the order in which the operator should be associated in several contiguous occurrences, may modify the result. In other words, expressions with nonassociative operators appearing consecutively are in fact grammatically ambiguous:

```
Maude> red (1 1) ^ (1 1) ^ (1 1) .
WARNING: <standard input>, line 896:
   Ambiguous term, two parses are:
(1 1) ^ ((1 1) ^ 1 1)
-versus-
((1 1) ^ 1 1) ^ 1 1

Arbitrarily taking the first as correct.
result Bits: 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 0
             0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1
```

[16]The syntactic order of evaluation is a *syntactic* notion, having to do with how parentheses are associated. It is different from the *semantic* notion of order of evaluation of the arguments of an operator specified by a strategy explained in Section 2.1.3.

We can distinguish two types of operators from the point of view of their syntactic order of evaluation:

- *Left Associative*: operators grouped and analyzed from left to right. This is the case of the arithmetic operators `_+_` and `_*_`.
- *Right Associative*: operators grouped and analyzed from right to left. For example, we shall see how we can make the exponentiation operator `_^_` right associative.

Although all the different parenthesized associations of an associative operator are guaranteed to yield the same result, nevertheless, as far as their syntactic order of evaluation is concerned such operators are grammatically ambiguous. In the extended signature of a module Maude incorporates rules to solve this ambiguity (Section 2.7.3). For nonassociative infix operators this source of ambiguity is solved by defining an order of evaluation.

These syntactic problems (grammatical scope, precedence and syntactic order of evaluation of operators) are solved in OBJ3 [27] and in Maude by means of precedence and gathering patterns.

In Maude, each operator has associated to it a precedence value and a gathering pattern. They can be specified by the user by means of the `precedence` (abbreviated `prec`) and `gather` attributes. If not specified, Maude assigns default values as explained in Section 2.7.5.

A *precedence* value is an integer greater than or equal to 0, which may be understood as an output value associated to terms having the corresponding operator as their top symbol.

On the other hand, *gathering* patterns are associated to the *arguments* of an operator. Gathering patterns are given as nonempty sequences of the following possible pattern:

- `E`: The argument must have a precedence value equal to or lower than the precedence value of the operator.
- `e`: The argument must have a precedence value strictly lower that the precedence value of the operator.
- `&`: The operator allows any precedence value for the corresponding argument.

We can illustrate the notions of precedence and gathering by considering a variant `BINARY-NAT-PREC` of the module `BINARY-NAT` whose only difference is that we have now specified precedence values and gathering patterns for the operators as follows.

```
op __ : Bits Bits -> Bits [assoc prec 1 gather (e E)] .
op |_| : Bits -> MachineInt .                        *** Length
op not_ : Bits -> Bits [prec 2 gather (E)] .
op normalize : Bits -> Bits .
op _+_  : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
op _*_  : Bits Bits -> Bits [assoc comm prec 4 gather (E e)] .
op _^_  : Bits Bits -> Bits [prec 3 gather (e E)] .
op _>_  : Bits Bits -> Bool [prec 6 gather (E E)] .
op _?_:_ : Bool Bits Bits -> Bits [prec 7 gather (& E E)] .
```

Constants have precedence 0. In our example, this rule is applied to `0`, `1` and `nil`. The precedence value of an operator is associated to the terms generated by this operator. In our example, the term `1` has precedence 0, the term `1 0 1` has precedence 1, the term `1 0 + 1` has precedence 5, and `1 0 * 1` has precedence 4.

To illustrate the behavior of the gathering patterns, let us focus on the declaration of the `_*_` operator. From it we can infer the following consequences:

- Every term with this operator as top symbol will have precedence 4 (by the attribute `prec 4`).
- The first argument of a binary multiplication (an `E` in the gathering pattern) must be a term with a precedence value smaller than or equal to the precedence of the operator, that is, the first argument of a binary multiplication must be a term with precedence 4 or less, for example, a constant (precedence 0), an exponentiation (precedence 3) or another multiplication (precedence 4), but *not* an addition (precedence 5) unless it is enclosed in parentheses.
- The second argument of a binary multiplication (an `e` in the gathering pattern) must have a precedence strictly lower than 4. That is, the precedence of the second argument of a binary multiplication may range from 0 to 3. This means that an expression where its top operator is a multiplication (precedence 4) cannot be the second argument of another expression whose top symbol is the multiplication operator, unless it is enclosed in parentheses. We elegantly solve in this way the problem of order of evaluation of this operator.

The simultaneous use of precedence and gathering attributes allows specifying any kind of precedence relations between operators and different types of syntactic order of evaluation.

To show in practice how this strategy works, we analyze some of the problems detected in `BINARY-NAT` using the signature of `BINARY-NAT-PREC`.

The first problem appeared with the term `1 0 + 1 0`. Without any information about precedence values and gathering patterns, we can think of two different analyses for this term: `(1 0) + (1 0)` and `1 (0 + 1) 0`. This ambiguity can be solved by the use of different precedence values for the operators involved, namely, `__` and `_+_`. In the module `BINARY-NAT-PREC` these operators have been defined as follows:

```
op __ : Bits Bits -> Bits [assoc id: nil prec 1 gather (e E)] .
op _+_ : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
```

Let us consider the two previous analyses proposed for the term `1 0 + 1 0`:

- *First analysis*: `(1 0) + (1 0)`. The top symbol of this term is `_+_`. The first argument of the addition is `(1 0)`. According to the definition of the operator `__`, the term `(1 0)` will have precedence 1. The second argument of the addition is the same and will have also the same precedence: 1. Now the gathering pattern of `_+_` constrains the precedence of the first argument `E` to a value 5 or smaller (the precedence of the operator `_+_`), and the second argument `e` must have a precedence value in the range 0 to 4. Since these conditions are fulfilled by the arguments of the addition, this analysis is taken as correct by the parser. Figure 2.2 shows

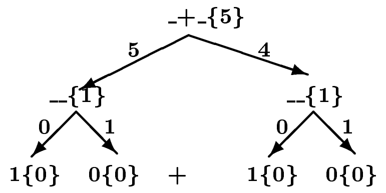


Figure 2.2: Correct parse tree of `1 0 + 1 0`.

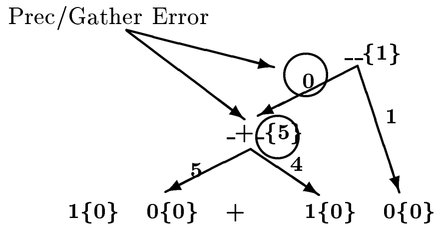


Figure 2.3: Incorrect parse tree of `1 0 + 1 0`.

graphically the situation just described. In the figure, each constant and each function symbol of a term has its corresponding precedence value in braces, and the maximum gathering value of each argument is associated to the corresponding arrow.

- *Second analysis*: `1 (0 + 1) 0`. This term is a sequence of bits, that is, the main or top operator of this term is `__`. The first argument of this operator is the term `1`, while the second argument is the subterm `(0 + 1) 0`, whose first argument is the addition `(0 + 1)` and whose second argument is the constant `0`. Since the precedence value of the addition is 5, and the first argument of the operator `__` must have a precedence value less than 1, the expression `1 (0 + 1) 0` is *not* a valid parse in `BINARY-NAT-PREC`. Graphically, the situation is shown in Figure 2.3: the parse error appears because the precedence value of an argument is higher than allowed by the gathering pattern of the corresponding argument in the operator declaration.

Of course, using parentheses is always a way of resolving ambiguities. In fact, it is worth noting that the parentheses operator automatically included in the extended signature of a module has precedence 0. Therefore, the term `1 (0 + 1) 0` is grammatically correct in `BINARY-NAT-PREC`.

```
Maude> red 1 0 + 1 0 .
result Bits: 1 0 0

Maude> red 1 (0 + 1) 0 .
result Bits: 1 1 0
```

Another subtle problem in the `BINARY-NAT` module was the different behaviors of the terms `(1 0) + (1 0) * (1 0)` and `(1 0) * (1 0) + (1 0)`. In that module, both operators `_+_` and `_*_` had no precedence attribute explicitly given. As both had the `assoc` attribute, the gathering patterns generated by default for both of them was `(e E)`, so the operators were associated from right to left.

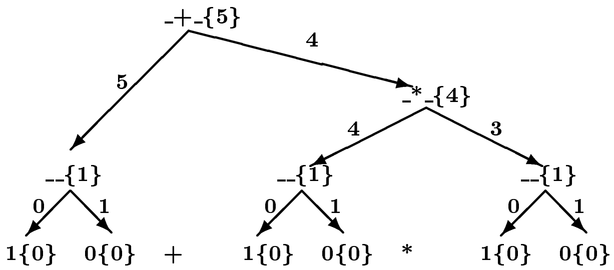


Figure 2.4: Correct interpretation of `1 0 + 1 0 * 1 0`.

In `BINARY-NAT-PREC` these two operators have different precedence values, and the gathering patterns have also been specified:

```
op _+_ : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
op _*_ : Bits Bits -> Bits [assoc comm prec 4 gather (E e)] .
op _^_  : Bits Bits -> Bits [prec 3 gather (e E)] .
```

The main consequence of this definition is that, as already explained, an addition cannot be an argument of a multiplication unless it is enclosed in parentheses. Thus, we have the following reductions.

```
Maude> red (1 0) + (1 0) * (1 0) .
result Bits: 1 1 0

Maude> red (1 0) * (1 0) + (1 0) .
result Bits: 1 1 0
```

The following examples illustrate how the precedence and gathering model of the operators of `BINARY-NAT-PREC` solves the problems of precedence and order of evaluation of operators.

- *Precedence*: Let us consider the term `1 0 + 1 0 * 1 0`, in which the two operators `_+_` and `_*_` are involved.

  ```
  op _+_  : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
  op _*_  : Bits Bits -> Bits [assoc comm prec 4 gather (E e)] .
  ```

  Taking into account the precedences of these operators, the expression will be evaluated by Maude as an addition of a number and a multiplication.

  ```
  Maude> red 1 0 + 1 0 * 1 0 .
  result Bits: 1 1 0
  ```

  Figure 2.4 shows the correct interpretation of this term in the signature of `BINARY-NAT-PREC`, that is, `1 0 + (1 0 * 1 0)`, while Figure 2.5 represents the incorrect interpretation `(1 0 + 1 0) * 1 0` and the precedence/gathering error that avoids such an interpretation.

- *Right Associativity*: We will illustrate the definition of a right associative operator by means of the exponentiation operator in the module `BINARY-NAT-PREC`. Left associativity is entirely analogous.

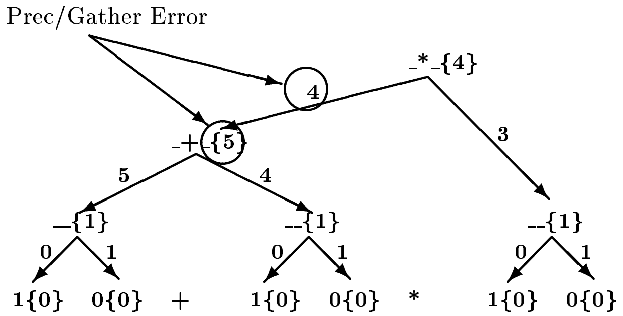


Figure 2.5: Incorrect interpretation of `1 0 + 1 0 * 1 0`.

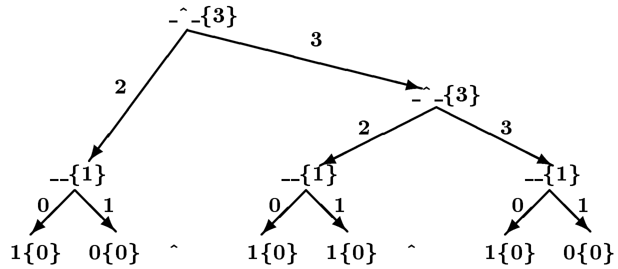


Figure 2.6: Correct interpretation of `1 0 ^ 1 1 ^ 1 0`.

```
op _^_  : Bits Bits -> Bits [prec 3 gather (e E)] .
```

Without parentheses, which modify the precedence and order of evaluation of the operators, the term `1 0 ^ 1 1 ^ 1 0` is evaluated using the right to left gathering pattern of the operator `_^_`. Figures 2.6 and 2.7 draw graphically the correct and incorrect interpretation of this term.

```
Maude> red 1 0 ^ 1 1 ^ 1 0 .
result Bits: 1 0 0 0 0 0 0 0 0 0 0

Maude> red (1 0 ^ 1 1) ^ 1 0 .
result Bits: 1 0 0 0 0 0 0
```

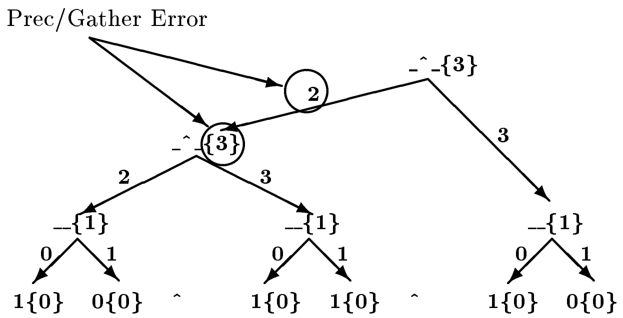


Figure 2.7: Incorrect interpretation of `1 0 ^ 1 1 ^ 1 0`.

2.7.5 Default Precedence and Gathering

This section sketches the rules used by Maude to generate the default precedence values and gathering patterns for operators; they are entirely similar to those used by OBJ3 [27]. These values will be associated to those operators for which the user does not specify this information as part of the operator declaration.

The rules for the assignment of default precedence values are:

- *Prefix operators* always have precedence 0, regardless of user settings. This rule is applied, for example, to the operator `normalize` in the module `BINARY-NAT`.
- *Outfix operators* have precedence 0. This is the case, for example, for the operator `|_|`.
- *Unary mixfix operators* have precedence 15. In the module `BINARY-NAT` this rule is applied to the operator `not_`.
- *Multi-ary mixfix operators* (with arity greater than 1) have precedence 41. In `BINARY-NAT` this rule is applied to the operators `__`, `_+_`, `_*_`, `_^_`, `_>_`, and `_?_:_`.

The rules for the generation of the default gathering patterns are:

- All arguments of prefix operators have a gathering pattern `&`, regardless of the user specification.
- For operators with *mixfix notation*, the gathering patterns are by default `&` for each argument. There are three exceptions to this rule (with exceptions 2 and 3 mutually exclusive, and with either of them overruling exception 1):
  - *Exception 1*: The gathering pattern of an argument will be `E` if the argument position (the corresponding underscore '`_`' in the operator name) is:
    1. the leftmost token in the operator name, or
    2. the rightmost token in the operator name, or
    3. it is adjacent to another underscore in the operator name.

    In `BINARY-NAT`, this exception changes to `E` the gathering pattern of the unique argument of the operator `not_`, and of the first and last arguments of the operators `_?_:_`, `_-_`, `__`, `_^_`, `_<_` and `_<=_`.
  - *Exception 2*: An operator will have gathering pattern `(e E)` if:
    1. starts with an underscore '`_`', and
    2. ends with an underscore '`_`', and
    3. has precedence (if supplied by the user) greater than 0, and
    4. has the `assoc` attribute.

    These conditions are fulfilled by the operators `_+_`, `_*_` and `__` of the `BINARY-NAT` module.
  - *Exception 3*: If an operator
    1. starts with an underscore '`_`', and
    2. ends with an underscore '`_`', and
    3. has precedence (if supplied by the user) greater than 0, and

iv. has the first and last arguments and its coarity in the same sort connected component, and

v. does not have the `assoc` attribute.

then:

a. the first argument's gathering pattern will change to `e` if and only if the subsort relations allow it to right associate but not left-associate, and

b. the last argument's gathering pattern will change to `e` if and only if the subsort relations allow it to left associate but not right-associate.

This exception will apply, for example, to the operator `_;_` in the following module.

```
fmod LIST is
  including MACHINE-INT .
  sorts Elt List .
  subsort MachineInt < Elt < List .
  op _;_ : Elt List -> List .
endfm
```

According to the general rule, the gathering pattern should be `gather (& &)`. Since exception 1 may be applied to the two arguments of the operator, this exception would change the gathering pattern to `gather (E E)`. But exception 3 prevails over exception 1 and should be applied to the first argument, so that the real gathering pattern of this operator is: `gather (e E)`.

To illustrate these rules, we show below the default precedence and gathering patterns generated by Maude for the module `BINARY-NAT` presented in Section 2.7.2.

```
op 0 : -> Bit .
op 1 : -> Bit .
op nil : -> Bits .
op __ : Bits Bits -> Bits [assoc id: nil prec 41 gather (e E)] .
op |_| : Bits -> MachineInt [prec 0 gather (&)] .
op normalize : Bits -> Bits .
op _+_ : Bits Bits -> Bits [assoc comm prec 41 gather (e E)] .
op _*_ : Bits Bits -> Bits [assoc comm prec 41 gather (e E)] .
op _^_ : Bits Bits -> Bits [prec 41 gather (E E)] .
op _>_ : Bits Bits -> Bool [prec 41 gather (E E)] .
op _?_:_ : Bool Bits Bits -> Bits [prec 41 gather (E & E)] .
op not_ : Bits -> Bits [prec 15 gather (E)] .
```

2.7.6 Tokens, Bubbles and Metaparsing

In order to generate in Maude an *environment* for a language $\mathcal{L}$, including the case of a language with user-definable syntax, the first thing we need to do is to define the syntax for $\mathcal{L}$-modules. This can be done by extending the module `META-LEVEL` with a data type $\texttt{Module}_{\mathcal{L}}$ for $\mathcal{L}$-modules, and with other auxiliary data types for commands and other constructs. Maude provides great flexibility to do this thanks to its mixfix front-end and to the use of bubbles[17] (any nonempty string of Maude identifiers). The intuition behind bubbles is that they correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available.

The idea is that for a language that allows modules with user-definable syntax—as it is the case for Maude—it is natural to see its syntax as a combined syntax, at two different levels: what we may call the *top level* syntax of the language, and the user-definable syntax introduced in each module. The data type bubble allows us to reflect this duality of levels in the syntax definition. Similar ideas have been exploited using ASF+SDF [59].

To illustrate this concept, suppose that we want to define the syntax of Maude in Maude. Consider the following Maude module:

```
fmod NAT3 is
   sort Nat3 .
   op s_ : Nat3 -> Nat3 .
   op 0 : -> Nat3 .
   eq s s s 0 = 0 .
endfm
```

Notice that the strings of characters inside the boxes are not part of the top level syntax of Maude. In fact, they can only be parsed with the grammar associated to the signature of the module `NAT3`. In this sense, we say that the syntax for Maude modules is a combination of two levels of syntax. The term `s s s 0`, for example, has to be parsed in the grammar associated to the signature of `NAT3`. The definition of the syntax of Maude in Maude must reflect this duality of syntax levels.

So far, we have talked about bubbles in a generic way. In fact, there can be many different kinds of bubbles. In Maude we can define different types of bubbles as built-in data types by parameterizing their definition. Thus, for example, a bubble of length one, which we call a *token*, can be defined as follows.

```
sort Token .

op token : Qid -> Token
       [special
           (id-hook Bubble        (1 1)
            op-hook qidBaseSymbol (<Qids> : -> Qid))] .
```

Any name can be used to define a bubble sort. It is the special attribute

```
id-hook Bubble        (1 1)
```

that makes the sort `Token` a bubble sort. The second argument of the `id-hook` special attribute indicates the minimum and maximum length of such bubbles as strings of identifiers. Therefore, `Token` has only bubbles of size 1. To specify a bubble of any length we would use the pair of values 1 and -1. The operator used in the declaration of the bubble, in this case the operator `token`, is a bubble constructor that represents tokens in terms of their quoted form. For example, the token `abc123` is represented as `token('abc123)`.

We can define bubbles of any length, that is, nonempty sequences of Maude identifiers, with the following declarations.

[17] In the current version bubbles can only be used in modules passed to the function `meta-parse` as arguments. The way of using and defining bubbles will change in future releases.

```
sort Bubble .

op bubble : QidList -> Bubble
       [special
           (id-hook Bubble        (1 -1)
            op-hook qidListSymbol
                    (__ : QidList QidList -> QidList)
            op-hook qidBaseSymbol (<Qids> : -> Qid))] .
```

In this case, the system will represent the bubble as a list of quoted identifiers under the constructor `bubble`. For example, the bubble `ab cd ef` is represented as `bubble('ab 'cd 'ef)`.

Different types of bubbles can be defined using the `id-hook` special attribute `Exclude`, which takes as parameter a list of identifiers to be excluded from the given bubble, that is, the bubble being defined cannot contain such identifiers.

Suppose that instead of the declarations given in the module `MINI-MAUDE-SYNTAX` below, we had instead given the following two declarations to specify the syntax of an operator declaration in a module.

```
op op_: ->_. : Token Token -> Decl .
op op_:_->_. : Token Bubble Token -> Decl .
```

With these declarations, bubbles could be bigger than expected. In particular, the following parse would be possible.

```
op  s_  :  Nat3 -> Nat3 . op 0 :  ->  Nat3  .
```

We can use the `id-hook` special attribute `Exclude` to avoid this situation. We can declare the sort `NeTokenList` with constructor `neTokenList` as a list of identifiers, of any length greater than one, excluding[18] the identifier '`.`' with the following declarations.

```
sort NeTokenList .

op neTokenList : QidList -> NeTokenList
       [special
           (id-hook Bubble        (1 -1)
            op-hook qidListSymbol
                    (__ : QidList QidList -> QidList)
            op-hook qidBaseSymbol (<Qids> : -> Qid)
            id-hook Exclude (.))] .
```

We are now ready to give the signature to parse modules such as `NAT3` above. The following module `MINI-MAUDE-SYNTAX` uses the above definitions of sorts `Token`, `Bubble` and `NeTokenList` to define the syntax of a sublanguage of Maude, namely, many sorted, unconditional, functional modules, in which the declarations of sorts, variables and operators have to be done one at a time, and in which no attributes are supported for operators.

```
fmod MINI-MAUDE-SYNTAX is
  including QID-LIST .
  sorts Bubble Token NeTokenList
        PreModule PreCommand
        Decl DeclList .
```

[18] In general, to exclude identifiers `I1, I2, ..., Ik`, we use the syntax `Exclude (I1 I2 ... Ik)`.

```
    subsort Decl < DeclList .

    op token : Qid -> Token
           [special
               (id-hook Bubble       (1 1)
                op-hook qidBaseSymbol (<Qids> : -> Qid))] .
    op bubble : QidList -> Bubble
           [special
               (id-hook Bubble       (1 -1)
                op-hook qidListSymbol
                        (__ : QidList QidList -> QidList)
                op-hook qidBaseSymbol (<Qids> : -> Qid))] .
    op neTokenList : QidList -> NeTokenList
           [special
               (id-hook Bubble       (1 -1)
                op-hook qidListSymbol
                        (__ : QidList QidList -> QidList)
                op-hook qidBaseSymbol (<Qids> : -> Qid)
                id-hook Exclude (.))] .

    *** sort declaration
    op sort_. : Token -> Decl .

    *** operator declaration
    op op_: ->_. : Token Token -> Decl .
    op op_:_->_. : Token NeTokenList Token -> Decl .

    *** variable declaration
    op var_:_. : Token Token -> Decl .

    *** equation declaration
    op eq_=_. : Bubble Bubble -> Decl .

    *** functional module
    op fmod_is_endfm : Token DeclList -> PreModule .
    op __ : DeclList DeclList -> DeclList [assoc gather(e E)] .
  endfm
```

Notice how we explicitly declare operators that correspond to the top level syntax of Maude, and how we represent as terms of sort `Bubble` those pieces of the module—namely, terms in equations—that can only be parsed with the user-defined syntax.

Then, the functional module `NAT3` above can be parsed as a term of sort `PreModule` in `MINI-MAUDE-SYNTAX`. The name of this sort reflects the fact that not all terms of sort `PreModule` do actually represent Maude modules. In particular, for a term of sort `PreModule` to represent a Maude module all the bubbles must be correctly parsed as terms in the module's user-defined syntax.

When calling the function `meta-parse` with the meta-representation of the module `MINI-MAUDE-SYNTAX`[19] and the previous module transformed into a list

[19] As pointed out in Section 2.5.5, in Core Maude the meta-representation of the module `MINI-MAUDE-SYNTAX` has to be explicitly given. In Full Maude we can refer to the representation of a module using its name by any of the techniques explained in Section 3.3.

of quoted identifiers[20], that is,

```
Maude> red meta-parse(MINI-MAUDE-SYNTAX,
                      'fmod 'NAT3 'is
                          'sort 'Nat3 '.
                          'op 's_ ': 'Nat3 '-> 'Nat3 '.
                          'op '0 ': '-> 'Nat3 '.
                          'eq 's 's 's '0 '= '0 '.
                      'endfm) .
```

we get the following metaterm as a result.

```
result Term:
   'fmod_is_endfm[
       'NAT3 ,
       '__['sort_.[ 'Nat3 ],
           '__['op_:_->_.[ 's_ , 'Nat3 , 'Nat3 ],
               '__['op_:`->_.[ '0 , 'Nat3 ],
                   'eq_=_.[ 's 's 's '0 , '0 ]]]]]
```

Of course, Maude does not return these boxes. Instead, the system returns the bubbles using their constructor form as specified in their corresponding declarations. For example, the bubbles $\boxed{\texttt{'Nat3}}$ and $\boxed{\texttt{'s 's 's '0}}$ are represented, respectively, as `token('Nat3)` and `bubble('s 's 's '0)`. Maude returns them meta-represented. The result given by Maude is therefore the following.

```
result Term:
   'fmod_is_endfm[
      'token[{''NAT3}'Qid],
      '__['sort_.['token[{''Nat3}'Qid]],
          '__['op_:_->_.['token[{''s_}'Qid],
                         'neTokenList[{''Nat3}'Qid],
                         'token[{''Nat3}'Qid]],
              '__['op_:`->_.['token[{''0}'Qid],
                             'token[{''Nat3}'Qid]],
                  'eq_=_.['bubble['__[{''s}'Qid, {''s}'Qid,
                                      {''s}'Qid, {''0}'Qid]],
                          'bubble[{''0}'Qid]]]]]]
```

This result is a metaterm of sort `Term`. To convert this term into a term of sort `FModule` is now straightforward. As already mentioned, we first have to extract from the term the module's signature. For this, we can use an equationally defined function

```
op extractSignature : Term -> FModule? .
```

Notice that `extractSignature` is a partial function that is not well defined for metaterms of sort `Term` that do not meta-represent terms of sort `PreModule` in `MINI-MAUDE-SYNTAX`. Therefore, the result is in general an element of an error supersort `FModule?` of `FModule`. Once we have the signature of the module—expressed as a functional module with no equations and no membership axioms—we can then build terms of sort `FModule` in the same way with

[20] We shall see in Section 2.8 that this representation of the input as a list of quoted identifiers is given automatically by the read-eval-print loop supported by the built-in module `LOOP-MODE`.

the equationally defined function `solveBubbles`, that recursively replaces each bubble in an equation by the result of calling `meta-parse` with the already extracted signature and with the quoted identifier form of the bubble.

```
op solveBubbles : Term FModule -> FModule? .
```

The partial function `processPreModuleTerm` takes a term and, if it has the appropriate form—that is, if it is a term meta-representing a term of sort `PreModule` in `MINI-MAUDE-SYNTAX`, and, furthermore, the `solveBubbles` function succeeds in parsing the bubbles in equations as terms—then it returns a term of sort `FModule`.

```
op processPreModuleTerm : Term -> FModule? .

eq processPreModuleTerm(T)
  = solveBubbles(T, extractSignature(T)) .
```

We have then the following reduction.

```
red processPreModuleTerm(
       meta-parse(MINI-MAUDE-SYNTAX,
                  'fmod 'NAT3 'is
                      'sort 'Nat3 '.
                      'op 's_ ': 'Nat3 '-> 'Nat3 '.
                      'op '0 ': '-> 'Nat3 '.
                      'eq 's 's 's '0 '= '0 '.
                  'endfm)) .

Result FModule : fmod 'NAT3 is
                    nil
                    sorts 'Nat3 .
                    none
                    op '0 : nil -> 'Nat3 [none] .
                    op 's_ : 'Nat3 -> 'Nat3 [none] .
                    none
                    none
                    eq 's_['s_['s_[{'0}'Nat3]]] = {'0}'Nat3 .
                 endfm
```

2.8 `LOOP-MODE` and Metalanguage Uses

Using object-oriented concepts, we can specify in Maude a general input/output facility provided by the `LOOP-MODE` module shown below, that extends the module `QID-LIST`, into a generic read-eval-print loop.

```
mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System
         [special ( ... )] .
endm
```

The operator `[_,_,_]` can be seen as a persistent object with an input channel (the first argument), an output channel (the third argument), and a state (given by its second argument). This read-eval-print loop that LOOP-MODE provides is a simple mechanism developed for this release that may not be maintained in future versions. We plan to endow Maude with general built-in support for objects; this will make possible more general and flexible solutions for dealing with input/output and persistent objects.

Besides having input and output channels, terms of sort System give us the possibility of maintaining a persistent state in their second component. This state has been declared in a completely generic way. In fact, the sort State in LOOP-MODE does not have any constructor. This gives complete flexibility for defining the terms we want to have for representing the persistent state of the loop in each particular application. In this way we can use this input/output facility not only for extensions of Maude like Full Maude, but also for other uses of Maude as a *metalanguage*, where the object language being implemented may be completely different from Maude. For each such language or tool the nature of the state of the system may be completely different. We can tailor the State sort to any such application by importing LOOP-MODE in a module in which we define the structure of the persitent state and the rewrite rules for changing that state and interacting with the loop.

2.8.1 The Use of the Loop

We can illustrate the basic ideas with a toy example, namely a system in which the loop is used to echo each input twice. In this case there is no need to maintain any state, so we can just declare a constant null to represent the empty state.

```
mod DUPLICATE is
  including LOOP-MODE .
  op null : -> State .
  vars Input Output : QidList .
  crl [duplicate] :
     [Input, null, Output]
     => [nil, null, Output Input Input]
        if Input =/= nil .
endm
```

Once this module has been entered, we must first initialize the loop by setting its initial state using the loop command. That is, we must give to the loop command the term of sort State that we desire as initial state. For this example, we can start a loop with empty input and output channels by typing

```
Maude> loop [nil, null, nil] .
```

Since in the current release only one input channel is supported (the current terminal), the way to distinguish the input passed to the loop from the input to the Maude system—modules or commands—is by enclosing them in parentheses. When something is written in the Maude prompt enclosed in parentheses it is converted into a list of quoted identifiers. This is done by first breaking the input stream into a sequence of tokens—that is, into a sequence of Maude identifiers—and then converting each of these tokens into a quoted identifier by putting a quote in front of it, and appending the results into a list of quoted identifiers, which is then placed in the first slot of the loop object. The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop is printed on the terminal after applying the inverse process of "unquoting" each of the tokens in the list. However, the output channel is not cleared at the time when the output is printed; it is instead cleared when the next input is entered. We can think of the input and output events as *implicit rewrites* that transfer—in a slightly modified, quoted or unquoted form—the input and output data between two objects, namely the loop object and the "user" or "terminal" object.

Once the loop has been initialized we can input any data by writing it after the prompt enclosed in parentheses. For example, we can write

```
Maude> (a s d )
```

and then we get the output

```
a s d a s d
```

A somewhat more interesting example is a loop that echoes the input, but only after every ten tokens, that is, it keeps the input until the number of tokens stored in the state is ten. In this case the input introduced so far has to be stored. Therefore we now really need a persistent state, albeit a simple one. We can represent the state as a pair consisting of a list of quoted identifiers—the tokens seen so far since the last printing—and a counter measuring the length of such a list.

```
mod DUPLICATE-TEN is
  including LOOP-MODE .
  protecting MACHINE-INT .
  op <_;_> : QidList MachineInt -> State .
  op init : -> System .
  vars Input StoredInput Output : QidList .
  vars QI QI0 QI1 QI2 QI3 QI4 QI5 QI6 QI7 QI8 QI9 : Qid .
  var Counter : MachineInt .
  rl [init] :
     init => [nil, <  nil ; 0 >, nil] .
  rl [in] :
     [QI Input, < StoredInput ; Counter >, Output]
     => [Input, < StoredInput QI ; Counter + 1 >, Output] .
  rl [out] :
     [Input,
      < QI0 QI1 QI2 QI3 QI4 QI5 QI6 QI7 QI8 QI9 StoredInput ;
        Counter >,
      Output]
     => [Input,
         < StoredInput ; Counter - 10 >,
         Output QI0 QI1 QI2 QI3 QI4 QI5 QI6 QI7 QI8 QI9] .
endm

Maude> loop init .

Maude> (a b)

Maude> (c d e f g h i)

Maude> (j k l)

a b c d e f g h i j
```

We can see the state of the loop with the `continue` command as follows.

```
Maude> cont .
result System: [nil,< 'k 'l ; 2 >,'a 'b 'c 'd 'e 'f 'g 'h 'i 'j]
```

Note that, as already mentioned, the data in the output channel remains there after being printed; it is removed at the time of the next input event.

2.8.2 Metalanguage Uses of Maude

The above examples are toy examples to illustrate the basic features of `LOOP-MODE`. However, the most interesting applications of this module are *metalanguage* applications, in which Maude is used to define the syntax, parse, execute, and pretty print the execution results of a given object language or tool. In such applications, most of the hard work is done by the `META-LEVEL` module, but handling the input/output and maintaining the persistent state of the object language interpreter or tool is done by `LOOP-MODE`.

The metalanguage uses of Maude are a natural consequence of the good properties of rewriting logic as a logical and semantic framework. Indeed, one of the key goals of rewriting logic from its beginning has been to provide a *semantic framework* in which many models of computation—particularly concurrent and distributed ones—and languages can be naturally represented. Because of the intrinsic duality between logic and computation that rewriting logic supports, the very same reasons making rewriting logic a suitable semantic framework, make it also an attractive *logical framework* [32] to represent many different logics.

What is common to all these logical and semantic framework applications is that the models of computation, logics, or languages are represented in rewriting logic by mappings of the form

$$(\dagger)\ \ \Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

The representations are typically very simple and natural. They map theories or modules in $\mathcal{L}$ to rewrite theories.

For language prototyping purposes, the obvious question to ask is: how can a rewriting logic language best support representation maps of the form (†), so that it becomes a *metalanguage* in which a very wide variety of programming, specification, and design languages, and of computational and logical systems can be both *semantically defined*, and *implemented* in it?

Our answer is: by being reflective. As already explained in Section 2.5, Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective and provide efficient support of reflective computation by means of the `META-LEVEL` module.

Indeed, using `META-LEVEL` we can both make the above representation map Φ executable, and we can execute the resulting rewrite theory representing a theory or module in $\mathcal{L}$, thus getting an implementation of $\mathcal{L}$ in Maude. Specifically, we can reify a representation map Φ of the form (†) by defining an abstract data type $\mathtt{Module}_{\mathcal{L}}$ representing modules in the logic or language $\mathcal{L}$. Since in `META-LEVEL` we also have a data type `Module` whose terms represent rewrite theories, we can then *internalize* the representation map Φ as an equationally defined function

$$\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module}.$$

In fact, thanks to the general meta-result of Bergstra and Tucker [1], any computable representation map Φ can be specified in this way by a finite number of Church-Rosser and terminating equations.

Having this representation map defined in Maude, we can then execute in Maude the rewrite theory $\overline{\Phi}(M)$ associated to a theory or module M in $\mathcal{L}$. This has been done, for example, for linear logic in [33, 10], and for structured Maude modules in Full Maude. But it could also be done for a very wide range of other languages and logics using the same method.

By defining the data type $\texttt{Module}_{\mathcal{L}}$ in an extension of `META-LEVEL` we can indeed define the syntax of $\mathcal{L}$ within Maude. However, to provide a satisfactory execution environment for $\mathcal{L}$ in Maude, we also have to support input/output and a persistent state for interacting with the interpreter for $\mathcal{L}$ that we want to define. That is, we want to be able to enter module definitions, execute commands, and get results of executions. This is precisely what `LOOP-MODE` makes possible. As a consequence, an environment for $\mathcal{L}$ in Maude will typically be realized by a module containing both `META-LEVEL` and `LOOP-MODE` as submodules.

To illustrate the way in which `LOOP-MODE` can be used in conjunction with `META-LEVEL` for metalanguage purposes, we discuss in some detail the way in which we make use of it in the implementation of Full Maude. The complete specification can be found in [19]. In Full Maude, the state of the system is given by a single object of class `database`. This object has attributes `db`, to keep the actual database in which all the modules being entered are stored, an attribute `default`, to keep the identifier of the current module by default, and attributes `input` and `output` to simplify the communication of the loop with the database. Using the notation for classes in object-oriented modules (see Section 3.2) we can declare the class `database` as follows[21].

```
class database | db : Database, input : TermList,
                 output : QidList, default : ModId .
```

Since we assume that `database` is the only object class that has been defined—so that the only objects of sort `Object` will belong to the `database` class—to specify the admissible states in the persistent state of `LOOP-MODE` for Full Maude, it is enough to give the subsort declaration

```
subsort Object < State .
```

We now give the rules to initialize the loop, and to specify the communication between the loop—the input/output of the system—and the database. Depending on the kind of input that the database receives, its state will be changed or some output will be generated. Before giving the rules we need some declarations. We start declaring a constant `o` of sort `Oid` to identify the persistent `database` object, and a constant `init` to name the initial value of the loop.

```
op o : -> Oid .
op init : -> System .
```

The rule specifying the initial value of the loop is given below. In it, `initialDatabase` is a constant naming the initial database[22].

```
rl [init] :
   init
```

[21] Note that since the module `FULL-MAUDE` is a system module in Core Maude, object-oriented declarations such as this one cannot be given directly. Instead, the equivalent declarations desugaring the desired object-oriented module have to be specified.

[22] Possibly containing the library of predefined modules and some other predefined modules, such as `CONFIGURATION`.

```
   => [nil,
       < o : database |
           db : initialDatabase, input : nilTermList,
           output : nil, default : nullModId >,
       nil] .
```

To initialize the loop we have to write

```
Maude> loop init .
```

When some text has been introduced in the loop, the first argument of the operator `[_,_,_]` is different from `nil`, and we can use this fact to activate the following rule, that enters an input such as a module or a command from the user into the database. The constant `grammar` names the module containing the signature defining the top level syntax of Full Maude (see Appendix C). This grammar is used by the `meta-parse` function in `META-LEVEL` to parse the input. In case of being a syntactically valid input, the parsed input is placed in the `input` attribute of the database object; otherwise, an error message is placed in the output channel of the loop.

```
crl [in] :
   [QIL,
    < o : database | db : DB,
                     input : nilTermList,
                     output : nil,
                     default : MI >,
    QIL']
   => if meta-parse(grammar, QIL) == error*
      then [nil,
            < o : database | db : DB,
                     input : nilTermList,
                     output : ('ERROR: 'incorrect 'input '.),
                     default : MI >,
            QIL']
      else [nil,
            < o : database | db : DB,
                     input : meta-parse(grammar, QIL),
                     output : nil,
                     default : MI >,
            QIL']
      fi
      if QIL =/= nil .
```

When the `output` attribute of the persistent object contains a nonempty list of quoted identifiers, the `out` rule moves it to the third argument of the loop. Then the Core Maude system displays it in the terminal.

```
crl [out] :
   [QIL,
    < o : database | db : DB, input : TL,
                     output : QIL', default : MI >,
    QIL'']
   => [QIL,
       < o : database | db : DB, input : TL,
                        output : nil, default : MI >,
```

```
        (QIL' QIL'')]
       if QIL' =/= nil .
```

For each particular language, the rewrite rules defining the system behavior for different language commands are specified according to the specific details of the language in question. We illustrate below the case of Full Maude. In Full Maude there is a function `processUnit` that takes as arguments the result of the call to `meta-parse`, an empty module of the kind of the module being introduced—named by the constant `emptyStrFModule` in the rule below—and the current database. It returns a modified copy of the database after processing the new module. Thus, for example, for the case of the constructor of functional modules in the module `grammar`

```
op fmod_is_endfm : Token DeclList -> PreModule .
```

the processing of a functional `PreModule` once it has been entered into the system is done by the rule

```
rl [functional-module] :
   < o : database | db : DB,
                    input : ('fmod_is_endfm[T, T']),
                    output : nil,
                    default : MI >
   => < o : database |
         db : processUnit(T, T', emptyStrFModule, DB),
         input : nilTermList,
         output : ('Introduced 'module:
                   modIdToQid(parseModName(T))),
         default : parseModName(T) > .
```

Note the message placed in the output channel, and the change in the current module by default, which is now the new module just processed. Since the name `T` of the module can be complex—a module expression—, some extra parsing has to be performed by the auxiliary function `parseModName`.

User-defined commands are handled by rules as well. For example, the `show module` command, which prints the specified module, or the current one if no module name is specified, is handled by the following rules.

```
rl [show-module-1] :
   < o : database | db : DB,
                    input : ({'show`module`.}'PreCommand),
                    output : nil, default : MI >
   => < o : database |
          db : DB, input : nilTermList,
          output : meta-pretty-print(
                      getFlatModule(MI, DB),
                      getTopModule(MI, DB)),
          default : MI >  .
rl [show-module-2] :
   < o : database | db : DB, input : ('show`module_.[T]),
                    output : nil, default : MI >
   => < o : database |
          db : DB, input : nilTermList,
          output : meta-pretty-print(
                      getFlatModule(parseModExp(T), DB),
```

```
                   getTopModule(parseModExp(T), DB)),
        default : MI > .
```

The functions `getTopModule` and `getFlatModule` return, respectively, the module as introduced by the user and its flattened version[23] as stored in the database.

2.9 System Issues and Debugging

2.9.1 Command Line Options

The interpreter is started by the command

`maude` *flag** *file**

where `maude` is the name of the executable (it might be called something like `maude.linux` on a linux box). The file `prelude.maude` should normally be in the same directory as the maude executable. If any files are specified, they will be read in after `prelude.maude`, but before the interpreter reads from the standard input. Currently understood flags are:

-no-mixfix
: Start the interpreter in prefix mode. This is intended for noninteractive use with a post processor.

-no-prelude
: Do not attempt to read in `prelude.maude` on start up.

-batch
: Do not handle control-C.

2.9.2 Debugging Core Maude Specifications

There are two approaches to debugging Core Maude specifications. The most general technique is to turn tracing on with the command

```
set trace on .
```

and capture a log of the trace using *script* or xterm logging. This can then be studied using a text editor. Since the trace is usually voluminous, there are a number of trace options to control just what is traced. One of the more useful is selective tracing:

```
set trace select on .
trace select foo bar ([_,_]) .
```

This will cause only rewrites where the redex is headed by operators with the selected names to be traced. Note that these operators need not be in existence at the time the `trace select` command is executed; thus it is possible to select operators that will only be created at runtime via the metalevel.

The other approach is to use the Core Maude debugger. When reductions are happening and a control-C interrupt is received, the debugger is automatically entered. The prompt changes to `Debug(`n`)>` where n is the debug level; or the number of times the debugger has been re-entered (it is fully re-entrant). All top level commands can be executed from the debugger, along with four commands that are special to the debugger:

[23] We call flattened module to the version of the module in which the module and all its submodules have been collapsed to a single module.

`where .`
Prints out the stack of pending rewrites, explaining how each one arose.

`step .`
Executes the next rewrite with tracing turned on.

`resume .`
Exits the debugger and continues with the current rewriting task.

`abort .`
Exits the debugger and abandons the current rewriting task.

Since it is sometimes useful to enter the debugger just before the first rewrite takes place, the `reduce`, `rewrite` and `continue` commands can be prefixed by the `debug` keyword to accomplish this. For example:

```
debug rewrite [8] in MY-SPEC : init-symbol .
```

2.9.3 User Facilities Not Yet Implemented

There are a number of important features that have not yet been implemented in Core Maude in this release. Here is a list of the most important omissions.

1. Module renamings[24].
2. Garbage collection for top level modules.
3. Special strategy for the `rewrite` command in the case of object-oriented modules, and built-in objects and messages.
4. Various built-in data types.
5. Error handling and recovery for certain situations.
6. Rules whose conditions can contain not only equations and membership axioms, but other rewrites.
7. Memoization.
8. Operators that have both `assoc` and `idem` attributes.

2.9.4 Miscellaneous Differences from OBJ3

1. True rewriting modulo identity is implemented rather than the restricted version done by OBJ3; this leads to nontermination more often.
2. The attribute `idr:` is not recognized.
3. Maude allows arbitrary forward references to sorts, variables and operators within a module. The order of statements in a module is irrelevant, except, possibly, on error messages, importations, and nonconfluent systems.
4. Identities are not restricted to being constants; an identity may be any ground term that does not have the parent operator on top. Cyclic dependencies between two or more identity elements is explicitly allowed and is correctly resolved by the matching algorithms.

[24] Renamings and, more generally, module expressions are of course supported in Full Maude.

5. The attribute `idem` means rewriting modulo idempotence, rather than adding the equation for idempotence.

6. The lefthand side of an equation can be a single variable; this often (but not always) leads to nontermination.

7. Error supersorts are used instead of retracts; this is due to the use of membership equational logic, so that the error supersorts exactly correspond to kinds.

8. Operator strategies available with the `strat` attribute do not support negative integers (evaluate on demand) and are restricted to a subset of "sensible" strategies that depend on the operator's other attributes. Various OBJ3 strategy bugs are not emulated [21].

9. Operators which have the `assoc` attribute may be used in "flattened" form, e.g., `f(a, b, c)` instead of `f(a, f(b, c))` or `f(f(a, b), c)`.

10. Operators may always be used in prefix form, e.g., `if_then_else_fi(b, t, e)` instead of `if b then t else e fi`.

11. Labeled equations are not supported (use labeled rules).

12. Sort declarations of operators with `assoc`, `comm`, `id:` and `idem` attributes must respect those attributes: rearrangement or permutation by associativity or commutativity must not change the sort of a term, while collapse due to identity or idempotence must either lower the sort of a term or leave it unchanged.

13. There is no support for Lisp—the Maude interpreter is written in C++.

2.9.5 Traps for the Unwary

Bare Variable Lefthand Sides

The use of a bare variable lefthand side for an equation, rule, or membership axiom may lead to unexpected nontermination. The recommended place to use them is in rules which are only going to be applied via a strategy language. Using them in membership axioms is seductive, but very tricky. For example:

```
subsort Prime < Nat .
var N : Nat .
cmb N : Prime if favoritePrimeTest(N) .
```

will end up with the membership axiom and `favoritePrimeTest` being applied to every reduced term of sort `Nat`, including those that arise during evaluation of `favoritePrimeTest(N)` with likely nontermination.

Collapse Theories

Using `id:` or `idem` attributes means that you are (notionally) working with infinite congruence classes and that many lefthand side patterns will match in unexpected ways. Unlike OBJ3, Maude has true collapse matching algorithms, rather than identity completion, and it does not try to omit problematic matches. Consider for example the module

```
fmod FOO is
  sort Foo .
  ops a e : -> Foo .
  op f : Foo Foo -> Foo [left id: e] .
var X : Foo .
  eq f(X, a) = ...
endfm
```

Then we have

```
a = f(e, a) = f(e, f(e, a)) = f(f(e, e), a) = ...
```

In particular, the pattern `f(X, a)` matches `a` with `X` $\leftarrow$ `e` leading to possible nontermination. You should be wary of having an operator with an identity element as the top symbol for a lefthand side. One useful trick when you need a pattern like `f(X, a)` is to use a pattern `f(Y, a)` where Y has a sort lower than that of the identity element. For example:

```
fmod NAT is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat .
  op s : Nat -> NzNat .
  op + : Nat Nat -> Nat [assoc comm id: 0] .
  op + : Nat NzNat -> Nat [assoc comm id: 0] .
var X : Nat .
var Y : NzNat .
  eq +(s(X), Y) = s(+(X, Y)) .
endfm
```

Here `+(s(X), Y)` cannot match `s(0)` because, although `s(0) = +(s(0), 0)` by the identity attribute, `Y` cannot match `0`.

Rewriting with the `idem` attribute is even riskier. For example:

```
fmod FOO2 is
  sort Foo .
  ops a b : -> Foo .
  op f : Foo Foo -> Foo [idem] .
var X : Foo .
  eq a = b .
endfm
```

We then have

```
a = f(a, a) = f(f(a, a), a) = f(a, f(a, a)) = ...
```

And thus, if `a` can be rewritten by an equation, then any number of rewrites can be done by using the `idem` axiom to create new copies of `a`. In fact, the current implementation would choose the obvious rewrite and just produce `b`, but this should not be relied on; `FOO2` is a nonterminating system. The only safe way to use `idem` is as follows. Whenever a connected component is the domain and range of an operator having the `idem` attribute, then its sorts are *poisoned*. Terms of poisoned sorts must never rewrite other than by rules under the control of a strategy. They must be built out of free (other than attributes) constructors. Of course it is ok to have defined functions that work on such constructor terms; it is just that the terms themselves may not rewrite.

One-sided Identities and Associativity

When the associativity axiom is combined with a one-sided identity axiom some unexpected matching properties result. Consider the module:

```
fmod BAR is
  sort Foo .
  ops a b 1f : -> Foo .
  op f : Foo Foo -> Foo [assoc left id: 1f] .
var X Y : Foo .
endfm
```

Then

```
match f(X, Y) <=? f(a, b) .
```

yields three solutions:

```
Solution 1
  X:Foo --> 1f
  Y:Foo --> f(a, b)

Solution 2
  X:Foo --> a
  Y:Foo --> b

Solution 3
  X:Foo --> f(a, 1f)
  Y:Foo --> b
```

whereas the naive user may not have expected the last solution. Matching with extension can be even more surprising:

```
xmatch f(X, Y) <=? f(a, b) .
```

yields five solutions:

```
Solution 1
  Matched portion = f(a, 1f)
  X:Foo --> a
  Y:Foo --> 1f

Solution 2
  Matched portion = f(a, 1f)
  X:Foo --> f(a, 1f)
  Y:Foo --> 1f

Solution 3
  Matched portion = (whole)
  X:Foo --> 1f
  Y:Foo --> f(a, b)

Solution 4
  Matched portion = (whole)
  X:Foo --> a
  Y:Foo --> b
```

```
Solution 5
  Matched portion = (whole)
  X:Foo --> f(a, 1f)
  Y:Foo --> b
```

Here the first two solutions match a portion `f(a, 1f)` of the subject that was not apparent from the original problem. However, if one considers the congruence class of `f(a, b)` they are valid solutions that are necessary for correct simulation of congruence class (conditional) rewriting.

2.9.6 Known Problems

1. The user input is not checked very rigorously, and some kinds of undetected errors can cause core dumps while other kinds of errors are fatal. This problem will gradually go away as error handling and recovery is improved.

2. The interpreter can be very stack hungry when working on deep terms. This is an unfortunate consequence of the highly modular design of the rewrite engine. Stack overflow usually manifests itself as a segmentation fault with a corrupted core dump; try the UNIX command *unlimit stack size* before running the interpreter. Stack overflow is also characteristic of nonterminating computations.

3. Heap usage for storing and processing large modules can be quite large. In particular, the memory required for parsing can be quadratic in the size of the (flattened) signature. This problem is magnified because top level modules are not yet garbage collected.

4. Response to control-C can be delayed a very long time when the interpreter is reading in modules or printing results.

Chapter 3

Full Maude

During the development of the Maude system we have put special emphasis on the creation of metaprogramming facilities to allow the generation of execution environments for a wide variety of languages and logics. The first most obvious area where Maude can be used as a metalanguage is in building language extensions for Maude itself. Our experience in this regard—first reported in [18], and further documented here and in [19]—is very encouraging. We have been able to define in Core Maude a language extension with notation for object-oriented programming, parameterized modules, views (for module instantiation) and module expressions [18]. Furthermore, using the `META-LEVEL` and `LOOP-MODE` modules, we have also been able to define in Core Maude all the additional functionality required for parsing, evaluating, and pretty printing modules in the extended language, and also for input/output interaction, as already discussed in Sections 2.5.6 and 2.8.

Thanks to the efficient implementation of the rewrite engine, the parser, and the module `META-LEVEL`, such a language extension executes with reasonable efficiency. In the future, however, we may support in Core Maude a significant part of the functionality currently supported by Full Maude. Full Maude contains Core Maude as a sublanguage, so that Core Maude modules can also be entered at the Full Maude level. However, at present there are a few syntactic restrictions that have to be satisfied by modules and commands in order to be acceptable inputs at the Full Maude level. These syntactic restrictions are explained in Section 3.6; they will be removed in the future.

Since the execution environment for Full Maude has been implemented in Core Maude, to initialize the system so that we can start using it, the first thing we have to do is to load the `FULL-MAUDE` module in the system. Assuming that the file `full-maude.maude`, containing such specification, is located in the current directory, we just need to type the corresponding `in` command in the Maude prompt.

```
Maude> in full-maude.maude
```

The Full Maude system is then loaded and we can use it as any other module. Before entering any module or executing any command in Full Maude we need to initialize the system. Full Maude uses the `LOOP-MODE` module in order to allow the entering of modules into the system and to maintain a persistent database in which to store all the modules, theories and views being introduced. To start the *loop* we need to type

```
Maude> loop init .
```

where `init` is a constant of sort `System` giving the initial state of the Full Maude database.

We are now ready. Let us recall from Section 2.8 that to get something into the `LOOP-MODE` system, the text *has to be enclosed in parentheses*. This means that any module or command intended for Full Maude has to be written enclosed in parentheses. Notice that, since Core Maude is still active—indeed, it now provides what might be called the *system programming* level—it will handle any input not enclosed in parentheses. This allows the possibility of using both systems at the same time.

3.1 Functional and System Modules

A Core Maude module, such as those presented in previous sections, can be entered in Full Maude by enclosing it in parentheses. For example, the module `PATH` given in Section 2.1 can be entered to Full Maude as follows.

```
Maude> (fmod PATH is
          protecting MACHINE-INT .

          sorts Edge Path Path? Node .
          subsorts Edge < Path < Path? .

          ops n1 n2 n3 n4 n5 : -> Node .
          ops a b c d e : -> Edge .
          op _;_ : Path? Path? -> Path? [assoc] .
          ops source target : Path -> Node .
          op length : Path -> MachineInt .

          var E : Edge .
          var P : Path .

          cmb E ; P : Path if target(E) == source(P) .

          ceq source(E ; P) = source(E) if E ; P : Path .
          ceq target(P ; E) = target(E) if P ; E : Path .
          eq length(E) = 1 .
          ceq length(E ; P) = 1 + length(P) if E ; P : Path .

          eq source(a) = n1 .
          eq target(a) = n2 .
          eq source(b) = n1 .
          eq target(b) = n3 .
          eq source(c) = n3 .
          eq target(c) = n4 .
          eq source(d) = n4 .
          eq target(d) = n2 .
          eq source(e) = n2 .
          eq target(e) = n5 .
        endfm)
```

As in Core Maude, we can enter any module or command by writing it directly after the prompt, or by having it in a file and then using the `in` command of Core Maude. Also as in Core Maude, we can write several Full Maude modules or commands in a file and then enter all of them with a single `in` command; but each of the modules or commands has to be enclosed in parentheses. In Full Maude, in addition to functional and system modules and some of the Core Maude commands, we can enter object-oriented modules, parameterized modules, theories, views and some additional commands. We discuss all these concepts in the coming sections.

As we will discuss in Section 3.4, we can do some reduction or rewriting using a syntax for commands such as that of Core Maude, although with some minor differences that will be explained in Section 3.4.

```
Maude> (red b ; c ; d .)
Result Path? : b ; c ; d

Maude> (red length(b ; c ; d) .)
Result NzMachineInt : 3

Maude> (red a ; b ; c .)
Result Path? : a ; b ; c

Maude> (red source(a ; b ; c) .)
Result errorSort(Node) : source(a ; b ; c)

Maude> (red target(a ; b ; c) .)
Result errorSort(Node) : target(a ; b ; c)

Maude> (red length(a ; b ; c) .)
Result errorSort(MachineInt) : length(a ; b ; c)
```

In the rest of this chapter we describe the syntax of Full Maude giving the declarations of sorts, subsort relations, and operators included in the actual module used in the parsing of the inputs to Full Maude. We use the techniques described in Section 2.7.6 to parse these inputs. In addition to the declarations for `Token`, `Bubble`, and `NeTokenList` introduced in Section 2.7.6, we need to add declarations for two new kinds of tokens, namely `ViewToken` and `SortToken`. These two sorts are just particular cases of tokens which exclude several identifiers.

```
op viewToken : Qid -> ViewToken
       [special
           (id-hook Bubble        (1 1)
            op-hook qidBaseSymbol (<Qids> : -> Qid)
            id-hook Exclude (assoc associative
                             comm commutative
                             idem idempotent))] .
op sortToken : Qid -> SortToken
       [special
           (id-hook Bubble        (1 1)
            op-hook qidBaseSymbol (<Qids> : -> Qid)
            id-hook Exclude ([ ] < to : , . ( )))] .
```

The basic syntax for the declarations of sorts, subsort relations, and operations in Full Maude modules is given by the following declarations[1].

[1] We do not give all the declarations in this chapter. The complete set of declarations can be found in Appendix C.

```
subsorts SortToken < Sort < SortList .
subsort Attr < AttrList .

op __ : SortList SortList -> SortList [assoc] .
op sort_. : SortList -> SortDecl .
op sorts_. : SortList -> SortDecl .

op _<_ : SortList SortList -> SubsortRel .
op _<_ : SortList SubsortRel -> SubsortRel .
op subsort_. : SubsortRel -> SubsortDecl .
op subsorts_. : SubsortRel -> SubsortDecl .

op assoc : -> Attr .
op comm : -> Attr .
op id:_ : Bubble -> Attr .
op left id:_ : Bubble -> Attr .
op right id:_ : Bubble -> Attr .
op strat(_) : NeTokenList -> AttrList .
op prec_ : Token -> Attr .
op gather(_) : NeTokenList -> Attr .
op idem : -> Attr .
op __ : AttrList AttrList -> AttrList [assoc] .

op op_: ->_. : Token Sort -> OpDecl .
op op_: ->_[_]. : Token Sort AttrList -> OpDecl .
op op_:_->_. : Token SortList Sort -> OpDecl .
op op_:_->_[_]. : Token SortList Sort AttrList -> OpDecl .
op ops_: ->_. : NeTokenList Sort -> OpDecl .
op ops_: ->_[_]. : NeTokenList Sort AttrList -> OpDecl .
op ops_:_->_. : NeTokenList SortList Sort -> OpDecl .
op ops_:_->_[_]. : NeTokenList SortList Sort AttrList -> OpDecl .
```

Full Maude supports, not only module hierachies, that is, acyclic graphs of module importations, as discussed in Section 2.3 for Core Maude, but also parameterized programming techniques in the OBJ3 style. In particular, Full Maude supports importations in `including` and `protecting` modes, not only for user-defined modules, but, as we will see in Section 3.5.4, for module expressions as well. The syntax for importation declarations is as follows.

```
subsort Token < ModExp .

op including_. : ModExp -> ImportDecl .
op protecting_. : ModExp -> ImportDecl .
```

The syntax for the declaration of variables, membership axioms, equations, and rules is the following.

```
op vars_:_. : NeTokenList Sort -> VarDecl .
op var_:_. : NeTokenList Sort -> VarDecl .

op mb_:_. : Bubble Sort -> MembAxDecl .
op cmb_:_if_. : Bubble Sort Bubble -> MembAxDecl .

op eq_=_. : Bubble Bubble -> EquationDecl .
```

```
op ceq_=_if_. : Bubble Bubble Bubble -> EquationDecl .

op rl[_]:_=>_. : Token Bubble Bubble -> RuleDecl .
op crl[_]:_=>_if_. : Token Bubble Bubble Bubble -> RuleDecl .
```

Finally, the top syntax for functional and system modules is given by the following declarations.

```
subsort VarDecl < VarDeclList .
op __ : VarDeclList VarDeclList -> VarDeclList [assoc] .

subsorts ImportDecl SortDecl SubsortDecl OpDecl
         MembAxDecl EquationDecl VarDeclList < FDeclList .
op __ : FDeclList FDeclList -> FDeclList [assoc] .

subsorts RuleDecl FDeclList < SDeclList .
op __ : SDeclList SDeclList -> SDeclList [assoc] .

subsort Token < ModuleName .
op fmod_is_endfm : ModuleName FDeclList -> PreModule .
op mod_is_endm : ModuleName SDeclList -> PreModule .
```

3.2 Object-Oriented Modules

In a concurrent object-oriented system the concurrent state, which is usually called a *configuration*, has typically the structure of a multiset made up of objects and messages that evolves by concurrent ACU-rewriting[2] using rules that describe the effects of *communication events* between some objects and messages. Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent, and then causing "communication events" by application of rewrite rules. Therefore, we can view concurrent object-oriented computation as *deduction* in rewriting logic; in this way, the configurations S that are *reachable* from a given initial configuration S_0 are exactly those such that the sequent $S_0 \longrightarrow S$ is *provable* in rewriting logic using the rewrite rules that specify the behavior of the given object-oriented system.

An *object* in a given state is represented as a term

$$\texttt{<}\ O : C \ \texttt{|}\ a_1 \texttt{:}\ v_1 \texttt{,}\ \ldots \texttt{,}\ a_n \texttt{:}\ v_n\ \texttt{>}$$

where O is the object's name or identifier, C is its class identifier, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*. An object with no attributes can be represented as

$$\texttt{<}\ O : C \ \texttt{|}\ \texttt{>}$$

Messages do not have a fixed syntactic form. Such syntactic form can be defined by the user for each application. The concurrent state of an object-oriented system is then a multiset of objects and messages, called a `Configuration`, with multiset union described with empty syntax `__`.

The following module `CONFIGURATION` defines the basic concepts of concurrent object systems. Note that the sorts `Message` and `Attribute`—as well as the sorts `Oid` and `Cid` of object and class identifiers—are for the moment left

[2] We call rewriting modulo associativity, commutativity and identity *ACU-rewriting*.

unspecified. They will become fully defined when the CONFIGURATION module is extended by specific object-oriented definitions in a given object-oriented module.

```
fmod CONFIGURATION is
  sorts Oid Cid Attribute AttributeSet
        Object Msg Configuration .
  subsorts Object Msg < Configuration .
  subsort Attribute < AttributeSet .

  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
          [assoc comm id: none] .

  op <_:_| > : Oid Cid -> Object .
  op <_:_|_> : Oid Cid AttributeSet -> Object .

  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration
          [assoc comm id: none] .
endfm
```

In Full Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword omod—using a syntax more convenient than that of system modules because it assumes acquaintance with the basic entities, such as objects, messages and configurations, and supports linguistic distinctions appropriate for the object-oriented case. In particular, all object-oriented modules implicitly include the above CONFIGURATION module and assume its syntax. For example, the ACCNT object-oriented module below specifies the concurrent behavior of objects in a very simple class Accnt of bank accounts, each having a bal(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts.

```
(omod ACCNT is
  protecting QID .
  protecting MACHINE-INT .

  subsort Qid < Oid .

  class Accnt | bal : MachineInt .

  msgs credit debit : Oid MachineInt -> Msg .
  msg transfer_from_to_ : MachineInt Oid Oid -> Msg .

  vars A B : Oid .
  vars M N N' : MachineInt .

  rl [credit] : credit(A, M) < A : Accnt | bal : N >
     => < A : Accnt | bal : (N + M) > .
  crl [debit] : debit(A, M) < A : Accnt | bal : N >
     => < A : Accnt | bal : (N - M) >
        if N > M .
  crl [transfer] : (transfer M from A to B)
```

```
    < A : Accnt | bal : N > < B : Accnt | bal : N' >
    => < A : Accnt | bal : (N - M) >
       < B : Accnt | bal : (N' + M) >
       if N > M .
endom)
```

3.2.1 The Syntax of Object-Oriented Modules

Classes are defined with the keyword `class`, followed by the name of the class C, and by a list of attribute declarations separated by commas. Each attribute declaration has the form $a : S$, where a is an attribute identifier and S is the sort in which the values of the attribute identifier range. That is, class declarations have the form

$$\texttt{class}\ C\ \texttt{|}\ a_1 \texttt{:}\ S_1\ \texttt{,}\ \ldots\ \texttt{,}\ a_n \texttt{:}\ S_n\ \texttt{.}$$

We can declare classes without attributes using syntax

$$\texttt{class}\ C\ \texttt{.}$$

The basic syntax for class declarations is given by the following operators and subsort relationship.

```
subsort AttrDecl < AttrDeclList .
op _:_ : Token Sort -> AttrDecl [prec 40] .
op _,_ : AttrDeclList AttrDeclList -> AttrDeclList [assoc] .

op class_|_. : Sort AttrDeclList -> ClassDecl .
op class_. : Sort -> ClassDecl .
```

In this example, the only attribute of an account is its `bal`(ance), which is declared to be a value in `MachineInt`, a sort declared in the module `MACHINE-INT`. The three kinds of messages involving accounts are `credit`, `debit`, and `transfer` messages, whose user-definable syntax is introduced by the keyword `msg`. Notice the use of `msgs` to define multiple messages with the same arity in a single declaration.

The syntax for message declarations is given by the following operators.

```
op msg_:_->_. : Token SortList Sort -> MsgDecl .
op msgs_:_->_. : NeTokenList SortList Sort -> MsgDecl .
```

The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The multiset structure of the configuration provides the top-level distributed structure of the system and allows concurrent application of the rules [38]. For example, we can rewrite a simple configuration consisting of an account and a message as follows.

```
Maude> (rew < 'Peter : Accnt | bal : 2000 >
            debit('Peter, 1000) .)

Result Object : < 'Peter : Accnt | bal : 1000 >
```

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'` in an object-oriented module is just a particular case of a subsort declaration `C < C'`. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses as well as the newly defined attributes, messages, and rules of the subclass characterize the structure and behavior of the objects in the subclass.

For example, we can define an object-oriented module `SAV-ACCNT` of saving accounts introducing a subclass `SavAccnt` of `Accnt` with a new attribute `rate` recording the interest rate of the account. We leave unspecified the rules for computing and crediting the interest of an account according to its rate whose proper expression should introduce a real-time[3] attribute in account objects.

```
(omod SAV-ACCNT is
  including ACCNT .
  class SavAccnt | rate : MachineInt .
  subclass SavAccnt < Accnt .
 endom)
```

In this example, there is only one class immediately above `SavAccnt`, namely, `Accnt`. In general, however, a class C may be defined as a subclass of several classes $D_1, \ldots, D_k$, i.e., *multiple inheritance* is supported. If an attribute and its sort have already been declared in a superclass, they should not be declared again in the subclass. Indeed, all such attributes are *inherited*. In the case of multiple inheritance, the only requirement that is made is that if an attribute occurs in two different superclasses, then the sort attributed to it in each of those superclasses must be the same. In summary, a class inherits all the attributes, messages, and rules from all its superclasses. An object in the subclass behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new attributes, messages, and rules in the subclass.

As for subsort relationships, we can declare multiple subclass relationships in the same declaration. Thus, given, for example, classes `A`, ..., `H`, we can have a declaration such as

```
subclasses A B C < D E < F G H .
```

Since class names have the same form as sorts, in the signature used to parse Full Maude given in Appendix C, we use the sort `Sort` to parse them, and the sort `SortList` for lists of names of classes. The syntax for subclass declarations is given by the following operators.

```
op subclass_. : SubsortRel -> SubclassDecl .
op subclasses_. : SubsortRel -> SubclassDecl .
```

Objects in the class `SavAccnt` will have an attribute `bal` and can receive messages debiting, crediting and transferring funds exactly as any other object in the class `Accnt`. We can now rewrite a configuration, obtaining the following result.

```
Maude> (rew < 'Paul : SavAccnt | bal : 5000, rate : 3 >
            < 'Peter : Accnt | bal : 2000 >
            < 'Mary : SavAccnt | bal : 9000, rate : 3 >
            debit('Peter, 1000)
            credit('Paul, 1300)
            credit('Mary, 200) .)

Result Configuration :
            < 'Peter : Accnt | bal : 1000 >
```

[3]See [49] for a general method to specify real-time systems in rewriting logic.

```
            < 'Paul : SavAccnt | bal : 6300 , rate : 3 >
            < 'Mary : SavAccnt | bal : 9200 , rate : 3 >
```

The top level syntax for object-oriented modules is given by the following declarations.

```
subsorts SDeclList MsgDecl SubclassDecl ClassDecl < ODeclList .

op omod_is_endom : ModuleName ODeclList -> PreModule .
```

3.2.2 Transforming Object-Oriented Modules into System Modules

The best way to understand classes and class inheritance in Maude is by making explicit the full structure of an object-oriented module, which is left somewhat implicit in the syntactic conventions adopted for them. Indeed, although Maude's object-oriented modules provide convenient syntax for programming object-oriented systems, their semantics can be reduced to that of system modules. We can regard the special syntax reserved for object-oriented modules as syntactic sugar. In fact, each object-oriented module can be translated into a corresponding system module whose semantics *is* by definition that of the original object-oriented module.

However, although Maude's object-oriented modules can in this way be reduced to system modules, there are of course important conceptual advantages provided by the syntax of object-oriented modules, because it allows the user to think and express his or her thoughts in object-oriented terms whenever such a viewpoint seems best suited for the problem at hand. Those conceptual advantages would be lost if only system modules were provided.

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the `CONFIGURATION` functional module defined at the beginning of this section. The translation of a given object-oriented module extends this structure with the classes, messages and rules introduced by the module. For example, the following system module is the translation of the `ACCNT` module introduced earlier. Note that a subsort `Accnt` of `Cid` is introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of `Accnt`. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort `Accnt` is the class identifier `Accnt`.

```
mod ACCNT is
      including MACHINE-INT .
      including QID .
      including CONFIGURATION .
      sorts Accnt .
      subsort Qid < Oid .
      subsort Accnt < Cid .
      op Accnt : -> Accnt .
      op credit : Oid MachineInt -> Msg .
      op debit : Oid MachineInt -> Msg .
      op transfer_from_to_ : MachineInt Oid Oid -> Msg .
      op bal :_ : MachineInt -> Attribute .
      var A : Oid .
      var B : Oid .
      var M : MachineInt .
```

```
    var N : MachineInt .
    var N' : MachineInt .
    var V@Accnt : Accnt .
    var ATTS@0 : AttributeSet .
    var V@Accnt1 : Accnt .
    var ATTS@2 : AttributeSet .
    rl [credit] : credit(A, M)
            < A : V@Accnt | bal : N, none, ATTS@0 >
      => < A : V@Accnt | bal : (N + M), ATTS@0 > .
    crl [debit] : debit(A, M)
            < A : V@Accnt | bal : N, none, ATTS@0 >
      => < A : V@Accnt | bal : (N - M), ATTS@0 >
         if N > M = true .
    crl [transfer] : (transfer M from A to B)
            < A : V@Accnt | bal : N, none, ATTS@0 >
            < B : V@Accnt1 | bal : N', none, ATTS@2 >
      => < A : V@Accnt | bal : (N - M), ATTS@0 >
         < B : V@Accnt1 | bal : (N' + M), ATTS@2 >
         if N > M = true .
endm
```

We can describe the desired transformation from an object-oriented module to a system module as follows[4]:

- The module `CONFIGURATION` is imported.
- For each class declaration of the form `class` C `|` a_1:S_1, ... ,a_n:S_n, the following has to be introduced: a subsort C of sort `Cid`, a constant C of sort C, and declarations of operations a_i `:_ :` S_i `-> Attribute` for each attribute a_i.
- For each subclass relation C `<` C' a subsort declaration

  `subsort` C `<` C' `.`

  is introduced, and the set of attributes for objects of class C are completed with those of C'.
- The rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses, that is, not only to objects whose class identifiers are those explicitly given. The rules are then "inherited" by all objects in their subclasses by replacing the class identifiers in the objects in the rules by variables of the corresponding class sort. Variables of sort `AttributeSet` are also introduced to range over the additional attributes that may appear in objects of a subclass. That is, each object `<` O `:` C `|` ... `>` appearing in a rule, is translated into `<` O `:` X `|` ..., *Atts* `>` where the new variable X is declared of sort C, and the new variable *Atts* has sort `AttributeSet`.
- As described in [38], we simplify the notation used in object-oriented modules by giving the user the possibility of not mentioning in a given rule those attributes of an object that are not relevant for that rule. To explain this convention, let $\overline{a : v}$ denote the attribute-value pairs $a_1 : v_1, \ldots, a_n : v_n$,

[4]Notice that we have simplified the transformation of object-oriented modules into system modules that originally appeared in [38].

where the $\overline{a}$ are the attribute identifiers of a given class C (after completing it with all the attributes in its superclasses) having $\overline{S}$ as the corresponding sorts of values prescribed for those attributes. Then, in object-oriented modules we allow rules where the attributes for an object O, mentioned in the lefthand and righthand sides of a rule, need not exhaust all the object's attributes, but can instead be in any two arbitrary subsets of the object's attributes. We can picture this as follows

$$\ldots \langle O : C \mid \overline{al : vl}, \overline{ab : vb} \rangle \ldots \;\longrightarrow\; \ldots \langle O : C \mid \overline{ab : vb'}, \overline{ar : vr} \rangle \ldots$$

where $\overline{al}$ are the attributes appearing only on the *left*, $\overline{ab}$ are the attributes appearing on *both* sides, and $\overline{ar}$ are the attributes appearing only on the *right*. In the transformation into a system module, this rule is translated into

$$\begin{array}{l} \ldots \langle O : X \mid \overline{al : vl}, \overline{ab : vb}, \overline{ar : x}, \overline{ac : x'}, Atts \rangle \ldots \\ \quad \longrightarrow\; \ldots \langle O : X \mid \overline{al : vl}, \overline{ab : vb'}, \overline{ar : vr}, \overline{ac : x'}, Atts \rangle \ldots \end{array}$$

where X is a variable of sort C, $\overline{ac}$ are the attributes defined in the class C that do not appear in $\overline{al}$, $\overline{ab}$, or $\overline{ar}$, the $\overline{x}$ and $\overline{x'}$ are new variables of the appropriate sorts, and $Atts$ matches the remaining attribute-value pairs. Although the form of the rule obtained is slightly different from the one given in [38], the convention is similar to the convention presented there: The attributes mentioned only on the left are preserved unchanged, the original values of attributes mentioned only on the right do not matter, and all attributes not explicitly mentioned are left unchanged.

The rewrite rules given in the original `ACCNT` module are interpreted here—according to the conventions already explained—in a form that can be inherited by subclasses of `Accnt` that could be defined later. Thus, `SavAccnt` inherits the rewrite rules for crediting and debiting accounts, and for transferring funds between accounts that had been defined for `Accnt`.

Let us illustrate the treatment of class inheritance with the system module resulting from the transformation of the module `SAV-ACCNT` introduced previously.

```
mod SAV-ACCNT is
  including CONFIGURATION .
  including ACCNT .
  sorts SavAccnt .
  subsort SavAccnt < Cid .
  subsort SavAccnt < Accnt .
  op SavAccnt : -> SavAccnt .
  op rate :_ : MachineInt -> Attribute .
endm
```

3.3 Structured Specifications and Extensions of `META-LEVEL`

As explained for Core Maude in Section 2.3, in Full Maude we can use keywords `protecting` and `including`, or `pr` and `inc` in abbreviated form, to define structured specifications. All the predefined modules introduced in Section 2.4, plus the module `META-LEVEL`, are also available in Full Maude[5]. As we will explain in Section 3.5, Full Maude supports not only the importation of modules, but the importation of module expressions as well.

In metalevel computations it is very convenient to be able to refer by name to the meta-representations of modules already entered into the system. To make this possible, Full Maude allows importation declarations of the form

protecting META-LEVEL[$Id_1, \ldots, Id_n$] .

where $Id_1, \ldots, Id_n$ is a list of names of user-defined modules. With this declaration, new constants $Id_1, \ldots, Id_n$ of sort `Module` are declared, and equations making each constant Id_i equal to the metalevel representation of the module with name Id_i declared previously by the user, for $i = 1 \ldots n$, are added. Thus, we can first enter the module.

```
(fmod NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm id: 0] .
  var N M : Nat .
  eq s N + s M = s s (N + M) .
 endfm)
```

and then we can declare a module that protects `META-LEVEL[NAT]` and defines a function to extract the set of operator declarations of a functional module as follows.

```
(fmod META-NAT is
  protecting META-LEVEL[NAT] .

  op getOpDeclSet : FModule -> OpDeclSet .

  var QI : Qid .
  var IL : ImportList .
  var SD : SortDecl .
  var SSDS : SubsortDeclSet .
  var ODS : OpDeclSet .
  var VDS : VarDeclSet .
  var MAS : MembAxSet .
  var EqS : EquationSet .

  eq getOpDeclSet(fmod QI is IL SD SSDS ODS VDS MAS EqS endfm)
    = ODS .
 endfm)
```

Then we can apply this function to the constant `NAT`, which in `META-NAT` has been declared to be equal to the meta-representation of the above module `NAT`, as follows.

```
Maude> (red getOpDeclSet(NAT) .)
Result OpDeclSet :
   op '0 : nil -> 'Nat [none] .
   op '_+_ : 'Nat 'Nat -> 'Nat [assoc comm id({'0}'Nat)] .
   op 's_ : 'Nat -> 'Nat [none] .
```

[5]The built-in module `LOOP-MODE` presented in Section 2.8 is not supported in Full Maude.

We can also use the descent functions as discussed in Section 2.5.5.

```
Maude> (red meta-reduce(NAT, '_+_[{'0}'Nat, 's_[{'0}'Nat]]) .)
Result Term : 's_[{'0}'Nat]
```

Note that we have written the actual meta-representation of the term `0 + s 0` instead of using the more intuitive notation $\overline{\texttt{0 + s 0}}$ used in Section 2.5.5. However, in Full Maude, we can use the `up` function to avoid the cumbersome task of explicitly writing the meta-representation of a term or the meta-representation of a module. For example, to obtain the meta-representation of a term as `s 0` in the module `NAT`, which we denote by $\overline{\texttt{s 0}}$, we can write

```
Maude> (red up(NAT, s 0) .)
Result Term : 's_[{'0}'Nat]
```

Thus, instead of explicitly writing the meta-representation $\overline{\texttt{0 + s 0}}$ in the above reduction we can write

```
Maude> (red meta-reduce(NAT, up(NAT, 0 + s 0)) .)
Result Term : 's_[{'0}'Nat]
```

Note that the module name is the first argument of the `up` function, with the term of that module to be meta-represented as the second argument. Since the same term can be parsed in different ways in different modules, and therefore can have different meta-representations depending on the module in which it is considered, the module to which the term belongs has to be used to obtain the correct meta-representation. Note also that the above reduction only makes sense at the metalevel, that is, in a module importing the module `META-LEVEL`.

The `up` function also gives us a second way of accessing the meta-representation of any module in the database. Evaluating in any module importing the module `META-LEVEL` the `up` function with the name of any module in the database as argument we obtain the meta-representation of such a module. Thus, assuming that the previous module `NAT` has been entered in Full Maude, and therefore is in the database, we can get its meta-representation, which we denote by $\overline{\texttt{NAT}}$, as follows.

```
Maude> (red up(NAT) .)
Result FModule :
    fmod 'NAT is
       nil
       sorts 'Nat .
       none
       op '0 : nil -> 'Nat [none] .
       op '_+_ : 'Nat 'Nat -> 'Nat [assoc comm id({'0}'Nat)] .
       op 's_ : 'Nat -> 'Nat [none] .
       var 'M : 'Nat .
       var 'N : 'Nat .
       none
       eq '_+_['s_['N], 's_['M]] = 's_['s_['_+_['N, 'M]]].
    endfm
```

This facility can be used to write reductions of terms as those presented in Section 2.5.5, for example of `meta-reduce(`$\overline{\texttt{NAT}}$`, `$\overline{\texttt{s s 0 + s s s 0}}$`)`, as follows.

```
Maude> (red meta-reduce(up(NAT), up(NAT, s s 0 + s s s 0)) .)
Result Term : 's_['s_['s_['s_['s_[{'0}'Nat]]]]]
```

The result of a metalevel computation that may use several levels of reflection can be a term or module meta-represented one or more times, which may be hard to read. Therefore, to display the output in a more readable form we can use the `down` command, which is in a sense inverse to `up`, since it gives us back the term from its meta-representation. The `down` command takes two arguments. The first argument is the name of the module to which the term to be returned belongs. The meta-representation of the desired output term should be the result of the command given as second argument. The syntax of the `down` command is as follows.

```
op down_:_ : ModExp PreCommand -> PreCommand .
```

Thus, we can give the following command.

```
Maude> (down NAT :
           red-in META-NAT :
              meta-reduce(NAT, up(NAT, 0 + s 0)) .)
Result Nat : s 0
```

Notice that this is equivalent to what we wrote in section 2.5.5 as

```
Maude> red meta-reduce(NAT, s 0 + 0) .
result Term: s 0
```

The use of `up` and `down` can be iterated with as many levels of reflection as we wish. For example, in a module

```
(fmod META-META-NAT is
   protecting META-LEVEL[META-NAT] .
 endfm)
```

we can give the command

```
Maude> (down NAT :
           down META-NAT :
               red meta-reduce(META-NAT,
                       up(META-NAT,
                          meta-reduce(NAT,
                              up(NAT, 0 + s 0)))) .)
Result Nat : s 0
```

This is equivalent to what we would have written using the overline notation as

```
Maude> red meta-reduce(META-NAT, meta-reduce(NAT, s 0 + 0)) .
result Term: s 0
```

3.4 Commands and the Module Database

As with modules, all commands at the Full Maude level should be entered enclosed in parentheses. In this way the system can distinguish between commands at the Core Maude level—that in this context are "system programming" commands in the module `FULL-MAUDE`—and commands to be handled by Full Maude.

The `reduce` and `rewrite` commands have the same effect in Full Maude specifications as their homonymous commands in Core Maude.

The syntax for the `reduce` commands is given by the following declarations. Notice that, as in Core Maude, `red` is used as an abbreviation for `reduce`. However, the command to reduce a term in a given module has a somewhat different syntax than the one used in Core Maude[6].

```
op red_. : Bubble -> PreCommand .
op red-in_:_. : ModExp Bubble -> PreCommand .
```

Similarly, the following are the declarations defining the syntax of the rewrite commands.

```
op rew_. : Bubble -> PreCommand .
op rew[_]_. : Token Bubble -> PreCommand .
op rew-in_:_. : ModExp Bubble -> PreCommand .
op rew-in[_]_:_. : Token ModExp Bubble -> PreCommand .
```

Full Maude maintains a database with all the modules that have been introduced since the begining of the session. Notice that a Full Maude session does not start automatically when we start the system. The Maude specification of Full Maude has to be loaded first, and the loop has to be initialized. Apart from the built-in modules, Core Maude and Full Maude keep independent module stores.

As in Core Maude, when a module is not explicitly specified the system uses a module by default, that in general is the last module introduced, although one can select another module from the database with the `select` command.

```
op select_. : ModExp -> PreCommand .
```

There are also several `show` commands as in Core Maude. There are commands to show a module or theory[7] as introduced by the user, to show the flattened version of any module in the database, and to show some of the components in a module in the same way as the commands for Core Maude, as explained in Section A. For any of these commands the name of a module can be given. If no name is specified, the default current module is used. The syntax of these commands is as follows.

```
op show module . : -> PreCommand .
op show module_. : ModExp -> PreCommand .
op show all . : -> PreCommand .
op show all_. : ModExp -> PreCommand .
op show sorts . : -> PreCommand .
op show sorts_. : ModExp -> PreCommand .
op show ops . : -> PreCommand .
op show ops_. : ModExp -> PreCommand .
op show vars . : -> PreCommand .
op show vars_. : ModExp -> PreCommand .
op show mbs . : -> PreCommand .
```

[6] To have, for example, the command `red in_:_.` we would need the operator declaration

```
op red in_:_. : ModExp Bubble -> PreCommand .
```

However, given a command like `red in NAT : s 0 .` there would be two possible parses: `red in` $\boxed{\texttt{NAT}}$ `:` $\boxed{\texttt{s 0}}$ `.` and `red` $\boxed{\texttt{in NAT : s 0}}$ `..` Both of them are correct parses, but the `meta-parse` function returns only one of them. In case of ambiguity, one of the possible parses is arbitrarily chosen, preventing us from the possibility of taking the right one. This syntactic limitation as well as those discussed in Section 3.6 will be overcome in a future version.

[7] Theories are discussed in Section 3.5.1.

```
op show mbs_. : ModExp -> PreCommand .
op show eqns . : -> PreCommand .
op show eqns_. : ModExp -> PreCommand .
op show rls . : -> PreCommand .
op show rls_. : ModExp -> PreCommand .
```

The `show view_` command prints the view[8] with the specified name.

```
op show view_. : ViewExp -> PreCommand .
```

The `show modules` and `show views` commands print, respectively, the list of the names of all modules, and the list of the names of all views, present in the database.

```
op show modules . : -> PreCommand .
op show views . : -> PreCommand .
```

3.5 Parameterized Programming

Parameterized modules, theories and views are the basic building blocks of parameterized programming [7, 27]. As in OBJ, a theory defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter. The instantiation of the formal parameters of a parameterized module with actual parameter modules requires a *view* from the formal interface theory to the corresponding actual module. That is, views provide the interpretation of the actual parameters.

3.5.1 Theories

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Full Maude supports three different types of theories: functional theories, system theories, and object-oriented theories. Their structure is the same as that of their module counterparts. All of them can have sorts, subsort relationships, operators, variables, membership assertions and equations, and can import other theories or modules. System theories can have rules as well, and object-oriented theories can have classes, subclass relationships and messages.

Theories are rewriting logic theories with a *loose interpretation*. Theories are then allowed to contain rules and equations with variables in their righthand sides or conditions that may not appear in their corresponding lefthand sides. Similarly, conditional membership axioms may have variables in their conditions that do not appear in their membership assertions. Also, the lefthand side may be a single variable. In the current version, theories are not executed and cannot be parameterized.

Functional theories are declared with the keywords `fth ... endfth`, system theories with the keywords `th ... endth`, and object-oriented theories with the keywords `oth ... endoth`. The syntax for the declaration of theories is as follows.

```
op fth_is_endfth : ModuleName FDeclList -> PreModule .
op th_is_endth : ModuleName SDeclList -> PreModule .
op oth_is_endoth : ModuleName ODeclList -> PreModule .
```

[8] Views are discussed in Section 3.5.3.

Let us begin by introducing the functional theory `TRIV`, which requires just a sort.

```
(fth TRIV is
  sort Elt .
 endfth)
```

The theory of partially ordered sets with an anti-reflexive and transitive binary operator can be expressed in the following way.

```
(fth POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false .
  ceq X < Z = true if X < Y and Y < Z .
 endfth)
```

The theory of totally ordered sets, that is, posets in which all pairs of distinct elements have to be related, can be given as follows.

```
(fth TOSET is
  including POSET .
  vars X Y : Elt .
  eq X < Y or Y < X or X == Y = true .
 endfth)
```

The `including` importation of a theory into another theory keeps its loose semantics. However, if the imported theory contains a module, which therefore must be interpreted with an initial semantics[9], then that initial semantics is maintained by the importation. For example, in the definition of the `POSET` theory, the declaration `protecting BOOL` ensures that the initial semantics of the functional module for the Booleans is preserved, which is in fact a crucial requirement[10]. This requirement is then preserved by `TOSET` when `POSET` is included. In fact, we are dealing with a structure in which part of it, not only the top theory, has a loose semantics, while other parts contain modules with an initial semantics. The kind of semantics of a module or theory is determined by the keyword used in its definition and the importation mode.

As an example of a system theory, let us consider the theory `CHOICE` of multisets of elements with a choice operator defined on the multisets by a rewrite rule that nondeterministically picks up one of the elements in the multiset. We can express this theory as indicated below, where we have a sort `MSet` declared as a supersort of the sort `Elt`.

```
(th CHOICE is
  sort MSet Elt .
  subsort Elt < MSet .
  op __ : Elt Elt -> Elt [assoc comm] .
  var E : Elt .
  var L : MSet .
```

[9] In Full Maude, the importation of a module into a theory is supported only in protecting mode.

[10] Note that a declaration importing `BOOL` is added to all modules and theories. There is no way in the current version of Full Maude of setting off this inclusion. In Core Maude it can be done with the `set include` command.

```
  rl [choice] : E L => E .
 endth)
```

Our last example is an object-oriented theory, namely, the theory of classes with at least one attribute of any sort. It is defined as follows.

```
(oth CELL is
  sort Elt .
  class Cell | contents : Elt .
 endoth)
```

This last theory could have been more naturally expressed as a parameterized theory. We could have defined `CELL` with a parameter `TRIV` to capture the idea of defining cells in a generic way. However, the present Full Maude implementation does not support parameterized theories. We plan to extend the language to support not only parameterized theories, but also *parameterized views*.

3.5.2 Parameterized Modules

Theories can be used to declare the interface requirements for parameterized modules. Modules can be parameterized by one or more theories. All theories appearing in the interface have to be labeled in such a way that their sorts can be uniquely identified. The general form for the interface of a parameterized module is $[X_1 :: T_1 , \ldots , X_n :: T_n]$, where $X_1 \ldots X_n$ are the labels and $T_1 \ldots T_n$ are the names of the parameter theories. Thus, the syntax of the interface of parameterized modules is given by the following declarations.

```
op _::_ : Token ModExp -> Parameter [prec 40 gather (e &)] .

subsort Parameter < ParameterList .
op _,_ : ParameterList ParameterList -> ParameterList [assoc] .

op _[_] : Token ParameterList -> ModuleName .
```

In the current version of Full Maude all the sorts coming from theories in the interface must be qualified by their labels, even if there is no ambiguity. If Z is the label of a parameter theory T, then each sort S in T has to be qualified as $S.Z$. Since, as we will see in Section 3.5.3, operator maps affect entire families of subsort-overloaded operators, there cannot be subsort overloading between an operator declared in a theory being used as parameter of a parameterized module and an operator declared in the body of the parameterized module, or between operators declared in two parameter theories of the same module. Thus, the parameterized module `SIMPLE-SET`, with `TRIV` as interface can be defined as follows.

```
(fmod SIMPLE-SET[X :: TRIV] is
  sorts Set NeSet .
  subsorts Elt.X < NeSet < Set .
  op mt : -> Set .
  op __ : Set Set -> Set [assoc comm id: mt] .
  op __ : NeSet NeSet -> NeSet [assoc comm id: mt] .
  var E : Elt.X .
  eq E E = E .
 endfm)
```

Note that, as discussed in Section 3.3, in Maude—unlike OBJ3—sorts are not systematically qualified by their module name. In the case of OBJ3, importing, for example, sets or lists of different elements introduces repeated sorts `Set` or `List` and operators that must be qualified by the names of the submodules they come from, that is, by module expressions often of considerable length. Of course, in OBJ3 it is possible to rename all these items. But this means that, to avoid the burden of long qualifications by module expressions, we have to include explicitly many more renamings than we would like.

The convention of not qualifying sorts may be particularly weak when dealing with parameterized modules. However, given that Maude supports ad-hoc overloading and that constants can be qualified in order to be disambiguated, the problem of ambiguity in a signature is reduced to collisions of sorts. Our proposal consists in qualifying parameterized sorts, not with the module expression they belong to, but with the name of the view or views used in the instantiation of the parameterized module. In the current version of Full Maude, we assume that all views are named, and that these names are the ones used in the qualification. Specifically, in the body of a parameterized module M`[`X_1 `::` T_1`,` ... `,` X_n `::` T_n`]`, any sort S can be written in the form S`[`X_1`,` ... `,` X_n`]`. When the module is instantiated with views $V_1, \ldots, V_n$ then this sort becomes S`[`V_1`,` ... `,` V_n`]`. Note that the parameterization of sorts is optional. The above specification, for example, is perfectly valid.

The declarations needed to allow parameterized sorts are the following.

```
subsort ViewToken < ViewExp .
op _,_ : ViewExp ViewExp -> ViewExp [assoc] .

op _[_] : Sort ViewExp -> Sort [prec 40] .
```

Thus, the previous module to define sets could instead have been defined as follows.

```
(fmod SET[X :: TRIV] is
   sorts Set[X] NeSet[X] .
   subsorts Elt.X < NeSet[X] < Set[X] .
   op mt : -> Set[X] .
   op __ : Set[X] Set[X] -> Set[X] [assoc comm id: mt] .
   op __ : NeSet[X] NeSet[X] -> NeSet[X] [assoc comm id: mt] .
   var E : Elt.X .
   eq E E = E .
  endfm)
```

In the coming sections we will see how this qualification convention for the sorts of a parameterized module avoids many unintended collisions of sort names, thus making renaming practically unnecessary.

The module `SET` has only one parameter. In general, however, parameterized modules can have several parameters. It can furthermore happen that several parameters are declared with the same parameter theory, that is, we can have an interface of the form `[X :: TRIV, Y :: TRIV]` involving the theory `TRIV`. Therefore, parameters cannot be treated as normal submodules, since we do not want them to be shared when their labels are different. We regard the relationship between the body of a parameterized module and the interface of its parameters not as an inclusion, but as a module constructor which is evaluated generating renamed copies of the parameters, which are then included. For the above interface, two copies of the theory `TRIV` are generated, with names

`X :: TRIV` and `Y :: TRIV`. In such copies of parameter theories sorts are renamed as follows: If Z is the label of a parameter theory T, then each sort S in T (for `TRIV` just the sort `Elt`) is renamed to S.Z. This is the reason why all occurrences of these sorts in the parameterized module must mention their corresponding renaming. In a future version of the system, this qualification will be necessary only in case of ambiguity.

Let us consider as an example the following module `PAIR`. Notice the use of the qualifications for the sorts coming from each of the parameters, and notice also the qualification of the sort `Pair[X, Y]`.

```
(fmod PAIR[X :: TRIV, Y :: TRIV] is
  sort Pair[X, Y] .
  op <_;_> : Elt.X Elt.Y -> Pair[X, Y] .
  op 1st : Pair[X, Y] -> Elt.X .
  op 2nd : Pair[X, Y] -> Elt.Y .
  var A : Elt.X .
  var B : Elt.Y .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
 endfm)
```

If a parameter theory is structured, this renaming process for parameter theories is carried out not only at the top level, but for the whole "theory part," that is, not renaming modules. Consider, for example, the following parameterized module defining a lexicographical ordering on pairs of elements of a totally ordered set.

```
(fmod TOSET-PAIR[X :: TOSET, Y :: TOSET] is
  sort Pair[X, Y] .
  op <_;_> : Elt.X Elt.Y -> Pair[X, Y] .
  op _<_ : Pair[X, Y] Pair[X, Y] -> Bool .
  op 1st : Pair[X, Y] -> Elt.X .
  op 2nd : Pair[X, Y] -> Elt.Y .
  var A A' : Elt.X .
  var B B' : Elt.Y .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
  eq < A ; B > < < A' ; B' >
    = (A < A') or (A == A' and B < B') .
 endfm)
```

Representing by $\hookrightarrow$ the inclusion relations between modules and theories, and by $\Rightarrow$ the initiality constraints, we can depict the resulting structure as follows.

$$
\begin{tikzcd}
 & & \mathtt{TOSET\text{-}PAIR} & & \\
 & & \mathtt{X::TOSET} + \mathtt{Y::TOSET} \arrow[u, Rrightarrow] & & \\
\mathtt{X::TOSET} \arrow[urr, hook] & & & & \mathtt{Y::TOSET} \arrow[ull, hook'] \\
\mathtt{X::POSET} \arrow[u, hook] & & & & \mathtt{Y::POSET} \arrow[u, hook'] \\
 & & \mathtt{BOOL} \arrow[ull, hook'] \arrow[urr, hook] & & \\
 & & \emptyset \arrow[u, Rrightarrow] & &
\end{tikzcd}
$$

where we have two copies not only of `TOSET` but also of the `POSET` subtheory.

An object-oriented parameterized module defining a stack of elements can be defined as follows. We define a class `Stack[X]`[11] as a linked sequence of node objects. Objects of class `Stack[X]` only have an attribute `first`, containing the identifier of the first node in the stack. If the stack is empty the value of the `first` attribute is `null`. Each object of class `Node[X]` has an attribute `next` holding the identifier of the next node—which will be `null` if there is no next node—and an attribute `contents` to store a value of sort `Elt.X`. Notice that the identifiers of the nodes are of the form `o(`S`,`N`)`, where S is the identifier of the stack object to which the node belongs, and N is a natural number. The messages `push`, `pop` and `top` have as their first argument the identifier of the object to which they are addressed, and will cause, respectively, the insertion at the top of the stack of a new element, the deletion of the top element, and the sending of a response message `elt` containing the element at the top of the stack to the object making the request.

```
(omod STACK[X :: TRIV] is
  protecting MACHINE-INT .
  protecting QID .
  subsort Qid < Oid .
  class Node[X] | next : Oid, contents : Elt.X .
  class Stack[X] | first : Oid .
  msg _push_ : Oid Elt.X -> Msg .
  msg _pop : Oid -> Msg .
  msg _top_ : Oid Oid -> Msg .
  msg _elt_ : Oid Elt.X -> Msg .

  op null : -> Oid .
  op o : Oid MachineInt -> Oid .
```

[11]Notice that naming of parameterized classes follows the same conventions discussed above for parameterized sorts.

```
    vars O O' O'' : Oid .
    var E : Elt.X .
    var N : MachineInt .

    rl [top] : *** top on a nonempty stack
       < O : Stack[X] | first : O' >
       < O' : Node[X] | contents : E >
       (O top O'')
       => < O : Stack[X] | >
          < O' : Node[X] | >
          (O'' elt E) .

    rl [push1] : *** push on a nonempty stack
       < O : Stack[X] | first : o(O, N) >
       (O push E)
       => < O : Stack[X] | first : o(O, N + 1) >
          < o(O, N + 1) : Node[X] |
                contents : E, next : o(O, N) > .
    rl [push2] : *** push on an empty stack
       < O : Stack[X] | first : null >
       (O push E)
       => < O : Stack[X] | first : o(O, 0) >
          < o(O, 0) : Node[X] | contents : E, next : null > .

    rl [pop] : *** pop on a nonempty stack
       < O : Stack[X] | first : O' >
       < O' : Node[X] | next : O'' >
       (O pop)
       => < O : Stack[X] | first : O'' > .
   endom)
```

We may want to define stacks not storing data elements of a particular sort, but actually objects in a particular class. We can define an object-oriented module with the intended behavior as a parameterized module `STACK2` parameterized by the object-oriented theory `CELL`, presented in Section 3.5.1, as follows.

```
  (omod STACK2[X :: CELL] is
    protecting MACHINE-INT .
    protecting QID .
    subsort Qid < Oid .
    class Node[X] | next : Oid, node : Oid .
    class Stack[X] | first : Oid .
    msg _push_ : Oid Oid -> Msg .
    msg _pop : Oid -> Msg .
    msg _top_ : Oid Oid -> Msg .
    msg _elt_ : Oid Elt.X -> Msg .

    op null : -> Oid .
    op o : Oid MachineInt -> Oid .

    vars O O' O'' O''' : Oid .
    var E : Elt.X .
    var N : MachineInt .
```

```
  rl [top] : *** top on a nonempty stack
     < O : Stack[X] | first : O' >
     < O' : Node[X] | node : O'' >
     < O'' : Cell.X | contents : E >
     (O top O''')
     => < O : Stack[X] | >
        < O' : Node[X] | >
        < O'' : Cell.X | >
        (O'' elt E) .

  rl [push1] : *** push on a nonempty stack
     < O : Stack[X] | first : o(O, N) >
     (O push O')
     => < O : Stack[X] | first : o(O, N + 1) >
        < o(O, N + 1) : Node[X] |
               next : o(O, N), node : O' > .
  rl [push2] : *** push on an empty stack
     < O : Stack[X] | first : null >
     (O push O')
     => < O : Stack[X] | first : o(O, 0) >
        < o(O, 0) : Node[X] | next : null, node : O' > .

  rl [pop] : *** pop on a nonempty stack
     < O : Stack[X] | first : O' >
     < O' : Node[X] | next : O'' >
     (O pop)
     => < O : Stack[X] | first : O'' > .
endom)
```

3.5.3 Views

We use views to assert how a particular target module or theory is claimed to satisfy[12] a source theory. In general, there may be several ways in which such requirements might be satisfied, if at all, by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target.

In the current version of Full Maude, default views are not supported. Therefore, all views have to be defined explicitly, and all of them must have a name. As any theory or module, views should have been defined before they are used.

In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort, function, class, and message in the source theory. Although the current version does not support default views in the style of OBJ3, "obvious" parts of a mapping do not need to be explicitly given, namely, any identical mapping of a function, message, or attribute f to itself such that its arity and coarity are mapped to those of an operator, message, or attribute with the same name in the target can be omitted. However, maps for all sorts in the source theory have to be given, even when they are identity maps.

[12]Each view declaration has an associated set of *proof obligations*, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques of the style supported for Maude's logic in [12].

The mapping of a sort S in the source theory to a sort S' in the target is expressed with syntax

`sort` S `to` S' `.`

The mapping of operators is expressed with syntax

`op` O `to` O' `.`

where O is an operator identifier or an operator identifier together with its arity and value sort. An operator map in which explicit arity and coarity are given affects, not only the operators with such arity and coarity, but the entire family of subsort-overloaded operators (see Section 2.1.1) associated to the given operator. The target operators can be derived operators, that is, they can be terms with variables. Therefore, we can map a function symbol, not only to another function symbol, but also to an expression. Consider, for example, the case in which we want to define a view from a theory in which we have a "less or equal" operator `_<=_`, defined with reflexivity, symmetry and transitivity equations, to a module in which such an operator does not exist but we have an operator "less than" `_<_`. Then, we can define a view with a map

```
op X <= Y to term X < Y or X == Y .
```

The mapping of a class C in the source theory to a class C' in the target is expressed with syntax

`class` C `to` C' `.`

Attribute maps have the form

`attr` A `.` C `to` A' `.`

where A is the name of an attribute of class C in the source theory and A' is an attribute of the image class of C under the view. The mapping of messages is expressed with syntax

`msg` M `to` M' `.`

where M is a message identifier or a message identifier together with its arity and value sort. As for operators, a message map in which explicit arity and coarity are given affects the entire family of subsort-overloaded message declarations associated to the declaration of the given message.

The syntax for views is given by the following declarations.

```
op op_to term_. : Bubble Bubble -> ViewDecl .
op op_to_. : Token Token -> ViewDecl .
op op_:_->_to_. : Token SortList Sort Token -> ViewDecl .
op op_: ->_to_. : Token Sort Token -> ViewDecl .
op sort_to_. : Sort Sort -> ViewDecl .
op class_to_. : Sort Sort -> ViewDecl .
op attr_._to_. : Token Sort Token -> ViewDecl .
op msg_to_. : Token Token -> ViewDecl .
op msg_:_->_to_. : Token SortList Sort Token -> ViewDecl .
op msg_: ->_to_. : Token Sort Token -> ViewDecl .

subsorts VarDecl < ViewDecl < ViewDeclList .
subsort VarDeclList < ViewDeclList .

op __ : ViewDeclList ViewDeclList -> ViewDeclList [assoc] .
```

```
op view_from_to_is_endv :
      ViewToken ModExp ModExp ViewDeclList -> PreView .
```

Thus, we can have a view

```
(view MachineInt from TRIV to MACHINE-INT is
   sort Elt to MachineInt .
 endv)
```

which defines a view from the theory TRIV to the module MACHINE-INT. In views from TRIV we encourage the convention of naming such views by the name of the *sort* to which Elt is mapped. Although it is not necessary to follow this convention, it can add understandability to the specifications. What has to be avoided is using the labels in interfaces of parameterized modules as names of views, since this can sometimes generate ambiguities.

We can have views between theories, which is particularly useful to compose instantiations of views to link the formal parameter of some parameterized module to some actual parameter via some intermediate formal parameter of another parameterized module. We will give some examples in the coming sections. An example of a view whose target is a theory is the following.

```
(view Toset from TRIV to TOSET is
   sort Elt to Elt .
 endv)
```

As already mentioned, in some cases it is useful to be able to express mappings of functions, not to other functions but to expressions. For example, the _<_ relation of a toset can be mapped to an expression using the "less than or equal" operator _<=_ and the inequality operator _=/=_ as follows.

```
(view MachIntAsToset from TOSET to MACHINE-INT is
   sort Elt to MachineInt .
   vars X Y : Elt .
   op X < Y to term X <= Y and X =/= Y .
 endv)
```

Notice that, when dealing with parameterized modules with structured theories as parameters, as in the TOSET-PAIR example discussed in Section 3.5.1, we have to give a view not only for the top theory but for the entire "loose part," that is, for all other subtheories imported by the theory. In the near future we plan to handle parameterized theories and views as well, as a way to give more structure to both theories and views.

3.5.4 Module Expressions

As in Clear [7], OBJ [27], and other specification languages in that tradition, the abstract syntax for writing specifications in Maude can be seen as given by *module expressions*, where the notion of module expression is understood as an expression that defines a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations, that is, according to a specific *module algebra*. In fact, structuring is essential in all specification languages, not only to facilitate the construction of specifications from already existing ones—with more or less flexible reusability mechanisms—but also for managing the complexity of understanding and analyzing large specifications.

A module importing some combination of modules, given by module expressions, can be seen as a structured module with more or less complex relationships among its component submodules. For execution purposes, however, we typically want to convert this structured module into an equivalent unstructured module, that is, into a "flattened" module without submodules. In the case of Maude, this flattened module will then be compiled into the rewrite engine. By systematically using the metaprogramming capabilities of Maude we can both evaluate module expressions into structured module hierarchies, and flatten such hierarchies into unstructured modules for execution. All such module operations are defined by rewrite rules that operate on the metalevel term representations of modules. This is essentially the idea behind the implementation of Full Maude in Maude.

The current version of Full Maude supports two types of module expression: instantiation of a module expression with a view expression, and renaming of a module expression with a set of mappings. The syntax used for both of them is the same one as in OBJ3, namely

```
op _[_] : ModExp ViewExp -> ModExp .
```

for instantiation of parameterized modules, and

```
op _*(_) : ModExp MapList -> ModExp .
```

for renamings. As we saw in Section 3.5.2, a view expression can be a single view name, or a sequence of view names separated by commas in case the module being instantiated has several parameters.

Module Instantiation

Instantiation is the process by which actual parameters are bound to the parameters of a parameterized module and a new module is created as a result. This can be seen in fact as the evaluation of a module expression. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the actual parameters to the corresponding sorts, operators (or expressions), etc. in the target.

The instantiation of a parameterized module has to be made with views explicitly defined previously. Thus, we can have a set of machine integers with the module expression `SET[MachineInt]`, or a pair of machine integers as tosets with `TOSET-PAIR[MachIntAsToset, MachIntAsToset]`.

As mentioned in Section 3.5.3, we can define views from theories to theories. Using such views we can, for example, instantiate the module `SET` with the view `Toset` given in the previous section. The result is a module `SET[Toset]` which is still parameterized, but now by the theory `TOSET`. We can instantiate it again with a view from `TOSET` to some other theory or module, for example, `MachIntAsToset`, obtaining the module `SET[Toset][MachIntAsToset]`, which is just a set of machine integers. For example, we can give a more concise definition of the parameterized module `TOSET-PAIR[X :: TOSET, Y :: TOSET]` using these ideas as follows.

```
(fmod TOSET-PAIR[X :: TOSET, Y :: TOSET] is
   protecting PAIR[Toset, Toset][X, Y] .
   op _<_ : Pair[Toset, Toset][X, Y] Pair[Toset, Toset][X, Y]
          -> Bool .
   var A A' : Elt.X .
```

```
  var B B' : Elt.Y .
  eq < A ; B > < < A' ; B' >
    = (A < A') or (A == A' and B < B') .
endfm)
```

Let us consider now the following module `MAX`, parameterized by the theory `TOSET`. Given a set of elements in this toset, the function `max` returns the maximum element in the set.

```
(fmod MAX[T :: TOSET] is
  protecting SET[Toset][T] .
  op max : NeSet[Toset][T] -> Elt.T .
  var E : Elt.T .
  var S : NeSet[Toset][T] .
  eq max(E) = E .
  ceq max(E S) = E if max(S) < E .
  ceq max(E S) = max(S) if not (max(S) < E) .
endfm)
```

Module expressions can be arguments of a `protecting` or `including` importation, or can be used as the module in which to reduce or rewrite by a `red` or `rew` command. In general we can use module expressions in any place where the name of a module is expected. In fact, in Full Maude the name of a module is given by a module expression, and each time a new module expression is entered, the module expression is evaluated, and a module is generated with this module expression as its name. It is this module which is passed to the engine to get the intended result.

Let us see how module expressions can be used in `red` commands using the instantiation of `MAX` with the view `MachIntAsToset` presented in Section 3.5.3.

```
Maude> (red-in MAX[MachIntAsToset] : max(5 4 8 4 6 5) .)
Result : 8
```

Notice that, if we have several parameters, we can instantiate the parameterized module with some views going to theories and others going to modules. The result in this case is the expected one, that is, we get a module parameterized by the targets of those views going to theories.

Module Renaming

A renaming can be considered as a function that, given a module M and a list of mappings S, returns a copy of the module in which the names of the sorts, operations, etc. are changed as indicated by the mappings. More precisely, given a structured specification, the renaming not only causes the creation of a copy of the top module in the structure, but renames also the part of the submodule structure that is affected by the renaming. For any other submodule M' in the structure which is affected by the mappings, a renamed copy of it is generated with name M' `*` (S'), where S' is the subset of mappings in S that affect M'.

The complete syntax for renaming maps is as follows.

```
op op_to_ : Token Token -> Map .
op op_:_->_to_ : Token SortList Sort Token -> Map .
op op_: ->_to_ : Token Sort Token -> Map .
op op_to_[_] : Token Token AttrList -> Map .
op op_:_->_to_[_] : Token SortList Sort Token AttrList -> Map .
```

```
op op_: ->_to_[_] : Token Sort Token AttrList -> Map .
op sort_to_ : Sort Sort -> Map .
op label_to_ : Token Token -> Map .
op class_to_ : Sort Sort -> Map .
op attr_._to_ : Token Sort Token -> Map .
op msg_to_ : Token Token -> Map .
op msg_:_->_to_ : Token SortList Sort Token -> Map .
op msg_: ->_to_ : Token Sort Token -> Map .

subsort Map < MapList .
op _,_ : MapList MapList -> MapList [assoc prec 42] .
```

Notice that we also allow the renaming of rule labels, which may be useful for metalevel applications.

A set of attributes can also be given in the renaming of an operator. This allows changing syntactic attributes, such as the precedence values and the gathering patterns, which may be of practical relevance when dealing with mixfix syntax. For example, when a change in the syntax of the operator could cause a parsing different from the intended one. Let us see an example in which modifying the grammatical attributes of an operator is useful. Suppose that we want to change the syntax of the function `max` in the module `MAX` presented above, to `maximum_`. We can do the following reduction.

```
Maude> (red-in MAX[MachIntAsToset]
                * (op max : NeSet[Toset][MachIntAsToset]
                        -> MachineInt to maximum_)
          : maximum 5 4 8 4 6 5 .)
Result : 4 5 6 8
```

This result may seem strange, but makes perfect sense. In fact the system indicates the term that it has reduced:

```
Reduce in MAX[MachIntAsToset]
           * (op max : NeSet[Toset][MachIntAsToset]
                 -> MachineInt to maximum_)
      : (maximum 5) 4 8 4 6 5 .
```

What has happened is that the precedence given by default to the operator with this new syntax is the same as that given to the operator `__`, and therefore, by the default gathering patterns, this is a valid parse. Notice that the other elements passed to the function `maximum` have "disappeared" by the equations in `SET`. We can obtain the intended result by placing parentheses around the set of numbers, but it is more convenient to change the precedence values of the attributes. We can, for example, raise the precedence of `maximum_`.

```
Maude> (red-in MAX[MachIntAsToset]
                * (op max : NeSet[Toset][MachIntAsToset]
                        -> MachineInt to maximum_
                     [prec 41])
          : maximum 5 4 8 4 6 5 .)
Reduce in MAX[MachIntAsToset]
              * (op max : NeSet[Toset][MachIntAsToset]
                      -> MachineInt to maximum_
                   [prec 41])
        : maximum (5 4 8 4 6 5) .
Result : 8
```

More examples can be found in Appendix E. We finish this section with an example involving object-oriented modules, namely, a stack of banking accounts.

```
(view Accnt from CELL to ACCNT is
  sort Elt to MachineInt .
  class Cell to Accnt .
  attr contents . Cell to bal .
 endv)
```

Now we can do the following rewriting.

```
Maude> (rew-in STACK2[Accnt] :
          < 'stack : Stack[Accnt] | first : null >
          < 'paul : Accnt | bal : 5000 >
          < 'peter : Accnt | bal : 2000 >
          < 'mary : Accnt | bal : 15000 >
          ('stack push 'paul)
          ('stack push 'peter)
          ('stack push 'mary)
          ('stack top 'peter)
          ('stack pop) .)
Result Configuration : ('peter elt 2000)
          < 'stack : Stack[Accnt] |
                  first : o('stack, 1) >
          < 'paul : Accnt | bal : 5000 >
          < 'peter : Accnt | bal : 2000 >
          < 'mary : Accnt | bal : 15000 >
          < o('stack, 0) : Node[Accnt] |
                  next : null, node : 'peter >
          < o('stack, 1) : Node[Accnt] |
                  next : o('stack, 0), node : 'mary >
```

3.6 Syntactic Restrictions and Caveats

We can write functional and system modules in Full Maude as we do it in Core Maude, but enclosing them in parentheses. However, there are some syntactic differences between what is currently allowed in Core Maude and in Full Maude. As a consequence, some syntactic restrictions should be taken into account when using Full Maude to write specifications:

1. Operator and message names have to be given in their equivalent *single identifier form* when they are declared, and

2. sort names used in term qualifications and in sort tests have to be in their equivalent *single identifier form*.

We plan to remove these syntactic restrictions in a future version. In the rest of the section we explain them in some detail and give some hints on how to avoid them.

Operator names have to be given as a single identifier. To declare multi-identifier operators they have to be given in their single identifier form, that is, each identifier in a multi-identifier name has to be preceded by a backquote. For example, to define an operator with name `_less than or equal_`, we have to use its single identifier form `_less`than`or`equal_`. Except for having to use the single identifier form in the operator name, the declaration of operations is exactly as for Core Maude. For example, the declaration of this operator on sort, say, `Int` is as follows.

```
op _less`than`or`equal_ : Int Int -> Bool .
```

Notice that not only blank spaces, but also the special characters '{', '}', '(', ')', '[', ']' and ',' break the identifiers. Therefore, to declare in Full Maude an operator such as {_} taking an element of sort, say, `Int` and with value sort `Set`, we should write

```
op `{_`} : Int -> Set .
```

As in Core Maude, several operators with the same arity and coarity can be defined in the same declaration using the keyword `ops`, but again, each operator name has to be given in its single identifier form. We could have for example the following declaration.

```
ops _`{_`} _`,_ : Foo Bar -> Baz .
```

Notice that, since each operator name is a single identifier, parentheses are not needed to indicate the boundaries between the syntactic forms of the different operators.

As for operator names, message names can be mixfix, but they have to be declared in single identifier form. Thus, to define a message `credit` with syntax, say, `(_)credit_` the declaration has to be given as follows.

```
msg `(_`)credit_ : Oid MachineInt -> Msg .
```

And the same applies to declarations of multiple message names:

```
msgs `(_`)credit_ `(_`)debit_ : Oid MachineInt -> Msg .
```

The last problem mentioned at the beginning of this section has to do with the qualification of terms by sort names and with sort tests. Since qualifications by sort and sort tests—as well as parentheses, polymorphism, and other syntactic features in the extended signature of a specification–are directly handled by Core Maude, and Core Maude does not know about parameterized sorts, the user is forced to use in these cases the names of parameterized sorts, not as he or she has defined them, but in their equivalent single identifier form. Thus, if we have, for example, a sort `List[Nat]` and a constant `nil` in it, if necessary, it should be qualified as ``(nil).List`[Nat`]``. Similarly, to check whether a term `T` has the sort `List[Nat]` we have to write ``T : List`[Nat`]`` or ``T :: List`[Nat`]``.

We plan to add to the Maude system new functionality supporting a more flexible treatment of the syntax of Full Maude and other languages, so that these syntactic restrictions will eventually be removed.

Chapter 4

The Semantics of Maude

We summarize the semantic foundations of Maude's functional, object-oriented, and system modules, including a brief discussion of parameterized modules. We first introduce the basic concepts of *membership equational logic*, whose initial algebras provide the mathematical semantics for *functional* modules. Then, we review the basic concepts of *rewriting logic*, whose initial models provide the mathematical semantics for *object-oriented* and *system* modules.

4.1 Membership Equational Logic and Functional Modules

Maude is a declarative language based on rewriting logic. But rewriting logic has its underlying equational logic as a parameter. There are, for example, unsorted, many-sorted, and order-sorted versions of rewriting logic, each containing the previous version as a special case. The underlying equational logic chosen for Maude is *membership equational logic* [41, 5], a conservative extension of both order-sorted equational logic and partial equational logic with existence equations [41]. It supports partiality, subsort relations, operator overloading, and error specification.

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, (K, Σ) a many-sorted (although it is better to say "many-kinded") signature, and $S = \{S_k\}_{k \in K}$ a K-kinded set of *sorts*.

An Ω-*algebra* is then a (K, Σ)-algebra A together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or nonerror, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulae, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ \ t = t' \ \ \mathit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *membership axioms* of the form

$$(\forall X)\ \ t : s \ \ \mathit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic has all the usual good properties: soundness and completeness of appropriate rules of deduction, initial and free algebras, relatively free algebras along theory morphisms, and so on [41].

In Maude, *functional modules* are equational theories in membership equational logic satisfying the additional requirement of being Church-Rosser and (preferably) terminating. *Functional theories* are also membership equational logic theories, but they do not need to be Church-Rosser; they have a *loose* interpretation, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model[1].

The semantics of an unparameterized functional module is the *initial algebra* specified by its theory. The semantics of a *parameterized* functional module is the free functor associated to the inclusion of the parameter theory[2] into the body of the parameterized module [41]. For example, a parameterized list module `LIST[X :: TRIV]` forms lists of models of the trivial parameter theory `TRIV` with one sort `Elt`, whose models are sets of elements and its semantics is the functor sending each set to the algebra of lists of the set. Similarly, a sorting module `SORTING[Y :: POSET]` sorts lists whose elements belong to a model of the `POSET` functional theory, that is, the data type of elements must have a partial order and its semantics is the functor sending each poset to the algebra of lists for that poset with a sorting function[3]. All this is entirely similar to the semantics of "objects" (that correspond to modules in our sense) and theories in OBJ [27]. Indeed, since membership equational logic conservatively extends order-sorted equational logic, Maude's functional modules extend OBJ modules.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. There is no need to declare kinds explicitly. The convenience of order-sorted notation is retained as syntactic sugar. Thus, an operator declaration

```
op push : Nat Stack -> NeStack .
```

is understood as syntactic sugar for the membership axiom

$$(\forall x, y)\ \ push(x, y) : NeStack\ \ if\ \ x : Nat \wedge y : Stack.$$

Similarly, a subsort declaration `NeStack < Stack` corresponds to the membership axiom

$$(\forall x)\ \ x : Stack\ \ if\ \ x : NeStack.$$

Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. Therefore, the equations must satisfy the additional requirements of being Church-Rosser, terminating, and sort-decreasing [5]. This guarantees that all terms in an equivalence class modulo the equations will rewrite to a unique canonical form, and that this canonical form can be assigned a sort that is smaller than all other sorts assignable to terms in the class. For a module satisfying such conditions any reduction strategy will reach a normal form; nevertheless, as explained in Section 2.1.3, the user can assign to each operator a functional evaluation strategy in the OBJ style [27] to control the reduction for efficiency purposes. If no such strategies are declared,

[1] However, a functional theory may contain functional submodules in `protecting` mode, imposing the additional requirement that those submodules should be interpreted initially.

[2] Of course, if the parameterized module has several parameter theories, we should form their colimit, and consider instead the inclusion of such a colimit into the body.

[3] Note that `POSET` is a good example of a theory where part of the semantics is loose and part of it initial, because it protects the functional module `BOOL`, which is used in an essential way to define the partial order predicate.

a bottom-up strategy is chosen. Since Maude supports rewriting modulo equational theories such as associativity or associativity/commutativity, all that we say has to be understood for equational rewriting *modulo* such axioms.

In membership equational logic the Church-Rosser property of terminating and sort-decreasing equations is indeed equivalent to the confluence of their critical pairs [5]. Furthermore, both equality and membership of a term in a sort are then *decidable* properties [5]. That is, the equality and membership predicates are *computable functions*. We can then use the metatheorem of Bergstra and Tucker [1] to conclude that such predicates are themselves specifiable by Church-Rosser and terminating equations as Boolean-valued functions. This has the pleasant consequence of allowing us to include inequalities $t \neq t'$ and negations of sort tests $not(t : s)$ in conditions of equations and of membership axioms, since such seemingly negative predicates can also be axiomatized *inside the logic* in a positive way, provided that we have a subspecification of (not necessarily free) constructors in which to do it, and that the specification is indeed Church-Rosser, terminating, and sort decreasing. Of course, in practice they do *not* have to be explicitly axiomatized, since they are built into the implementation of rewriting deduction in a much more efficient way (see Section 2.4.1).

Let us denote membership equational logic by *MEqtl* and its associated rewriting logic by *MRWLogic*. Regarding an equational theory as a rewrite theory whose sets of rules are empty defines a conservative map of logics [32]

$$\mathit{MEqtl} \longrightarrow \mathit{MRWLogic}.$$

This is the way in which Maude's functional modules are regarded as a special case of its more general system modules.

4.2 Rewriting Logic

We first define rewrite theories and give the logic's rules of deduction. Then, the models of rewrite theories, including initial and free models, are discussed.

4.2.1 Theories and Deduction

A *signature* in rewriting logic is an equational theory[4] (Σ, E), where Σ is an equational signature and E is a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [17] and can be used to implement rewriting modulo many equational theories of interest. This is precisely what Maude does, using the equational attributes given in operator declarations—such as associativity, commutativity, identity, and idempotency—to rewrite modulo such axioms.

[4] Rewriting logic is parameterized by the choice of its underlying equational logic, that can be unsorted, many-sorted, order-sorted, membership equational logic, and so on. For Maude, the underlying equational logic is of course membership equational logic. However, to ease the exposition we give here an *unsorted* presentation.

Given a signature (Σ, E), *sentences* of rewriting logic are sequents of the form

$$[t]_E \longrightarrow [t']_E,$$

where t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of pairs $R \subseteq L \times T_{\Sigma,E}(X)^2$ whose first component is a label and whose second component is a pair of E-equivalence classes of terms, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called *rewrite rules*.[5] We understand a rule $(r, ([t], [t']))$ as a labeled sequent and use for it the notation $r : [t] \longrightarrow [t']$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sentence $[t] \longrightarrow [t']$, or that $[t] \longrightarrow [t']$ is a *(concurrent) $\mathcal{R}$-rewrite*, and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction* (where we assume that all the terms are well formed and $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t):

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

2. **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,
$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

3. **Replacement**. For each rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,
$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

4. **Transitivity**
$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

Rewriting logic is a logic for reasoning correctly about *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The rewrite rules in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Thus, computationally, each rewriting step is a parallel local transition in a concurrent system.

[5] To simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas presented here have been extended to conditional rules in [37] with very general rules of the form

$$r : [t] \longrightarrow [t'] \;\; \textit{if} \;\; [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k].$$

This increases considerably the expressive power of rewrite theories.

Alternatively, however, we can adopt a logical viewpoint instead, and regard the rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system.

The computational and the logical viewpoints under which rewriting logic can be interpreted can be summarized in the following diagram of correspondences:

State	$\leftrightarrow$	*Term*	$\leftrightarrow$	*Proposition*
Transition	$\leftrightarrow$	*Rewriting*	$\leftrightarrow$	*Deduction*
Distributed Structure	$\leftrightarrow$	*Algebraic Structure*	$\leftrightarrow$	*Propositional Structure*

The last row of equivalences is actually quite important. Roughly speaking, it expresses the fact that a state can be transformed in a concurrent way only if it is nonatomic, that is, if it is *composed* out of smaller state components that can be changed independently. In rewriting logic this composition of a concurrent state is formalized by the *operations* of the signature Σ of the rewrite theory $\mathcal{R}$ that axiomatizes the system. From the logical point of view such operations can naturally be regarded as user-definable *propositional connectives* stating the particular structure that a given state has. A subtle additional point about the last row of equivalences is that the algebraic structure of a system also involves *equations*. Such equations describe the system's global state as a *concurrent data structure*; they can have a dramatic impact on the amount of concurrency available in a system.

Note that it follows from this discussion that rewriting logic is primarily a logic *of* change—in which the deduction directly corresponds to the change—as opposed to a logic to talk *about* change in a more indirect and global manner such as the different variants of modal and temporal logic.

4.2.2 Models

We first sketch the construction of initial and free models for a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$. Such models capture nicely the intuitive idea of a "rewrite system" in the sense that they are systems whose states are E-equivalence classes of terms, and whose transitions are concurrent rewritings using the rules in R. By adopting a logical instead of a computational perspective, we can alternatively view such models as "logical systems" in which formulas are validly rewritten to other formulas by concurrent rewritings which correspond to proofs for the logic in question. Such models have a natural *category* structure, with states (or formulas) as objects, transitions (or proofs) as morphisms, and sequential composition as morphism composition, and in them dynamic behavior exactly corresponds to deduction.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, for which we assume that different labels in L name different rules in R, the model that we are seeking is a category $\mathcal{T}_{\mathcal{R}}(X)$ whose objects are equivalence classes of terms $[t] \in T_{\Sigma,E}(X)$ and whose morphisms are equivalence classes of "proof terms" representing proofs in rewriting deduction, i.e., concurrent $\mathcal{R}$-rewrites. The rules for generating such proof terms, with the specification of their respective domains and codomains, are given below; they just "decorate" with proof terms the rules 1–4 of rewriting logic. Note that we always use "diagrammatic" notation for morphism composition, i.e., $\alpha;\beta$ always means the composition of α *followed by* β.

1. **Identities**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] : [t] \longrightarrow [t]}$.

2. **Σ-structure**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t_1'] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t_n']}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \longrightarrow [f(t_1', \ldots, t_n')]}.$$

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

$$\frac{\alpha_1 : [w_1] \longrightarrow [w_1'] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w_n']}{r(\alpha_1, \ldots, \alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

4. **Composition** $\dfrac{\alpha : [t_1] \longrightarrow [t_2] \quad \beta : [t_2] \longrightarrow [t_3]}{\alpha ; \beta : [t_1] \longrightarrow [t_3]}.$

Each of the above rules of generation defines a different operation taking certain proof terms as arguments and returning a resulting proof term. In other words, proof terms form an algebraic structure $\mathcal{P}_{\mathcal{R}}(X)$ consisting of a graph with nodes $T_{\Sigma,E}(X)$, with identity arrows, and with operations f (for each $f \in \Sigma$), r (for each rewrite rule), and $\_ ; \_$ (for composing arrows). Our desired model $\mathcal{T}_{\mathcal{R}}(X)$ is the quotient of $\mathcal{P}_{\mathcal{R}}(X)$ modulo the following equations:[6]

1. **Category**
   (a) *Associativity*. For all α, β, γ, $\quad (\alpha;\beta);\gamma = \alpha;(\beta;\gamma)$.
   (b) *Identities*. For each $\alpha : [t] \longrightarrow [t']$, $\quad \alpha;[t'] = \alpha$ and $[t];\alpha = \alpha$.
2. **Functoriality of the Σ-algebraic structure**. For each $f \in \Sigma_n$,
   (a) *Preservation of composition*. For all $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$,
   $$f(\alpha_1;\beta_1, \ldots, \alpha_n;\beta_n) = f(\alpha_1, \ldots, \alpha_n); f(\beta_1, \ldots, \beta_n).$$
   (b) *Preservation of identities*. $\quad f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$.
3. **Axioms in E**. For $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ an axiom in E, for all $\alpha_1, \ldots, \alpha_n$, $\quad t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$.
4. **Exchange**. For each $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,
   $$\frac{\alpha_1 : [w_1] \longrightarrow [w_1'] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w_n']}{r(\overline{\alpha}) = r(\overline{[w]}); t'(\overline{\alpha}) = t(\overline{\alpha}); r(\overline{[w']})}.$$

Note that the set X of variables is actually a parameter of these constructions, and we need not assume X to be fixed and countable. In particular, for $X = \emptyset$, we adopt the notation $\mathcal{T}_{\mathcal{R}}$. The equations in 1 make $\mathcal{T}_{\mathcal{R}}(X)$ a category, the equations in 2 make each $f \in \Sigma$ a functor, and 3 forces the axioms E. The exchange law states that any rewriting of the form $r(\overline{\alpha})$—which represents the *simultaneous* rewriting of the term at the top using rule r *and* "below," i.e., in the subterms matched by the variables, using the rewrites $\overline{\alpha}$—is equivalent to the sequential composition $r(\overline{[w]}); t'(\overline{\alpha})$, corresponding to first rewriting on top with r and then below on the subterms matched by the variables with $\overline{\alpha}$, and is also equivalent to the sequential composition $t(\overline{\alpha}); r(\overline{[w']})$ corresponding to first rewriting below with $\overline{\alpha}$ and then on top with r. Therefore, the exchange law states that rewriting at the top by means of rule r and rewriting "below"

[6] In the expressions appearing in the equations, when compositions of morphisms are involved, we always implicitly assume that the corresponding domains and codomains match.

using $\overline{\alpha}$ are processes that are independent of each other and can be done either simultaneously or in any order.

Since each proof term is a description of a concurrent computation, what these equations provide is an equational theory of *true concurrency* allowing us to characterize when two such descriptions specify the same abstract computation.

Note that, since $[t(x_1,\ldots,x_n)]$ and $[t'(x_1,\ldots,x_n)]$ can both be regarded as functors $\mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$, from the mathematical point of view the exchange law just asserts that r is a *natural transformation*.

Lemma 1 *[37] For each rewrite rule $r : [t(x_1,\ldots,x_n)] \longrightarrow [t'(x_1,\ldots,x_n)]$ in R, the family of morphisms*

$$\{r(\overline{[w]}) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w}/\overline{x})] \mid \overline{[w]} \in T_{\Sigma,E}(X)^n\}$$

is a natural transformation $r : [t(x_1,\ldots,x_n)] \Rightarrow [t'(x_1,\ldots,x_n)]$ between the functors $[t(x_1,\ldots,x_n)], [t'(x_1,\ldots,x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$.

The category $\mathcal{T}_{\mathcal{R}}(X)$ is just one among many *models* that can be assigned to the rewrite theory $\mathcal{R}$. The general notion of model, called an $\mathcal{R}$*-system*, is defined as follows:

Definition 1 *Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an* $\mathcal{R}$-system $\mathcal{S}$ *is a category $\mathcal{S}$ together with:*

- *a (Σ, E)-algebra structure given by a family of functors*

  $$\{f_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S} \mid f \in \Sigma_n, n \in \mathbb{N}\}$$

  satisfying the equations E, i.e., for any $t(x_1,\ldots,x_n) = t'(x_1,\ldots,x_n)$ in E we have an identity of functors $t_{\mathcal{S}} = t'_{\mathcal{S}}$, where the functor $t_{\mathcal{S}}$ is defined inductively from the functors $f_{\mathcal{S}}$ in the obvious way.

- *for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$.*

An $\mathcal{R}$-homomorphism *$F : \mathcal{S} \longrightarrow \mathcal{S}'$ between two $\mathcal{R}$-systems is then a functor $F : \mathcal{S} \longrightarrow \mathcal{S}'$ such that it is a Σ-algebra homomorphism, i.e., $f_{\mathcal{S}} * F = F^n * f_{\mathcal{S}'}$, for each f in Σ_n, $n \in \mathbb{N}$, and such that "F preserves R," i.e., for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R we have the identity of natural transformations*[7] *$r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, where n is the number of variables appearing in the rule. This defines a category* $\mathcal{R}$-**Sys** *in the obvious way.*

A detailed proof of the following theorem on the existence of initial and free $\mathcal{R}$-systems for the more general case of conditional rewrite theories is given in [37], where the soundness and completeness of rewriting logic for $\mathcal{R}$-system models is also proved.

Theorem 1 *$\mathcal{T}_{\mathcal{R}}$ is an initial object in the category* $\mathcal{R}$-**Sys**. *More generally, $\mathcal{T}_{\mathcal{R}}(X)$ has the following universal property: Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \longrightarrow |\mathcal{S}|$ extends uniquely to an $\mathcal{R}$-homomorphism $F^{\natural} : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$.*

[7] Note that we use diagrammatic order for the *horizontal*, $\alpha * \beta$, and *vertical*, $\gamma;\delta$, composition of natural transformations [31].

4.3 Semantics of Object-Oriented and System Modules

As already pointed out, the logic of Maude is the membership logic variant of rewriting logic *MRWLogic*. A *system module* is then a rewrite theory $\mathcal{R}$ in such a logic. In the unparameterized case its semantics is the initial model $\mathcal{T}_{\mathcal{R}}$, that was constructed for the unsorted case in Section 4.2.2. That is, the initial model $\mathcal{T}_{\mathcal{R}}$ is the algebra of all rewriting computations for ground terms in the theory. From a systems perspective this model describes all the concurrent behaviors that the system so axiomatized can exhibit. From that perspective, a term t denotes a state of the system, and a proof term $\alpha : t \longrightarrow t'$ denotes a possibly concurrent computation.

A system module can contain one or more parameter theories. The inclusion from the parameter(s) into the module then gives rise to a free extension functor [36], which provides the semantics for the module. This of course means that we can compose systems by putting together the rewrite theories in which they are specified, as done in Full Maude.

A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean that the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary theory is undecidable. What we instead require for theories in system modules is that:

- The equations are divided into a set A of axioms, for which matching algorithms exist in the Maude implementation,[8] and a set E of equations that are Church-Rosser, terminating and sort decreasing *modulo* A; that is, the equational part must be equivalent to a functional module.

- The rules R in the module are *coherent* [61] (or at least what might be called "weakly coherent" [38, Section 5.2.1][60]) with the equations E modulo A. This means that appropriate critical pairs exist between rules and equations, allowing us to intermix rewriting with rules and rewriting with equations without loosing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A. In particular, a simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R. This is precisely the strategy adopted by the Maude interpreter.

Since the state of the system specified by a system module is axiomatized as an abstract data type by the equations E modulo A, and the rules in R are local rules for changing such a state, in practice the lefthand sides of rules in R only involve constructor patterns, so that coherence is a natural byproduct of good specification practice. Besides, using the completion methods in [61], one can check coherence, and one can try to make a set of rules coherent when they are not so.

The semantics of object-oriented modules is entirely reducible to that of system modules, in the sense that—as explained in Section 3.2.2 and in [38]—there is a systematic desugaring process translating each object-oriented module into

[8] Maude's rewrite engine has an extensible design, so that matching algorithms for new theories can be added and can be combined with existing ones [22]. As already mentioned, in the present version, matching modulo associativity, commutativity, (left-, right- or two-sided) identity, and idempotency, and most combinations of these attributes are supported.

its corresponding system module [38]. The particular ontology supported by object-oriented modules is something very much worth keeping, and it does not exist for general system modules. For example, in an object-oriented configuration we have objects that maintain their *identity* across their state changes, and the notions of fairness adequate for them are more specialized than those appropriate for arbitrary system modules. The approach taken in Maude is to provide a logical semantics for concurrent object-oriented programming by taking rewriting logic as its foundation, and then defining in a rigorous way higher-level object-oriented concepts above such a foundation. The papers [38, 39] provide good background on such foundations. Talcott's paper [58] gives rewriting logic foundations for actors from a somewhat different viewpoint. The paper [45] shows how, for object-oriented modules satisfying some simple requirements, their initial model semantics coincides with a very natural partial order of events truly concurrent semantics.

The basic ideas about the reflective semantics of Maude have already been discussed in Section 2.5. Much more detail can be found in [15, 10].

Bibliography

[1] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[2] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proc. SOFTSEM'95*, pages 363–368. Springer LNCS 1012, 1995.

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[4] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. To appear in *Theoretical Computer Science*, `http://maude.csl.sri.com`.

[5] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.

[6] R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, July 1998.

[7] Rod Burstall and Joseph A. Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.

[8] C. Castro. An approach to solving binary CSP using computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[9] M. Clavel, F. Durán, S. Eker, J. Meseguer, and P. Lincoln. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[10] Manuel Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.

[11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.

[12] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. `http://maude.csl.sri.com`.

[13] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[14] Manuel Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288, 1996. `http://jerry.cs.uiuc.edu/reflection/`.

[15] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[16] Manuel Clavel and José Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.

[17] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.

[18] Francisco Durán and José Meseguer. An extensible module algebra for Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.

[19] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Technical report, Computer Science Laboratory, SRI International, February 1999.

[20] José Meseguer (ed.). *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.

[21] Steven Eker. Term rewriting with operator evaluation strategy. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.

[22] Steven Eker. Fast matching in combination of regular equational theories. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[23] K. Futatsugi and R. Diaconescu. CafeOBJ report. AMAST Series, World Scientific, 1998.

[24] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.

[25] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

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

[27] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1999.

[28] C. Kirchner and H. Kirchner (eds.). *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.

[29] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.

[30] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.

[31] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[32] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[33] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[34] J. Meseguer and C. Talcott. Using rewriting logic to interoperate architectural description languages (I and II). Lectures at the Santa Fe and Seattle DARPA-EDCS Workshops, March and July 1997. `http://www-formal.stanford.edu/clt/ArpaNsf/adl-interop.html`.

[35] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[36] José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.

[37] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[38] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[39] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.

[40] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96, Pisa, August 1996*, pages 331–372. Springer LNCS 1119, 1996.

[41] José Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., *Proc. WADT'97*, 18–61, Springer LNCS 1376, 1998.

[42] José Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997.* Springer-Verlag, 1999.

[43] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.

[44] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[45] José Meseguer and Carolyn Talcott. A partial order event model for concurrent objects. Manuscript, November 1998.

[46] José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[47] E. Najm and J-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Vol. 2*, pages 157–176. Chapman & Hall, 1997.

[48] S. Nakajima. Encoding mobility in CafeOBJ: an exercise of describing mobile code-based software architecture. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan.* CafeOBJ Project, April 1998.

[49] Peter Csaba Ölveczky and José Meseguer. Specifying real-time systems in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[50] J. F. Quesada. Bidirectional and event-driven parsing with multi-virtual trees. In C. Martin-Vide, editor, *Mathematical and Computational Models in Linguistics*. John Benjamins, 1996.

[51] J. F. Quesada. Overparsing. In *Workshop on Mathematical Linguistics*. Pennsylvania State University, State College, 1998.

[52] J. F. Quesada. The Maude parser: Parsing and meta-parsing β-extended Technical Report , SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 1999. *To appear.*

[53] J. F. Quesada. The SCP parsing algorithm. Technical Report , SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 1999. *To appear.*

[54] J.F. Quesada. *The SCP parsing algorithm based on syntactic constraint propagation*. PhD thesis, University of Seville, 1997.

[55] Christophe Ringeissen. Prototyping combination of unification algorithms with the ELAN rule-based programming language. In H. Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*. Springer LNCS 1232, 1997.

[56] M.-O. Stehr. A rewriting semantics for algebraic Petri nets. Manuscript, March 1998, SRI International and C.S. Dept., Univ. of Hamburg, 1998.

[57] C. L. Talcott. An actor rewrite theory. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[58] C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, number 4 in Electronic Notes in Theoretical Computer Science. North Holland, 1996.

[59] A. van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[60] P. Viry. Rewriting modulo a rewrite system. TR-95-20, C.S. Department, University of Pisa, 1996.

[61] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.

[62] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. PhD thesis, Université Henry Poincaré — Nancy I, 1994.

[63] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

Appendix A

List of Core Maude Commands

A.1 Rewriting Commands

We use curly bracket pairs, { and } to enclose optional syntax.

reduce {**in** *module* **:**} *term* **.**

Causes the specified term to be reduced using the equations and membership axioms in the given module. **reduce** may be abbreviated to **red**. If the **in** clause is omitted the current module is assumed. Examples:

```
reduce 6 * 7 == 42 .
reduce in QID : conc('a, 'b) .
```

rewrite {[*number*]} {**in** *module* **:**} *term* **.**

Causes the specified term to be rewritten using the rules, equations and membership axioms in the given module. The default interpreter for rules applies rules using a top down (lazy) strategy and stops when the number of rule applications reaches the given bound. No rule will be applied if an equation can be applied. If the **in** clause is omitted the current module is assumed. If the upper bound clause is omitted, infinity is assumed. Examples:

```
rewrite 6 * 7 == 42 .
rewrite in FOO : f(6, g(a, b)) .
rewrite [50] f(6, g(a, b)) .
rewrite [1] in BAR : h(a) .
```

loop {**in** *module* **:**} *term* **.**

This command is used to initialize the read-eval-print loop in a module importing `LOOP-MODE` (see Section 2.8). The specified term is rewritten as far as possible using the rules, equations and membership axioms in the given module. If the result has a loop constructor symbol on the top then it becomes the current state of the loop; also, the list of quoted identifiers in the output position of the loop constructor is printed as a sequence of identifiers.

(*identifier**)
This command is used to input a list of identifiers to the loop in a module importing `LOOP-MODE`. If the current module has not changed since the last rewriting command, the result of previous rewrites has a loop constructor symbol on the top, and the last rewriting command was not **reduce** then:

1. the sequence of identifiers in the parentheses are converted into a list of quoted identifiers and are placed under the input position of the loop constructor;
2. a nil list of quoted identifiers is placed under the output position of the loop constructor;
3. the new term is rewritten as far as possible using the rules, equations and membership axioms in the module to which the term belongs; and
4. if the new result has a loop constructor symbol on the top, this list of quoted identifiers in the output position of the loop constructor is printed as a sequence of identifiers.

continue {*number*} **.**
Attempts to continue rewriting the result of the last rewriting command using the rules, equations and membership axioms, stopping if the upper bound on the number of rule applications is reached. This command is only usable if the current module has not changed since the last rewriting command, and the last rewriting command was not **reduce**. If no upper bound clause is given, infinity is assumed.

A.2 Matching Commands

Matching commands are used to directly invoke the rewriting engine's term pattern matcher. They can be useful for figuring out exactly what subjects can be matched by a complex pattern.

match {[*number*]} {**in** *module* **:**} *pattern* `<=?` *subject* **.**
This performs straight-forward matching in the given module. This kind of matching is used by the engine for applying membership axioms. The result is a list of at most *number* matching substitutions. If the upper bound clause is omitted, infinity is assumed. Example:

```
match [5] in FOO : +(X, *(X, Y)) <=? +(*(a, b), *(c, d)) .
```

xmatch {[*number*]} {**in** *module* **:**} *pattern* `<=?` *subject* **.**
This works similarly to the previous command, except that it performs matching with extension for those theories that need it (currently just those including the *assoc* attribute). If the subject (after theory-normalization) has a symbol f from an extension theory on top, only a piece of the top theory layer with f on top need be matched. This kind of matching is used by the engine for applying equations and rules in order to accurately simulate congruence class rewriting. The result is a list of all matches. If only part of the subject was matched, that part is given. Example:

```
xmatch +(*(P, Q), *(X, Y)) <=? +(*(a, b), *(c, d), *(a, e)) .
```

A.3 Tracing Commands

Tracing produces detailed information about each rewrite performed and each conditional rewrite attempted. Since this typically results in an unmanageably huge volume of output there are commands to control what is actually displayed.

set trace on . / set trace off .
These commands turn tracing on and off. If tracing is turned on, all trace information will be generated internally even if none of it is displayed, thus considerably slowing the speed of interpretation.

set trace condition on . / set trace condition off .
Determines whether the evaluations of conditions are traced.

set trace whole on . / set trace whole off .
Determines whether the whole term is printed before and after a rewrite.

set trace substitution on . / set trace substitution off .
Determines whether the substitution is printed.

set trace mb on . / set trace mb off .
Determines whether membership axiom applications are printed.

set trace eq on . / set trace eq off .
Determines whether equation applications are printed.

set trace rl on . / set trace rl off .
Determines whether rule applications are printed.

set trace select on . / set trace select off .
Determines whether only trace information for selected operator symbols is printed (rather than all symbols).

trace select *symbols* **. / trace deselect** *symbols* **.**
Selects/deselects operator symbols from the current module for tracing with the select option. Examples:

```
trace select foo bar baz .
trace deselect baz .
```

A.4 Print Option Commands

set print mixfix on . / set print mixfix off .
Controls whether operators with mixfix syntax are printed in mixfix or prefix form. User-defined syntax is supported for pretty printing even though it is not currently supported for parsing. It is sometimes advantageous to have uniform prefix notation for output; for example, if the output is going to be post-processed by some other tool. Default is on.

set print graph on . / set print graph off .
If on, terms that are internally represented by graphs (currently, result terms together with terms being reduced and terms in substitutions during tracing) are printed as graph representations rather than as terms, together with the number of operator symbols in the full term. This can be useful in some pathological cases where the size of the term is exponential in the size of the graph. Default is off.

set print flattened on . / set print flattened off .
Controls whether arguments under function symbols with the associative attribute are printed in flattened form or not. Default is on.

set print with parentheses on . / set print with parentheses off .
If on, additional mixfix terms are printed with additional parentheses to make grouping explicit. Default is off.

set show stats on . / set show stats off .
Determines whether the number of rewrites is printed with the results of the **reduce**, **rewrite** and **continue** commands. Default is on.

set show timing on . / set show timing off .
Determines whether the cpu and real time used during rewriting is printed with the results of the **reduce**, **rewrite** and **continue** commands. Default is on.

set show command on . / set show command off .
Determines whether the full form of certain commands is printed before they are executed. Default is on.

set show gc on . / set show gc off .
Determines which message is printed when a garbage collect is performed. Default is off.

A.5 Show Commands

show module {*module*} **.**
Prints out a representation of the given module (or of the current module if none is given).

show all {*module*} **.**
Prints out a *flattened* representation of the given module (or of the current module if none is given).

show sorts {*module*} **.**
Prints out a representation of the sort and subsort information for the given module (or for the current module if none is given).

show ops {*module*} **.**
Lists the operators in the given module (or in the current module if none is given).

show vars {*module*} **.**
Lists the variables in the given module (or in the current module if none is given).

show mbs {*module*} **.**
Lists the membership axioms in the given module (or in the current module if none is given).

show eqs {*module*} **.**
Lists the equations in the given module (or in the current module if none is given).

show rls {*module*} **.**
Lists the rules in the given module (or in the current module if none is given).

show components {*module*} **.**
Lists the connected components of the poset of sorts for the given module (or for the current module if none is given).

show summary {*module*} **.**
Shows a summary of statistics for the context free grammar and term rewriting system generated for the given module (or for the current module if none is given).

A.6 Debugger Commands

debug reduce {**in** *module* **:**} *term* **.**
Works just like the **reduce** command above, except that it drops into the debugger before executing the first rewrite.

debug rewrite {[*number*]} {**in** *module* **:**} *term* **.**
Works just like the **rewrite** command above, except that it drops into the debugger before executing the first rewrite.

resume .
Only usable from the debugger. Exits the debugger and resumes the current rewriting activity.

abort .
Only usable from the debugger. Exits the debugger and abandons the current rewriting activity.

step .
Only usable from the debugger. Performs a single step of the current rewriting activity with tracing switched on.

where .
Only usable from the debugger. Prints the stack of pending rewrite tasks together with explanations of how they arose.

A.7 Miscellaneous Commands

select *module* **.**
Selects a named module to be the current module. All commands that require a module refer to the current module unless a module is explicitly given. The current module is usually the last module entered or used.

set include *module* **on .** / **set include** *module* **off .**
Adds or removes the named module from the set of modules that are automatically included in every module.

A.8 System Commands

These commands control system level things. Unlike all the above commands they are not followed by a period. Unlike in OBJ3, they may be used inside a module definition at any point at which a keyword such as *var* could legally be used.

pwd
: Prints the path of the working directory.

ls {*flags*} {*directories*}
: Runs the UNIX *ls* command to list the files in the specified directories or working directory if none specified. The allowable flags depend on your local implementation of *ls*. Example:

```
ls -lF /usr/bin /usr/local
```

cd *directory-name*
: Changes the working directory to *directory-name*.

pushd *directory-name*
: Saves the current working directory on a stack and then changes the working directory to *directory-name*.

popd
: Changes the working directory to that which is on the top of the directory stack and pops the directory stack.

in *file-name*
: Causes a specified file to be included at this point. The full file name must be given, together with a full path name if the file is not in the current working directory. May be nested, i.e. the included file may contain *in* commands. Example:

```
in ../Examples/foo.maude
```

eof
: Causes the interpreter to respond as if it had reached the end of file.

quit
: Causes the interpreter to exit.

A.9 Abbreviations and Synonyms

The following abbreviations and synonyms are supported for module syntax.

pr	=	protecting
inc	=	including
sorts	=	sort
subsorts	=	subsort
assoc	=	associative
comm	=	commutative
idem	=	idempotent
id:	=	identity:
strat	=	strategy
prec	=	precedence
vars	=	var

The following abbreviations and synonyms are supported for command syntax.

red	=	reduce
rew	=	rewrite
cond	=	condition
subst	=	substitution
cont	=	continue
flat	=	flattened
parens	=	parentheses
cmd	=	command
sort	=	sorts
op	=	ops
var	=	vars
mb	=	mbs
eq	=	eqs
rl	=	rls
kinds	=	components
mod	=	module

A.10 Deprecated features

The following features support (very) limited backward compatibility with the OBJ family of languages. They may be omitted in future releases and thus should not be used in new code.

Commands
The following OBJ commands are recognized as equivalent to Maude commands:

set gc show on .	=	set show gc on .
set gc show off .	=	set show gc off .
set stats on .	=	set show stats on .
set stats off .	=	set show stats off .

Abbreviations and Synonyms
The following abbreviations and synonyms are supported for module syntax.

obj	=	fmod
endo	=	endfm
jbo	=	endfm
cq	=	ceq

Appendix B

The Grammar of Core Maude

This appendix describes the syntax of Maude using the following extended BNF notation: the symbols '{' and '}' are used as meta-parentheses; the symbol '|' is used to separate alternatives; square bracket pairs, '[' and ']' enclose optional syntax; '*' indicate zero or more repetitions of preceding unit; '+' indicate one or more repetitions of preceding unit; and "x" denotes x literally. As an application of this notation, A{, A}... indicates a nonempty list of A's separated by commas. Finally, %%% indicates comments in the syntactic description, as opposed to comments in the Maude code.

```
⟨MaudeTop⟩ ::=
    { ⟨SystemCommand⟩ | ⟨Command⟩ | ⟨DebuggerCommand⟩ | ⟨Module⟩ }+

⟨SystemCommand⟩ ::= in ⟨FileName⟩ |
    quit | eof | popd | pwd |
    cd ⟨Directory⟩ | push ⟨Directory⟩ |
    ls [ ⟨LsFlag⟩ ] [ ⟨Directory⟩ ] |

⟨Command⟩ ::= select ⟨ModId⟩ . |
    parse [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    reduce [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    rewrite [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    { match | xmatch } [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ <=? ⟨Term⟩ . |
    continue ⟨Nat⟩ . |
    loop [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    ( ⟨TokenString⟩ ) |
    trace { select | deselect } { ⟨OpId⟩ | ( ⟨OpForm⟩ ) }+ . |
    show ⟨ShowItem⟩ [ ⟨ModId⟩ ] . |
    set ⟨SetOption⟩ { on | off } .

⟨ShowItem⟩ ::= module | all | sorts | ops | vars | mbs |
    eqs | rls | summary | components

⟨SetOption⟩ ::= show ⟨ShowOption⟩ |
    print ⟨PrintOption⟩ |
    trace [ ⟨TraceOption⟩ ] |
    include ⟨ModId⟩
```

```
⟨ShowOption⟩ ::= stats | timing | command | gc

⟨PrintOption⟩ ::= mixfix | flat | with parentheses | graph

⟨TraceOption⟩ ::= condition | whole | substitution | select |
    mbs | eqs | rls

⟨DebuggerCommand⟩ ::= debug reduce [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    debug rewrite [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    debug continue ⟨Nat⟩ . |
    resume . | abort . | step . | where .

⟨Module⟩ ::= fmod ⟨ModId⟩ is ⟨ModElt⟩* endfm |
    mod ⟨ModId⟩ is ⟨ModElt'⟩* endfm |

⟨ModElt⟩ ::= including ⟨ModId⟩ . |
    sorts ⟨SortId⟩+ . |
    subsort ⟨SortId⟩+ { < ⟨SortId⟩+ }+ . |
    op ⟨OpForm⟩ : ⟨SortId⟩* -> ⟨SortId⟩ [ ⟨Attr⟩ ] . |
    ops { ⟨OpId⟩ | ( ⟨OpForm⟩ ) }+ : ⟨SortId⟩* -> ⟨SortId⟩ [ ⟨Attr⟩ ] . |
    var ⟨VarId⟩+ : ⟨SortId⟩ . |
    mb ⟨Term⟩ : ⟨SortId⟩ . |
    cmb ⟨Term⟩ : ⟨SortId⟩ if ⟨Condition⟩ . |
    eq ⟨Term⟩ = ⟨Term⟩ . |
    ceq ⟨Term⟩ = ⟨Term⟩ if ⟨Condition⟩ .

⟨ModElt'⟩ ::= ⟨ModElt⟩ |
    rl [ [ ⟨LabelId⟩ ] : ] ⟨Term⟩ => ⟨Term⟩ . |
    crl [ [ ⟨LabelId⟩ ] : ] ⟨Term⟩ => ⟨Term⟩ if ⟨Condition⟩ .

⟨Condition⟩ ::= ⟨Term⟩ = ⟨Term⟩ | ⟨Term⟩

⟨Attr⟩ ::=
    [ { assoc | comm |
        [ left | right ] id: ⟨Term⟩ |
        idem | memo |
        strat ( ⟨Nat⟩+ ) |
        prec Nat |
        gather ( { e | E | & }+ ) |
        special ( ⟨Hook⟩+ ) }+ ]

⟨Hook⟩ ::= id-hook ⟨Token⟩ [ ( ⟨TokenString⟩ ) ] |
    { op-hook | term-hook } ( ⟨TokenString⟩ )

⟨FileName⟩    %%% OS dependent
⟨Directory⟩   %%% OS dependent
⟨LsFlag⟩      %%% OS dependent

⟨ModId⟩       %%% simple identifier, by convention all caps
⟨SortId⟩      %%% simple identifier, by convention capitalized
⟨VarId⟩       %%% simple identifier, by convention capitalized
```

⟨*OpId*⟩ `%%% identifier possibly with underscores`
⟨*OpForm*⟩ `::=` ⟨*OpId*⟩ `|` `(` ⟨*OpForm*⟩ `)` `|` ⟨*OpForm*⟩`+`
⟨*Nat*⟩ `%%% a natural number`
⟨*Term*⟩ `::=` ⟨*Token*⟩ `|` `(` ⟨*Term*⟩ `)` `|` ⟨*Term*⟩`+`
⟨*Token*⟩ `%%% Any symbol other than ( or )`
⟨*TokenString*⟩ `::=` ⟨*Token*⟩ `|` `(` ⟨*TokenString*⟩ `)` `|` ⟨*TokenString*⟩`*`
⟨*LabelId*⟩ `%%% simple identifier`

B.1 Lexical Issues

Tokens are sequences of printable ASCII characters delimited by white space, except that '`(`', '`)`', '`[`', '`]`', '`{`', '`}`', and '`,`' are always considered as single character tokens unless backquoted.

Single line comments are started by one of `***` or `---`, and ended by the end of line. Multiline comments are started by `***(` and ended by `)`. Parentheses (whether backquoted or not) must balance within multiline comments.

Appendix C

The Signature of Full Maude

The Full Maude *system* is defined as a Core Maude module. That is, the entire semantics of Full Maude is defined and executed in Core Maude. The full definition of the Full Maude system, including the definition of all the internal functions implementing the system can be found in [19]. In particular, the *grammar* of the Full Maude *language*, that a user should follow to enter modules and commands, is itself a functional submodule of the overall Full Maude system specification. This allows giving the following specification of the Full Maude grammar in a form more perspicuous in certain ways than the corresponding BNF grammar form.

```
fmod SIGNS&VIEW-EXPRS is
  sorts Token NeTokenList Bubble
        SortToken Sort SortList SortDecl
        ViewToken ViewExp
        SubsortRel SubsortDecl
        OpDecl Attr AttrList Hook HookList .
  subsorts SortToken < Sort < SortList .
  subsort ViewToken < ViewExp .
  subsort Attr < AttrList .
  subsort Hook < HookList .

  op ((_)) : Token -> Token .

  *** extended sorts
  op _[_] : Sort ViewExp -> Sort [prec 40] .
  op __ : SortList SortList -> SortList [assoc] .
  op _,_ : ViewExp ViewExp -> ViewExp [assoc] .

  *** sort declaration
  op sorts_. : SortList -> SortDecl .
  op sort_. : SortList -> SortDecl .

  *** subsort declaration
  op subsort_. : SubsortRel -> SubsortDecl .
  op subsorts_. : SubsortRel -> SubsortDecl .
  op _<_ : SortList SortList -> SubsortRel .
  op _<_ : SortList SubsortRel -> SubsortRel .
```

```
  *** operator attributes
  op __ : AttrList AttrList -> AttrList [assoc] .
  op assoc : -> Attr .
  op associative : -> Attr .
  op comm : -> Attr .
  op commutative : -> Attr .
  op id:_ : Bubble -> Attr .
  op identity:_ : Bubble -> Attr .
  op left id:_ : Bubble -> Attr .
  op left identity:_ : Bubble -> Attr .
  op right id:_ : Bubble -> Attr .
  op right identity:_ : Bubble -> Attr .
  op strat(_) : NeTokenList -> AttrList .
  op strategy(_) : NeTokenList -> AttrList .
  op prec_ : Token -> Attr .
  op precedence_ : Token -> Attr .
  op gather(_) : NeTokenList -> Attr .
  op gathering(_) : NeTokenList -> Attr .
  op idem : -> Attr .
  op idempotent : -> Attr .

  op special(_) : HookList -> Attr .
  op __ : HookList HookList -> HookList [assoc] .
  op id-hook_ : Token -> Hook .
  op id-hook_(_) : Token NeTokenList -> Hook .
  op op-hook_(_:_->_) : Token Token NeTokenList Token -> Hook .
  op op-hook_(_: ->_) : Token Token Token -> Hook .
  op term-hook_(_) : Token Bubble -> Hook .

  *** operator declaration
  op op_: ->_. : Token Sort -> OpDecl .
  op op_: ->_[_]. : Token Sort AttrList -> OpDecl .
  op op_:_->_. : Token SortList Sort -> OpDecl .
  op op_:_->_[_]. : Token SortList Sort AttrList -> OpDecl .
  op ops_: ->_. : NeTokenList Sort -> OpDecl .
  op ops_: ->_[_]. : NeTokenList Sort AttrList -> OpDecl .
  op ops_:_->_. : NeTokenList SortList Sort -> OpDecl .
  op ops_:_->_[_]. : NeTokenList SortList Sort AttrList -> OpDecl .
endfm

fmod F&S-MODS&THS is
  including SIGNS&VIEW-EXPRS .

  sorts FDeclList SDeclList PreModule
        ImportDecl ModExp Parameter ParameterList
        ModuleName EquationDecl RuleDecl MembAxDecl
        VarDecl VarDeclList .
  subsort Parameter < ParameterList .
  subsorts Token < ModExp ModuleName .
  subsort VarDecl < VarDeclList .
  subsorts VarDecl ImportDecl SortDecl SubsortDecl
           OpDecl MembAxDecl EquationDecl VarDeclList < FDeclList .
  subsorts RuleDecl FDeclList < SDeclList .

  *** variable declaration
  op vars_:_. : NeTokenList Sort -> VarDecl .
  op var_:_. : NeTokenList Sort -> VarDecl .
```

```
  *** membership axiom declaration
  op mb_:_. : Bubble Sort -> MembAxDecl .
  op cmb_:_if_. : Bubble Sort Bubble -> MembAxDecl .

  *** equation declaration
  op eq_=_. : Bubble Bubble -> EquationDecl .
  op ceq_=_if_. : Bubble Bubble Bubble -> EquationDecl .

  *** rule declaration
  op rl[_] :_=>_. : Token Bubble Bubble -> RuleDecl .
  op crl[_]:_=>_if_. : Token Bubble Bubble Bubble -> RuleDecl .

  *** importation declaration
  op including_. : ModExp -> ImportDecl .
  op inc_. : ModExp -> ImportDecl .
  op protecting_. : ModExp -> ImportDecl .
  op pr_. : ModExp -> ImportDecl .

  *** parameterized module interface
  op _::_ : Token ModExp -> Parameter [prec 40 gather (e &)] .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc] .
  op _[_] : Token ParameterList -> ModuleName .

  *** declaration list
  op __ : VarDeclList VarDeclList -> VarDeclList [assoc] .
  op __ : SDeclList SDeclList -> SDeclList [assoc] .
  op __ : FDeclList FDeclList -> FDeclList [assoc] .

  *** functional and system module and theory
  op fmod_is_endfm : ModuleName FDeclList -> PreModule .
  op mod_is_endm : ModuleName SDeclList -> PreModule .
  op fth_is_endfth : ModuleName FDeclList -> PreModule .
  op th_is_endth : ModuleName SDeclList -> PreModule .
endfm

fmod OO-MODS&THS is
  including F&S-MODS&THS .

  sorts ClassDecl AttrDecl AttrDeclList
        SubclassDecl MsgDecl ODeclList .
  subsorts SDeclList MsgDecl SubclassDecl ClassDecl < ODeclList .
  subsort AttrDecl < AttrDeclList .

  *** object-oriented module and theory
  op omod_is_endom : ModuleName ODeclList -> PreModule .
  op oth_is_endoth : ModuleName ODeclList -> PreModule .

  *** class declaration
  op class_|_. : Sort AttrDeclList -> ClassDecl .
  op class_. : Sort -> ClassDecl .
  op _,_ : AttrDeclList AttrDeclList -> AttrDeclList [assoc] .
  op _:_ : Token Sort -> AttrDecl [prec 40] .

  *** subclass declaration
  op subclass_. : SubsortRel -> SubclassDecl .
  op subclasses_. : SubsortRel -> SubclassDecl .
```

```
  op _<_ : SortList SortList -> SubsortRel .
  op _<_ : SortList SubsortRel -> SubsortRel .

  *** message declaration
  op msg_:_->_. : Token SortList Sort -> MsgDecl .
  op msgs_:_->_. : NeTokenList SortList Sort -> MsgDecl .
endfm

fmod MOD-EXPRS is
  including OO-MODS&THS .

  sorts Map MapList .
  subsort Map < MapList .

  *** module expression
  op _*(_) : ModExp MapList -> ModExp .
  op _[_] : ModExp ViewExp -> ModExp .

  *** renaming maps
  op op_to_ : Token Token -> Map .
  op op_:_->_to_ : Token SortList Sort Token -> Map .
  op op_: ->_to_ : Token Sort Token -> Map .
  op op_to_[_] : Token Token AttrList -> Map .
  op op_:_->_to_[_] : Token SortList Sort Token AttrList -> Map .
  op op_: ->_to_[_] : Token Sort Token AttrList -> Map .
  op sort_to_ : Sort Sort -> Map .
  op label_to_ : Token Token -> Map .
  op class_to_ : Sort Sort -> Map .
  op attr_._to_ : Token Sort Token -> Map .
  op msg_to_ : Token Token -> Map .
  op msg_:_->_to_ : Token SortList Sort Token -> Map .
  op msg_: ->_to_ : Token Sort Token -> Map .

  op _,_ : MapList MapList -> MapList [assoc prec 42] .
endfm

fmod VIEWS is
  including OO-MODS&THS .

  sorts ViewDecl ViewDeclList PreView .
  subsorts VarDecl < ViewDecl < ViewDeclList .
  subsort VarDeclList < ViewDeclList .

  *** view maps
  op op_to term_. : Bubble Bubble -> ViewDecl .
  op op_to_. : Token Token -> ViewDecl .
  op op_:_->_to_. : Token SortList Sort Token -> ViewDecl .
  op op_: ->_to_. : Token Sort Token -> ViewDecl .
  op sort_to_. : Sort Sort -> ViewDecl .
  op class_to_. : Sort Sort -> ViewDecl .
  op attr_._to_. : Token Sort Token -> ViewDecl .
  op msg_to_. : Token Token -> ViewDecl .
  op msg_:_->_to_. : Token SortList Sort Token -> ViewDecl .
  op msg_: ->_to_. : Token Sort Token -> ViewDecl .

  *** view
  op view_from_to_is_endv :
```

```
        ViewToken ModExp ModExp ViewDeclList -> PreView .
  op __ : ViewDeclList ViewDeclList -> ViewDeclList [assoc] .
endfm

fmod COMMANDS is
  including MOD-EXPRS .

  sorts PreCommand .

  *** down function
  op down_:_ : ModExp PreCommand -> PreCommand .

  *** reduce command
  op red_. : Bubble -> PreCommand .
  op red-in_:_. : ModExp Bubble -> PreCommand .
  op reduce_. : Bubble -> PreCommand .
  op reduce-in_:_. : ModExp Bubble -> PreCommand .

  *** rewrite command
  op rew_. : Bubble -> PreCommand .
  op rew[_]_. : Token Bubble -> PreCommand .
  op rew-in_:_. : ModExp Bubble -> PreCommand .
  op rew-in[_]_:_. : Token ModExp Bubble -> PreCommand .
  op rewrite_. : Bubble -> PreCommand .
  op rewrite[_]_. : Token Bubble -> PreCommand .
  op rewrite-in_:_. : ModExp Bubble -> PreCommand .
  op rewrite-in[_]_:_. : Token ModExp Bubble -> PreCommand .

  *** select command
  op select_. : ModExp -> PreCommand .

  *** show commands
  op show module . : -> PreCommand .
  op show module_. : ModExp -> PreCommand .
  op show all . : -> PreCommand .
  op show all_. : ModExp -> PreCommand .
  op show sorts . : -> PreCommand .
  op show sorts_. : ModExp -> PreCommand .
  op show ops . : -> PreCommand .
  op show ops_. : ModExp -> PreCommand .
  op show vars . : -> PreCommand .
  op show vars_. : ModExp -> PreCommand .
  op show mbs . : -> PreCommand .
  op show mbs_. : ModExp -> PreCommand .
  op show eqns . : -> PreCommand .
  op show eqns_. : ModExp -> PreCommand .
  op show rls . : -> PreCommand .
  op show rls_. : ModExp -> PreCommand .
  op show view_. : ViewExp -> PreCommand .
  op show modules . : -> PreCommand .
  op show views . : -> PreCommand .
endfm

fmod FULL-MAUDE-SIGN is
  including VIEWS .
  including COMMANDS .
endfm
```

Appendix D

Standard Library of Predefined Modules

```
***        Maude interpreter standard prelude
***
***        Some of the overall structure is adapted from the OBJ3
***        interpreter standard prelude.
***

fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [special (id-hook SystemTrue)] .
  op false : -> Bool [special (id-hook SystemFalse)] .
endfm

fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
        [special (id-hook BranchSymbol
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _==_ : Universal Universal -> Bool
        [prec 51
         special (id-hook EqualitySymbol
                  term-hook equalTerm (true)
                  term-hook notEqualTerm (false))] .

  op _=/=_ : Universal Universal -> Bool
        [prec 51
         special (id-hook EqualitySymbol
                  term-hook equalTerm (false)
                  term-hook notEqualTerm (true))] .
endfm

fmod BOOL is
  protecting TRUTH .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
```

```
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  vars A B C : Bool .
  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

set include BOOL on .

fmod IDENTICAL is
  op _===_ : Universal Universal -> Bool
        [prec 51 strat (0)
         special (id-hook EqualitySymbol
                  term-hook equalTerm (true)
                  term-hook notEqualTerm (false))] .

  op _=/==_ : Universal Universal -> Bool
        [prec 51 strat (0)
         special (id-hook EqualitySymbol
                  term-hook equalTerm (false)
                  term-hook notEqualTerm (true))] .
endfm

fmod MACHINE-INT is
  sorts MachineInt NzMachineInt .
  subsort NzMachineInt < MachineInt .
  op <MachineInts> : -> NzMachineInt [special (id-hook MachineIntegerSymbol)] .
  op <MachineInts> : -> MachineInt [special (id-hook MachineIntegerSymbol)] .

  op -_ : MachineInt -> MachineInt
        [prec 15
         special (id-hook MachineIntegerOpSymbol (-)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op -_ : NzMachineInt -> NzMachineInt
        [prec 15
         special (id-hook MachineIntegerOpSymbol (-)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op ~_ : MachineInt -> MachineInt
        [prec 15
         special (id-hook MachineIntegerOpSymbol (~)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _+_ : MachineInt MachineInt -> MachineInt
        [prec 33 gather (E e)
         special (id-hook MachineIntegerOpSymbol (+)
                  op-hook machineIntBaseSymbol
```

```
                      (<MachineInts> : -> MachineInt))] .

  op _-_ : MachineInt MachineInt -> MachineInt
        [prec 33 gather (E e)
         special (id-hook MachineIntegerOpSymbol (-)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _*_ : MachineInt MachineInt -> MachineInt
        [prec 31 gather (E e)
         special (id-hook MachineIntegerOpSymbol (*)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _*_ : NzMachineInt NzMachineInt -> NzMachineInt
        [prec 31 gather (E e)
         special (id-hook MachineIntegerOpSymbol (*)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _/_ : MachineInt NzMachineInt -> MachineInt
        [prec 31 gather (E e)
         special (id-hook MachineIntegerOpSymbol (/)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _%_ : MachineInt NzMachineInt -> MachineInt
        [prec 31 gather (E e)
         special (id-hook MachineIntegerOpSymbol (%)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _&_ : MachineInt MachineInt -> MachineInt
        [prec 53 gather (E e)
         special (id-hook MachineIntegerOpSymbol (&)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _|_ : MachineInt MachineInt -> MachineInt
        [prec 57 gather (E e)
         special (id-hook MachineIntegerOpSymbol (|)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _|_ : NzMachineInt NzMachineInt -> NzMachineInt
        [prec 57 gather (E e)
         special (id-hook MachineIntegerOpSymbol (|)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _^_ : MachineInt MachineInt -> MachineInt
        [prec 55 gather (E e)
         special (id-hook MachineIntegerOpSymbol (^)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _>>_ : MachineInt MachineInt -> MachineInt
```

```
        [prec 35 gather (E e)
         special (id-hook MachineIntegerOpSymbol (>>)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _<<_ : MachineInt MachineInt -> MachineInt
        [prec 35 gather (E e)
         special (id-hook MachineIntegerOpSymbol (<<)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op _<_ : MachineInt MachineInt -> Bool
        [prec 37
         special (id-hook MachineIntegerOpSymbol (<)
                  op-hook machineIntBaseSymbol (<MachineInts> : -> MachineInt)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _<=_ : MachineInt MachineInt -> Bool
        [prec 37
         special (id-hook MachineIntegerOpSymbol (<=)
                  op-hook machineIntBaseSymbol (<MachineInts> : -> MachineInt)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _>_ : MachineInt MachineInt -> Bool
        [prec 37
         special (id-hook MachineIntegerOpSymbol (>)
                  op-hook machineIntBaseSymbol (<MachineInts> : -> MachineInt)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .

  op _>=_ : MachineInt MachineInt -> Bool
        [prec 37
         special (id-hook MachineIntegerOpSymbol (>=)
                  op-hook machineIntBaseSymbol (<MachineInts> : -> MachineInt)
                  term-hook trueTerm (true)
                  term-hook falseTerm (false))] .
endfm

fmod QID is
  protecting MACHINE-INT .
  sort Qid .
  op <Qids> : -> Qid [special (id-hook QuotedIdentifierSymbol)] .

  op conc : Qid Qid -> Qid
        [special (id-hook QuotedIdentifierOpSymbol (conc)
                  op-hook qidBaseSymbol (<Qids> : -> Qid))] .

  op index : Qid MachineInt -> Qid
        [special (id-hook QuotedIdentifierOpSymbol (index)
                  op-hook qidBaseSymbol (<Qids> : -> Qid)
                  op-hook machineIntBaseSymbol
                          (<MachineInts> : -> MachineInt))] .

  op strip : Qid -> Qid
        [special (id-hook QuotedIdentifierOpSymbol (strip)
```

```
                  op-hook qidBaseSymbol (<Qids> : -> Qid))] .
endfm

fmod QID-LIST is
  protecting QID .
  sort QidList .
  subsort Qid < QidList .
  op nil : -> QidList .
  op __ : QidList QidList -> QidList [assoc id: nil] .
endfm

fmod META-LEVEL is
  protecting QID-LIST .
  sorts FModule Module ModuleExpression Import ImportList
        QidSet MachineIntList Sort SortDecl SubsortDecl SubsortDeclSet Attr
        AttrSet OpDecl OpDeclSet VarDecl VarDeclSet Term TermList Equation
        EquationSet Rule RuleSet MembAx MembAxSet Assignment Substitution
        ResultPair Hook HookList .
  subsort FModule < Module .
  subsort Import < ImportList .
  subsort Qid < ModuleExpression .
  subsort Qid < QidSet .
  subsort MachineInt < MachineIntList .
  subsort Qid < Sort .
  subsort SubsortDecl < SubsortDeclSet .
  subsort Attr < AttrSet .
  subsort OpDecl < OpDeclSet .
  subsort VarDecl < VarDeclSet .
  subsort Qid < Term .
  subsort Term < TermList .
  subsort Equation < EquationSet .
  subsort Rule < RuleSet .
  subsort MembAx < MembAxSet .
  subsort Assignment < Substitution .
  subsort Hook < HookList .

  op _[_] : Qid TermList -> Term .
  op {_}_ : Qid Qid -> Term .
  op _,_ : TermList TermList -> TermList [assoc gather (e E) prec 120] .
  op _:_ : Term Qid -> Term .
  op _::_ : Term Qid -> Term .
  op none : -> QidSet .
  op _;_ : QidSet QidSet -> QidSet [assoc comm id: none] .
  op nil : -> MachineIntList .
  op __ : MachineIntList MachineIntList -> MachineIntList [assoc id: nil] .

  op fmod_is_______endfm : Qid ImportList SortDecl SubsortDeclSet
        OpDeclSet VarDeclSet MembAxSet EquationSet -> FModule
        [gather (& & & & & & & &)] .
  op mod_is________endm : Qid ImportList SortDecl SubsortDeclSet
        OpDeclSet VarDeclSet MembAxSet EquationSet RuleSet -> Module
        [gather (& & & & & & & & &)] .
  op nil : -> ImportList .
  op __ : ImportList ImportList -> ImportList [assoc id: nil] .
  op including_. : ModuleExpression -> Import .

  op sorts_. : QidSet -> SortDecl .
```

```
  op subsort_<_. : Qid Qid -> SubsortDecl .
  op none : -> SubsortDeclSet .
  op __ : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
        [assoc comm id: none] .

  op (op_:_->_[_].) : Qid QidList Qid AttrSet -> OpDecl .
  op none : -> OpDeclSet .
  op __ : OpDeclSet OpDeclSet -> OpDeclSet [assoc comm id: none] .

  op none : -> AttrSet .
  op __ : AttrSet AttrSet -> AttrSet [assoc comm id: none] .
  op assoc : -> Attr .
  op comm : -> Attr .
  op idem : -> Attr .
  op id : Term -> Attr .
  op left-id : Term -> Attr .
  op right-id : Term -> Attr .
  op strat : MachineIntList -> Attr .
  op memo : -> Attr .
  op prec : MachineInt -> Attr .
  op gather : QidList -> Attr .

  op special : HookList -> Attr .
  op __ : HookList HookList -> HookList [assoc] .
  op id-hook : Qid QidList -> Hook .
  op op-hook : Qid Qid QidList Qid -> Hook .
  op term-hook : Qid Term -> Hook .

  op var_:_. : Qid Qid -> VarDecl .
  op none : -> VarDeclSet .
  op __ : VarDeclSet VarDeclSet -> VarDeclSet [assoc comm id: none] .

  op mb_:_. : Term Qid -> MembAx .
  op cmb_:_if_=_. : Term Qid Term Term -> MembAx .
  op none : -> MembAxSet .
  op __ : MembAxSet MembAxSet -> MembAxSet [assoc comm id: none] .

  op eq_=_. : Term Term -> Equation .
  op ceq_=_if_=_. : Term Term Term Term -> Equation .
  op none : -> EquationSet .
  op __ : EquationSet EquationSet -> EquationSet [assoc comm id: none] .

  op rl[_]:_=>_. : Qid Term Term -> Rule .
  op crl[_]:_=>_if_=_. : Qid Term Term Term Term -> Rule .
  op none : -> RuleSet .
  op __ : RuleSet RuleSet -> RuleSet [assoc comm id: none] .

  op _<-_ : Qid Term -> Assignment .
  op none : -> Substitution .
  op _;_ : Substitution Substitution -> Substitution [assoc comm id: none] .
  op {_,_} : Term Substitution -> ResultPair .

  op error* : -> Term .
  op errorSort : QidSet -> Sort .

  op meta-reduce : Module Term -> Term
     [special (
```

```
id-hook MetaLevelOpSymbol       (meta-reduce)

op-hook machineIntBaseSymbol    (<MachineInts> : -> MachineInt)
op-hook qidBaseSymbol           (<Qids> : -> Qid)

op-hook nilMachineIntListSymbol (nil : -> MachineIntList)
op-hook machineIntListSymbol
        (__ : MachineIntList MachineIntList -> MachineIntList)
op-hook emptyQidSetSymbol       (none : -> QidSet)
op-hook qidSetSymbol            (_;_ : QidSet QidSet -> QidSet)
op-hook nilQidListSymbol        (nil : -> QidList)
op-hook qidListSymbol           (__ : QidList QidList -> QidList)

op-hook fmodSymbol
        (fmod_is_______endfm :
                Qid ImportList SortDecl SubsortDeclSet OpDeclSet
                VarDeclSet MembAxSet EquationSet -> FModule)
op-hook modSymbol
        (mod_is________endm :
                Qid ImportList SortDecl SubsortDeclSet OpDeclSet
                VarDeclSet MembAxSet EquationSet RuleSet -> Module)
op-hook nilImportListSymbol     (nil : -> ImportList)
op-hook importListSymbol        (__ : ImportList ImportList -> ImportList)
op-hook includingSymbol         (including_. : ModuleExpression -> Import)

op-hook sortSymbol              (sorts_. : QidSet -> SortDecl)
op-hook emptySubsortDeclSetSymbol (none : -> SubsortDeclSet)
op-hook subsortDeclSetSymbol
        (__ : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet)
op-hook subsortSymbol           (subsort_<_. : Qid Qid -> SubsortDecl)

op-hook opDeclSetSymbol         (__ : OpDeclSet OpDeclSet -> OpDeclSet)
op-hook emptyOpDeclSetSymbol    (none : -> OpDeclSet)
op-hook opDeclSymbol
        (op_:_->_[_]. : Qid QidList Qid AttrSet -> OpDecl)

op-hook emptyAttrSetSymbol      (none : -> AttrSet)
op-hook attrSetSymbol           (__ : AttrSet AttrSet -> AttrSet)
op-hook assocSymbol             (assoc : -> Attr)
op-hook commSymbol              (comm : -> Attr)
op-hook idemSymbol              (idem : -> Attr)
op-hook idSymbol                (id : Term -> Attr)
op-hook leftIdSymbol            (left-id : Term -> Attr)
op-hook rightIdSymbol           (right-id : Term -> Attr)
op-hook stratSymbol             (strat : MachineIntList -> Attr)
op-hook memoSymbol              (memo : -> Attr)
op-hook precSymbol              (prec : MachineInt -> Attr)
op-hook gatherSymbol            (gather : QidList -> Attr)

op-hook specialSymbol           (special : HookList -> Attr)
op-hook hookListSymbol          (__ : HookList HookList -> HookList)
op-hook idHookSymbol            (id-hook : Qid QidList -> Hook)
op-hook opHookSymbol            (op-hook : Qid Qid QidList Qid -> Hook)
op-hook termHookSymbol          (term-hook : Qid Term -> Hook)

op-hook emptyVarDeclSetSymbol   (none : -> VarDeclSet)
op-hook varDeclSetSymbol
```

```
           (__ : VarDeclSet VarDeclSet -> VarDeclSet)
      op-hook varDeclSymbol           (var_:_. : Qid Qid -> VarDecl)

      op-hook metaTermSymbol          (_[_] : Qid TermList -> Term)
      op-hook metaDisambigSymbol      ({_}_ : Qid Qid -> Term)
      op-hook metaArgSymbol           (_,_ : TermList TermList -> TermList)

      op-hook emptyMembAxSetSymbol    (none : -> MembAxSet)
      op-hook membAxSetSymbol         (__ : MembAxSet MembAxSet -> MembAxSet)
      op-hook mbSymbol                (mb_:_. : Term Qid -> MembAx)
      op-hook cmbSymbol
           (cmb_:_if_=_. : Term Qid Term Term -> MembAx)

      op-hook emptyEquationSetSymbol  (none : -> EquationSet)
      op-hook equationSetSymbol
           (__ : EquationSet EquationSet -> EquationSet)
      op-hook eqSymbol                (eq_=_. : Term Term -> Equation)
      op-hook ceqSymbol
           (ceq_=_if_=_. : Term Term Term Term -> Equation)

      op-hook emptyRuleSetSymbol      (none : -> RuleSet)
      op-hook ruleSetSymbol           (__ : RuleSet RuleSet -> RuleSet)
      op-hook rlSymbol                (rl[_]:_=>_. : Qid Term Term -> Rule)
      op-hook crlSymbol
           (crl[_]:_=>_if_=_. : Qid Term Term Term Term -> Rule)

      op-hook membPredSymbol          (_:_ : Term Qid -> Term)
      op-hook lazyMembPredSymbol      (_::_ : Term Qid -> Term)

      op-hook substitutionSymbol
           (_;_ : Substitution Substitution -> Substitution)
      op-hook emptySubstitutionSymbol (none : -> Substitution)
      op-hook assignmentSymbol        (_<-_ : Qid Term -> Assignment)
      op-hook resultPairSymbol
           ({_,_} : Term Substitution -> ResultPair)

      op-hook metaErrorSymbol         (error* : -> Term)
      op-hook errorSortSymbol         (errorSort : QidSet -> Sort)
      term-hook trueTerm              (true)
      term-hook falseTerm             (false))] .

op meta-rewrite : Module Term MachineInt -> Term
      [special (
         id-hook MetaLevelOpSymbol    (meta-rewrite)
         op-hook shareWith            (meta-reduce : Module Term -> Term))] .

op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair
      [special (
         id-hook MetaLevelOpSymbol    (meta-apply)
         op-hook shareWith            (meta-reduce : Module Term -> Term))] .

op meta-parse : Module QidList -> Term
      [special (
             id-hook MetaLevelOpSymbol (meta-parse)
         op-hook shareWith            (meta-reduce : Module Term -> Term))] .

op meta-pretty-print : Module Term -> QidList
```

```
       [special (
             id-hook MetaLevelOpSymbol (meta-pretty-print)
          op-hook shareWith            (meta-reduce : Module Term -> Term))] .

  op sortLeq : Module Sort Sort -> Bool
       [special (
          id-hook MetaLevelOpSymbol    (sortLeq)
          op-hook shareWith            (meta-reduce : Module Term -> Term))] .

  op sameComponent : Module Sort Sort -> Bool
       [special (
          id-hook MetaLevelOpSymbol    (sameComponent)
          op-hook shareWith            (meta-reduce : Module Term -> Term))] .

  op leastSort : Module Term -> Sort
       [special (
          id-hook MetaLevelOpSymbol    (leastSort)
          op-hook shareWith            (meta-reduce : Module Term -> Term))] .

  op lesserSorts : Module Sort -> QidSet
       [special (
          id-hook MetaLevelOpSymbol    (lesserSorts)
          op-hook shareWith            (meta-reduce : Module Term -> Term))] .

  op glbSorts : Module Sort Qid -> QidSet
       [special (
          id-hook MetaLevelOpSymbol    (glbSorts)
          op-hook shareWith            (meta-reduce : Module Term -> Term))] .
endfm

mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System
       [special (
          id-hook LoopSymbol
          op-hook qidBaseSymbol        (<Qids> : -> Qid)
          op-hook nilQidListSymbol     (nil : -> QidList)
          op-hook qidListSymbol        (__ : QidList QidList -> QidList))] .
endm
```

Appendix E

A Software Architecture Interoperation Example

The following example—developed by Francisco Durán and José Meseguer as part of a broader joint project with Carolyn Talcott on the uses of rewriting logic as a semantic framework for the interoperation of architectural description languages (ADLs) [34]—is included for two purposes.

On the one hand, it is a medium-size example that exhibits many of the object-oriented and parameterized programming features of Full Maude; it can therefore be profitably used together with the examples in Sections 3.2 and 3.5 to become more familiar with the object-oriented and parameterized programming features of Full Maude.

On the other hand, the example has an interest in its own right as a non-trivial case study demonstrating the suitability of rewriting logic as a semantic framework, to give a formal semantics to, and to interoperate, different architectural description languages. Intuitively, such languages have different *semantic models*, but such models are not always precisely defined, and it is even less clear how one can interoperate in a correct way ADLs based on quite different semantic models.

The example is somewhat modest in its goals. In particular, we do not model the *syntax* of any existing ADLs. Instead, we specify in rewriting logic the *semantics* for the *semantic models* of several typical ADLs, including dataflow—in both a static and a reflective-dynamic form—message passing, and implicit invocation.

The need for having to use and interoperate several of these models is motivated by the example itself, namely a system in which images from ships in the ocean, together with information about their location, are first sent to an image recognition subsystem that has a typical "pipes and filters" dataflow architecture. In this case the dataflow architecture happens to be dynamic, so that it can be modified at runtime in a reflexive way by adding new recognition units to it. The results of the image recognition subsystem are then summarized and sent in a message-passing style to a command center that has an implicit invocation architecture, so that each object will react in its own particular way to the same event broadcast to all of them. Each of the objects in the command center can then send appropriate messages in response to the information that it receives. The overall architecture of the system is summarized in a pictorial way in Figure E.1.

```
(fmod MACHINE-INT* is
```

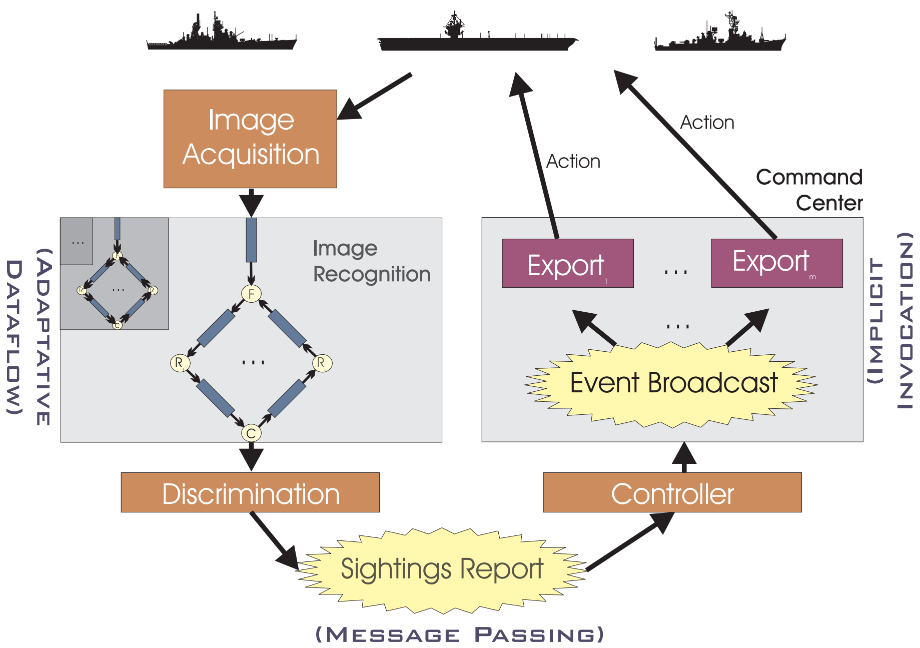


Figure E.1: Semantic interoperability of heterogeneous architectures.

```
   protecting MACHINE-INT .

   var N : MachineInt .
   var M : MachineInt .

   *** difference of two numbers
   op dif : MachineInt MachineInt -> MachineInt .
   eq dif(N, M)
     = if N > M
       then N - M
       else M - N
       fi .

   *** maximum of two numbers
   op max : MachineInt MachineInt -> MachineInt .
   eq max(N, M)
     = if N > M
       then N
       else M
       fi .
 endfm)


(fth TRIV is
  sort Elt .
 endfth)


(fmod DEFAULT[Y :: TRIV] is
  sort Default[Y] .
```

```
  subsort Elt.Y < Default[Y] .
  op null : -> Default[Y] .
 endfm)


(fmod PAIR[X :: TRIV, Y :: TRIV] is
  sort Pair[X, Y] .

  op <_;_> : Elt.X Elt.Y -> Pair[X, Y] .
  op 1st : Pair[X, Y] -> Elt.X .
  op 2nd : Pair[X, Y] -> Elt.Y .

  var A : Elt.X .
  var B : Elt.Y .

  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
 endfm)


(fmod TRIPLE[X :: TRIV, Y :: TRIV, Z :: TRIV] is
  sort Triple[X, Y, Z] .

  op <_;_;_> : Elt.X Elt.Y Elt.Z -> Triple[X, Y, Z] .
  op 1st : Triple[X, Y, Z] -> Elt.X .
  op 2nd : Triple[X, Y, Z] -> Elt.Y .
  op 3rd : Triple[X, Y, Z] -> Elt.Z .

  var A : Elt.X .
  var B : Elt.Y .
  var C : Elt.Z .

  eq 1st(< A ; B ; C >) = A .
  eq 2nd(< A ; B ; C >) = B .
  eq 3rd(< A ; B ; C >) = C .
 endfm)


(fmod LIST[X :: TRIV] is
  sort List[X] .
  subsort Elt.X < List[X] .

  op nil : -> List[X] .
  op _._ : List[X] List[X] -> List[X] [assoc id: nil] .
 endfm)


(fmod SET[X :: TRIV] is
  protecting BOOL .

  sorts Set[X] NeSet[X] .
  subsorts Elt.X < NeSet[X] < Set[X] .

  op mt : -> Set[X] .
  op __ : Set[X] Set[X] -> Set[X] [assoc comm id: mt] .
  op __ : NeSet[X] NeSet[X] -> NeSet[X] [assoc comm id: mt] .
  op _in_ : Elt.X Set[X] -> Bool .
```

```
  vars E E' : Elt.X .
  var S : Set[X] .

  eq E E = E .

  eq E in mt = false .
  eq E in (E' S) = E == E' or (E in S) .
 endfm)


(fth FUNCTION is
  sorts Domain Codomain .
  op f : Domain -> Codomain .
 endfth)


(view Domain from TRIV to FUNCTION is
  sort Elt to Domain .
 endv)


(view Codomain from TRIV to FUNCTION is
  sort Elt to Codomain .
 endv)


(fmod MAP[F :: FUNCTION] is
  protecting (SET[Domain] * (op __ to _;_))[F] .
  protecting SET[Codomain][F] .

  op map : Set[Domain][F] -> Set[Codomain][F] .

  var A : Domain.F .
  var S : Set[Domain][F] .

  eq map(mt) = (mt).Set`[Codomain`]`[F`] .
  eq map(A ; S) = f(A) map(S) .
 endfm)


(fmod PFUN[U :: TRIV, V :: TRIV] is
  protecting BOOL .

  protecting DEFAULT[V] .

  *** protecting PAIR[U, Default[V]] not supported.
  *** We use Pair[U, V] instead of Pair[U, Default[V]]

  sort Pair[U, V] .

  op <_;_> : Elt.U Default[V] -> Pair[U, V] .
  op 1st : Pair[U, V] -> Elt.U .
  op 2nd : Pair[U, V] -> Default[V] .

  var A : Elt.U .
  var B : Default[V] .
```

```
eq 1st(< A ; B >) = A .
eq 2nd(< A ; B >) = B .

*** protecting SET[Pair[U, Default[V]]] not supported.
*** We would like to be able to write
*** pr SET[Pair[U, Default[V]]]
***      * (sort Set[Pair[U, Default[V]]] to PairSet[U, Default[V]],
***         sort NeSet[Pair[U, Default[V]]] to NePairSet[U, Default[V]]) .
*** We use PairSet[U, V] and NePairSet[U, V] instead of
*** PairSet[U, Default[V]] and NePairSet[U, Default[V]].

sorts PairSet[U, V] NePairSet[U, V] .
subsorts Pair[U, V] < NePairSet[U, V] < PairSet[U, V] .

op mt : -> PairSet[U, V] .
op __ : PairSet[U, V] PairSet[U, V] -> PairSet[U, V]
      [assoc comm id: (mt).PairSet‘[U‘,V‘]] .
op __ : NePairSet[U, V] NePairSet[U, V] -> NePairSet[U, V]
      [assoc comm id: (mt).PairSet‘[U‘,V‘]] .
op _in_ : Pair[U, V] PairSet[U, V] -> Bool .

vars E E’ : Pair[U, V] .
var S : PairSet[U, V] .

eq E E = E .

eq E in (mt).PairSet‘[U‘,V‘] = false .
eq E in (E’ S) = (E == E’) or (E in S) .

*** We would like to be able to write
***
*** pr MAP[view to PAIR[U, Default[V]] is
***          sort Domain to Pair[U, Default[V]] .
***          sort Codomain to Elt.U .
***          op f to 1st .
***        endv]
***      * (sort Set[Domain][U] to PairSet[U, Default[V]],
***         sort Set[Codomain][Default[V]] to Set[U],
***         op map to dom) .
***
*** and
***
*** pr MAP[view to PAIR[U, Default[V]] is
***          sort Domain to Pair[U, Default[V]] .
***          sort Codomain to Default[V] .
***          op f to 2nd .
***        endv]
***      * (sort Set[Domain][U] to PairSet[U, Default[V]],
***         sort Set[Codomain][Default[V]] to Set[Default[V]],
***         op map to im) .
***
*** Instead, we use Set[U] and Set[V]

protecting SET[U] .
*** protecting SET[Default[V]] not supported.
*** We use Set[V] instead of Set[Default[V]].
```

```
  protecting SET[V] .

  sort PFun[U, V] .

  subsorts Pair[U, V] < PFun[U, V] < PairSet[U, V] .

  op mt : -> PFun[U, V] .
  op _`[_`] : PFun[U, V] Elt.U -> Default[V] .
  op _`[_->_`] : PFun[U, V] Elt.U Default[V] -> PFun[U, V] .

  vars C : Default[V] .
  var F : PFun[U, V] .

  op dom : PairSet[U, V] -> Set[U] .  *** domain
  eq dom((mt).PairSet`[U`,V`]) = (mt).Set`[U`] .
  eq dom(< A ; B > S) = A dom(S) .

  op im : PairSet[U, V] -> Set[V] .   *** image
  eq im((mt).PairSet`[U`,V`]) = (mt).Set`[V`] .
  eq im(< A ; B > S) = B im(S) .

  cmb < A ; B > F : PFun[U, V]
    if not(A in dom(F)) .

  eq (< A ; B > F)[ A ] = B .
  ceq F [ A ]
    = null
      if not(A in dom(F)) .
  eq (< A ; B > F)[ A -> C ]
    = < A ; C > F .
  ceq F [ A -> C ]
    = < A ; C > F
      if not(A in dom(F)) .
 endfm)


*** the following data type of Ports will be used for the input and output
*** ports of filter objects in the pipe and filters model.


(view NzMachineInt from TRIV to MACHINE-INT* is
  sort Elt to NzMachineInt .
 endv)


(fmod PORTS[X :: TRIV] is
  protecting BOOL .
  protecting PFUN[NzMachineInt, X]
             * (sort Pair[NzMachineInt, X] to Port[X],
                sort PairSet[NzMachineInt, X] to PortSet[X],
                sort NePairSet[NzMachineInt, X] to NePortSet[X]) .

  op put : Default[X] PortSet[X] -> PortSet[X] .
     *** set the value of all the ports to the given value
  op flush : PortSet[X] -> PortSet[X] .
     *** set all the ports to null
  op empty : PortSet[X] -> Bool .
```

```
    *** true if all the ports in the set are set to null
  op full : PortSet[X] -> Bool .
    *** true if all the ports in the set ar different from null

  var N : NzMachineInt .
  vars A B : Default[X] .
  var S : PortSet[X] .

  eq put(B, mt)
    = (mt).PortSet`[X`] .
  eq put(B, (< N ; A > S))
    = < N ; B > put(B, S) .

  eq flush(S)
    = put(null, S) .

  eq empty(mt)
    = true .
  eq empty(< N ; A > S)
    = (A == null) and empty(S) .

  eq full(mt)
    = true .
  eq full(< N ; A > S)
    = (A =/= null) and full(S) .
 endfm)


(omod OID is
  protecting MACHINE-INT* .
  op o : MachineInt -> Oid .
 endom)


(view MachineInt from TRIV to MACHINE-INT* is
  sort Elt to MachineInt .
 endv)


(view Triple`[MachineInt`,MachineInt`,MachineInt`]
      from TRIV
      to TRIPLE[MachineInt,MachineInt,MachineInt] is
   sort Elt to Triple[MachineInt, MachineInt, MachineInt] .
 endv)


(view List`[MachineInt`] from TRIV to LIST[MachineInt] is
   sort Elt to List[MachineInt] .
 endv)


(view Image from TRIV to LIST[MachineInt]*(sort List[MachineInt] to Image) is
  sort Elt to Image .
 endv)


*** A sighting consists of two-dimensional coords plus a time, plus
```

```
*** an image described as a list of block heights.
*** For example, the image of a destroyer, with shape
***
***                          #
***                          # # #
***                      # # # # # #
***                      # # # # # #,
***
*** is given by the list 2 . 2 . 4 . 3 . 3 . 2.

(fmod SIGHTING is
  pr LIST[MachineInt]*(sort List[MachineInt] to Image) .
  pr PAIR[Triple`[MachineInt`,MachineInt`,MachineInt`], Image]
       *(sort Triple[MachineInt, MachineInt, MachineInt] to Location,
         sort Pair[Triple`[MachineInt`,MachineInt`,MachineInt`], Image]
                   to Sighting,
         op 1st : Pair[Triple`[MachineInt`,MachineInt`,MachineInt`], Image]
                   -> Triple`[MachineInt`,MachineInt`,MachineInt`] to sighting,
         op 2nd : Pair[Triple`[MachineInt`,MachineInt`,MachineInt`], Image]
                   -> Image to image) .

  op distance : Image Image -> MachineInt .  *** distance between two images
  op size : Image -> MachineInt .
  op aircraft-carrier : -> Image .
  op oil-tanker : -> Image .
  op destroyer : -> Image .
  op speedboat : -> Image .

  vars N M : MachineInt .
  vars L Q : Image .

  eq size(nil) = 0 .
  eq size(N) = N .
  eq size(N . L) = N + size(L) .
  eq distance(nil, L) = size(L) .
  eq distance(L, nil) = size(L) .
  eq distance(N, M . L) = dif(N, M) + size(L) .
  eq distance(M . L, N) = dif(N, M) + size(L) .
  eq distance(N . L, M . Q) = dif(N, M) + distance(L, Q) .
  eq aircraft-carrier = 3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3 .
  eq oil-tanker = 2 . 2 . 2 . 2 . 2 . 2 . 2 . 3 .
  eq destroyer = 2 . 2 . 4 . 3 . 3 . 2 .
  eq speedboat = 1 . 1 .
 endfm)


(view Oid from TRIV to OID is
  sort Elt to Oid .
 endv)


(view Pair`[Oid`,NzMachineInt`] from TRIV to PAIR[Oid, NzMachineInt] is
  sort Elt to Pair[Oid, NzMachineInt] .
 endv)


(omod PIPE[X :: TRIV] is
```

```
  pr DEFAULT[Pair`[Oid`,NzMachineInt`]]
             * (sort Default[Pair`[Oid`,NzMachineInt`]] to DfltAddr) .
  pr LIST[X] .

  class Pipe[X] | from : DfltAddr, to : DfltAddr, q : List[X] .
 endom)


(omod FILTER[X :: TRIV] is
  including PIPE[X] .
  protecting PORTS[X] .

  class Filter[X] | in : PortSet[X], out : PortSet[X] .

  vars O P : Oid .
  var N : NzMachineInt .
  var E : Elt.X .
  var S : PortSet[X] .
  var Q : List[X] .

  rl [filter-1] :
   < O : Filter[X] | out : (< N ; E > S) >
   < P : Pipe[X] | from : < O ; N >, q : Q >
   => < O : Filter[X] | out : (< N ; null > S) >
      < P : Pipe[X] | q : (E . Q) > .

  rl [filter-2] :
   < O : Filter[X] | in : (< N ; null > S) >
   < P : Pipe[X] | to : < O ; N >, q : E >
   => < O : Filter[X] | in : (< N ; E > S) >
      < P : Pipe[X] | q : nil > .

  rl [filter-3] :
   < O : Filter[X] | in : (< N ; null > S) >
   < P : Pipe[X] | to : < O ; N >, q : (Q . E) >
   => < O : Filter[X] | in : (< N ; E > S) >
      < P : Pipe[X] | q : Q > .
 endom)


(omod FANOUT[X :: TRIV] is
  including FILTER[X] .

  class Fanout[X] | cnt : MachineInt .
  subclass Fanout[X] < Filter[X] .

  var E : Elt.X .
  var S : PortSet[X] .
  var F : Oid .
  var N : MachineInt .

  crl [fanout] :
   < F : Fanout[X] | in : < 1 ; E >, out : S, cnt : N >
   => < F : Fanout[X] | in : < 1 ; (null).Default`[X`] >,
                        out : put(E, S),
                        cnt : (N + 1) >
      if empty(S) and not (E == (null).Default`[X`]) .
```

```
 endom)


(view Sighting from TRIV to SIGHTING is
  sort Elt to Sighting .
 endv)


(view Loc-Eval from FUNCTION
               to PAIR[Sighting, MachineInt]
                    * (sort Pair[Sighting, MachineInt] to Loc-Eval) is
    sort Domain to Loc-Eval .
    sort Codomain to Location .
    var D : Domain .
    op f(D) to term sighting(1st(D)) .
 endv)


(view ImageSight from TRIV to SIGHTING is
  sort Elt to Image .
 endv)


(view Pair`[ImageSight`,MachineInt`]
       from TRIV to PAIR[ImageSight, MachineInt] is
    sort Elt to Pair[ImageSight, MachineInt] .
 endv)


(fmod DATA is
  protecting SIGHTING .
  protecting MAP[Loc-Eval]
              *(sort Set[Domain][Loc-Eval] to Loc-Evals,
                sort NeSet[Domain][Loc-Eval] to Ne-Loc-Evals,
                op mt : -> Set[Domain][Loc-Eval] to mt-Loc-Evals,
                sort Set[Codomain][Loc-Eval] to Locations,
                sort NeSet[Codomain][Loc-Eval] to Ne-Locations,
                op mt : -> Set[Codomain][Loc-Eval] to mt-Locations,
                op map to locs) .
  protecting SET[Pair`[ImageSight`,MachineInt`]]
               *(sort Pair[ImageSight, MachineInt] to Eval,
                 sort Set[Pair`[ImageSight`,MachineInt`]] to Evals,
                 sort NeSet[Pair`[ImageSight`,MachineInt`]] to NeEvals,
                 op mt to mt-Evals) .

  sort Data .
  subsort Sighting < Data .
  subsort Loc-Evals < Data .

  op unidentified-object : -> Eval .
  op winners : Loc-Evals NzMachineInt -> Evals .

  var LEVS : Loc-Evals .
  vars N M : MachineInt .
  var L : Location .
  var I : Image .
```

```
  eq winners(mt-Loc-Evals, N) = mt-Evals .
  eq winners((< < L ; I > ; M > ; LEVS), N)
    = if N > M
      then (< I ; M > winners(LEVS, N))
      else winners(LEVS, N)
      fi .
 endfm)


(view Data from TRIV to DATA is
  sort Elt to Data .
 endv)


(omod RECOGNIZER is
  including FILTER[Data] .

  class Recognizer | model : Image .
  subclass Recognizer < Filter[Data] .

  var L : Location .
  vars I M : Image .
  var R : Oid .

  rl [recognizer] :
   < R : Recognizer | in : < 1 ; < L ; I > >, out : < 1 ; null >, model : M >
   => < R : Recognizer |
         in : < 1 ; null >,
         out : < 1 ; < < L ; M > ; distance(I, M) > > > .

  *** this rule assumes unique input and output ports and singleton
  *** data values in them as the way recognizer filters are used.
 endom)


(omod COLLECTOR is
  including FILTER[Data] .

  class Collector | cnt : MachineInt .
  subclass Collector < Filter[Data] .

  op evals : PortSet[Data] -> Loc-Evals .

  var S : PortSet[Data] .
  var C : Oid .
  var N : MachineInt .
  var D : Data .

  eq evals(< N ; D > S) = (D ; evals(S)).Loc-Evals .
  eq evals((mt).PortSet`[Data`]) = mt-Loc-Evals .

  crl [collector] :
   < C : Collector | in : S, out : < 1 ; null >, cnt : N >
   => < C : Collector | in : flush(S), out : < 1 ; evals(S) >, cnt : (N + 1) >
      if full(S) and (S =/= mt) .
 endom)
```

```
(omod DISCRIMINATOR is
  including COLLECTOR .

  class Discriminator |
          collector : Oid, threshold : NzMachineInt, controller : Oid .

  msg to_at_evals_ot : Oid Location Evals -> Msg .

  var S : PortSet[Data] .
  vars C D G : Oid .
  var N : NzMachineInt .
  var LEVS : Loc-Evals .

  crl [discriminator] :
   < D : Discriminator | collector : C, threshold : N, controller : G >
   < C : Collector | out : < 1 ; LEVS > >
     => < D : Discriminator | >
        < C : Collector | out : < 1 ; null > >
        (to G at locs(LEVS) evals unidentified-object ot)
     if (locs(LEVS) : Location) and (winners(LEVS, N) == mt-Evals) .
  crl [discriminator] :
   < D : Discriminator | collector : C, threshold : N, controller : G >
   < C : Collector | out : < 1 ; LEVS > >
     => < D : Discriminator | >
        < C : Collector | out : < 1 ; null > >
        (to G at locs(LEVS) evals winners(LEVS, N) ot)
     if (locs(LEVS) : Location) and not (winners(LEVS, N) == mt-Evals) .

  *** the above rule uses the subsort inclusion Location < Locations and
  *** assumes that all the located evaluations originate from
  *** the same sighting and therefore have the same location

 endom)


(omod META is
  inc RECOGNIZER .
  inc DISCRIMINATOR .
  inc FANOUT[Data] .
  pr DEFAULT[Oid] * (sort Default[Oid] to DftOid) .
  pr DEFAULT[MachineInt] * (sort Default[MachineInt] to DftMachineInt) .

  sort State .

  op lock : Cid -> Cid .
  op _._ : Oid MachineInt -> Oid .
  op ready : -> State .
  op busy : -> State .

  msg to_install-recognizer_ot : Oid Image -> Msg .

  class Meta | in-pipe : DftOid, out-pipe : DftOid, fanout : DftOid,
             recognizer : DftOid, collector : DftOid, state : State,
             cnt : DftMachineInt, tag : MachineInt .

  vars M P P' R F C : Oid .
```

```
  var I : Image .
  vars N N' : MachineInt .
  vars S S' : PortSet[Data] .

  op max : NeSet[NzMachineInt] -> NzMachineInt .
  var T : NeSet[NzMachineInt] .
  eq max(N) = N .
  ceq max(N T) = N if N > max(T) .
  ceq max(N T) = max(T) if not (N > max(T)) .

  *** A meta-object can, upon request, dynamically change the configuration
  *** of the dataflow network it controls, by adding a new recognizer object
  *** to recognize a given image and two new pipes and by hooking it up
  *** through the pipes to the fanout and collector objects.

  rl [create] :
   (to M install-recognizer I ot)
   < M : Meta | state : ready, tag : N >
   => < M : Meta | state : busy, recognizer : (M . N), in-pipe : (M . (N + 1)),
           out-pipe : (M . (N + 2)), tag : (N + 3) >
      < (M . N) : Recognizer | in : < 1 ; (null).Default`[Data`] >,
           out : < 1 ; (null).Default`[Data`] >, model : I >
      < (M . (N + 1)) : Pipe[Data] |
           from : (null).DfltAddr, to : < (M . N) ; 1 >, q : nil >
      < (M . (N + 2)) : Pipe[Data] |
           from : < (M . N) ; 1 >, to : (null).DfltAddr, q : nil > .

  crl [hook-fanout-collector] :
   < M : Meta | state : busy, recognizer : R,
              fanout : F, in-pipe : P,
              collector : C, out-pipe : P' >
   < F : Fanout[Data] | cnt : N', out : S >
   < P : Pipe[Data] | from : (null).DfltAddr >
   < C : Collector | cnt : N', in : S' >
   < P' : Pipe[Data] | to : (null).DfltAddr >
   => < M : Meta | state : ready, recognizer : (null).DftOid,
              in-pipe : (null).DftOid, out-pipe : (null).DftOid >
      < F : Fanout[Data] |
              out : (S < (max(dom(S)) + 1) ; (null).Default`[Data`] >) >
      < P : Pipe[Data] | from : < F ; (max(dom(S)) + 1) > >
      < C : Collector |
              in : (S' < (max(dom(S')) + 1) ; (null).Default`[Data`] >) >
      < P' : Pipe[Data] | to : < C ; (max(dom(S')) + 1) > >
      if empty(S) and empty(S') .
 endom)


(omod IMAGE-RECOGNITION is
  including META .

  op init-conf : -> Configuration .

  *** this initial configuration has a pipe feeding images and linked
  *** to a fanout object that is then linked by pipes to three
  *** recognizers for speedboats, destroyers, and aircraft carriers
  *** their evaluations are then fed by other pipes into a collector
  *** object.  A discriminator object then selects the evaluations
```

```
*** accurate within a threshold and sends them (or an unidentified
*** object report) to a controller object.  Finally, there is a
*** metaobject that can dynamically change the dataflow network
*** by hooking up to it new recognizers via new pipes.

eq init-conf
  = < o(0) : Pipe[Data] |
          from : (null).DfltAddr,
          to : < o(1) ; 1 >,
          q : (< < 1 ; 2 ; 3 > ; (2 . 2 . 4 . 4 . 3 . 2) > .
               < < 4 ; 5 ; 2 > ; (2 . 1) > .
               < < 7 ; 5 ; 1 > ; (3 . 3 . 3 . 3 . 3 . 5 . 4 . 3 . 3) > .
               < < 0 ; 0 ; 0 > ; (2 . 2 . 2 . 2 . 2 . 2 . 2 . 3) >) >
  < o(1) : Fanout[Data] |
          in : < 1 ; (null).Default`[Data`] >,
          out : (< 1 ; (null).Default`[Data`] >
                 < 2 ; (null).Default`[Data`] >
                 < 3 ; (null).Default`[Data`] >),
          cnt : 0 >
  < o(2) : Pipe[Data] |
          from : < o(1) ; 1 >,
          to : < o(5) ; 1 >,
          q : nil >
  < o(3) : Pipe[Data] |
          from : < o(1) ; 2 >,
          to : < o(6) ; 1 >,
          q : nil >
  < o(4) : Pipe[Data] |
          from : < o(1) ; 3 >,
          to : < o(7) ; 1 >,
          q : nil >
  < o(5) : Recognizer |
          in : < 1 ; (null).Default`[Data`] >,
          out : < 1 ; (null).Default`[Data`] >,
          model : destroyer >
  < o(6) : Recognizer |
          in : < 1 ; (null).Default`[Data`] >,
          out : < 1 ; (null).Default`[Data`] >,
          model : speedboat >
  < o(7) : Recognizer |
          in : < 1 ; (null).Default`[Data`] >,
          out : < 1 ; (null).Default`[Data`] >,
          model : aircraft-carrier >
  < o(8) : Pipe[Data] |
          from : < o(5) ; 1 >,
          to : < o(11) ; 1 >,
          q : nil >
  < o(9) : Pipe[Data] |
          from : < o(6) ; 1 >,
          to : < o(11) ; 2 >,
          q : nil >
  < o(10) : Pipe[Data] |
          from : < o(7) ; 1 >,
          to : < o(11) ; 3 >,
          q : nil >
  < o(11) : Collector |
          in : (< 1 ; (null).Default`[Data`] >
```

```
                  < 2 ; (null).Default`[Data`] >
                  < 3 ; (null).Default`[Data`] >),
            out : < 1 ; (null).Default`[Data`] >,
            cnt : 0 >
    < o(12) : Discriminator |
            collector : o(11),
            threshold : 4,
            controller : o(13) >
    < o(13) : Meta |
            in-pipe : (null).DftOid,
            out-pipe : (null).DftOid,
            fanout : o(1),
            recognizer : (null).DftOid,
            collector : o(11),
            state : ready,
            cnt : (null).DftMachineInt,
            tag : 0 > .
 endom)


(omod IMPLICIT-INVOCATION[M :: TRIV, E :: TRIV, P :: TRIV] is

  pr PFUN[E, M] * (sort PFun[E, M] to Table[E, M]) .

  class Bubble | conf : Configuration .
  class Implicit | table : Table[E, M] .

  msg to_msg_with_ot : Oid Elt.M Elt.P -> Msg .

  op bc_with_in_cb : Elt.E Elt.P Configuration -> Configuration .

  sort ExtAct .
  subsort ExtAct < Msg .

  vars B O : Oid .
  var M : Elt.M .
  var E : Elt.E .
  var P : Elt.P .
  var T : Table[E, M] .
  vars C C' : Configuration .
  var MSG : Msg .
  var EA : ExtAct .

  crl [bc1] :
   bc E with P in C C' cb
   => bc E with P in C cb
      bc E with P in C' cb
      if (C =/= empty) and (C' =/= empty) .
  rl [bc2] :
   bc E with P in empty cb
   => (empty).Configuration .
  rl [bc3] :
   bc E with P in MSG cb
   => MSG .
  crl [bc4] :
   bc E with P in < O : Implicit | table : T > cb
   => < O : Implicit | table : T >
```

```
      if T[E] == null .
  crl [bc4] :
   bc E with P in < O : Implicit | table : T > cb
   => to O msg T[E] with P ot
      < O : Implicit | table : T >
      if not (T[E] == null) .

  rl [out] :
   < B : Bubble | conf : (EA C) >
   => < B : Bubble | conf : C > EA .
 endom)


(fmod OVERALL-SYSTEM0 is
  sorts Event MsgId .

  op sghtng : -> Event .
  op air-act : -> MsgId .
  op bttlshp-act : -> MsgId .
 endfm)


(view MsgId from TRIV to OVERALL-SYSTEM0 is
  sort Elt to MsgId .
 endv)


(view Event from TRIV to OVERALL-SYSTEM0 is
  sort Elt to Event .
 endv)


(view Location from TRIV to IMAGE-RECOGNITION is
  sort Elt to Location .
 endv)


(view Evals from TRIV to IMAGE-RECOGNITION is
  sort Elt to Evals .
 endv)


(view Pair`[Location`,Evals`] from TRIV to PAIR[Location, Evals] is
  sort Elt to Pair[Location, Evals] .
 endv)


(omod OVERALL-SYSTEM is
  inc IMAGE-RECOGNITION .

  inc IMPLICIT-INVOCATION[MsgId, Event, Pair`[Location`,Evals`]]
            * (class Bubble to Controller) .

  class Commander | table : Table[Event, MsgId], subordinate : Oid .
  subclass Commander < Implicit .

  msg to_threat-at_ot : Oid Location -> ExtAct .
```

```
  msg to_recon-at_ot : Oid Location -> ExtAct .

  op q : MachineInt -> Oid .

  vars O Q S : Oid .
  var L : Location .
  var EVS : Evals .
  var C : Configuration .

  rl [control] :
    to O at L evals EVS ot
    < O : Controller | conf : C >
    => < O : Controller | conf : (bc sghtng with < L ; EVS > in C cb) > .

  rl [bttlshp-act] :
    to Q msg bttlshp-act with < L ; EVS > ot
    < O : Commander | subordinate : S >
    => < O : Commander | >
      to S threat-at L ot .
  rl [air-act] :
    to Q msg air-act with < L ; EVS > ot
    < O : Commander | subordinate : S >
    => < O : Commander | >
       to S recon-at L ot .

  op syst-conf : -> Configuration .

  eq syst-conf
    = init-conf
    < o(14) : Controller |
         conf : (< q(0) : Commander | table : < sghtng ; air-act >,
                                      subordinate : q(1) >
                 < q(2) : Commander | table : < sghtng ; air-act >,
                                      subordinate : q(3) >) > .
endom)

Maude> (rew init-conf .)

Rewrite in OVERALL-SYSTEM : init-conf .
Result Configuration :
   < o(0) : Pipe[Data] | q : nil, from : null, to : < o(1) ; 1 > >
   < o(1) : Fanout[Data] | out : (< 1 ; null > < 2 ; null > < 3 ; null >),
                           in : < 1 ; null >, cnt : 4 >
   < o(2) : Pipe[Data] | q : nil, from : < o(1) ; 1 >, to : < o(5) ; 1 > >
   < o(3) : Pipe[Data] | q : nil, from : < o(1) ; 2 >, to : < o(6) ; 1 > >
   < o(4) : Pipe[Data] | q : nil, from : < o(1) ; 3 >, to : < o(7) ; 1 > >
   < o(5) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                         model : (2 . 2 . 4 . 3 . 3 . 2) >
   < o(6) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                         model : (1 . 1) >
   < o(7) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                         model : (3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3) >
   < o(8) : Pipe[Data] | q : nil, from : < o(5) ; 1 >, to : < o(11) ; 1 > >
   < o(9) : Pipe[Data] | q : nil, from : < o(6) ; 1 >, to : < o(11) ; 2 > >
   < o(10) : Pipe[Data] | q : nil, from : < o(7) ; 1 >, to : < o(11) ; 3 > >
   < o(11) : Collector | out : < 1 ; null >,
                         in : (< 1 ; null > < 2 ; null > < 3 ; null >),
```

```
                        cnt : 4 >
   < o(12) : Discriminator | controller : o(13), threshold : 4,
                            collector : o(11) >
   < o(13) : Meta | cnt : null, collector : o(11), tag : 0,
                   state : ready, out-pipe : null, in-pipe : null,
                   recognizer : null, fanout : o(1) >
   to o(13) at < 0 ; 0 ; 0 > evals unidentified-object ot
   to o(13) at < 1 ; 2 ; 3 > evals < 2 . 2 . 4 . 3 . 3 . 2 ; 1 > ot
   to o(13) at < 4 ; 5 ; 2 > evals < 1 . 1 ; 1 > ot
   to o(13) at < 7 ; 5 ; 1 > evals < 3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3 ; 2 > ot

Maude> (rew init-conf (to o(13) install-recognizer oil-tanker ot) .)
Rewrite in OVERALL-SYSTEM :
    init-conf
    to o(13) install-recognizer oil-tanker ot .
Result Configuration :
   < o(0) : Pipe[Data] | q : nil, from : null, to : < o(1) ; 1 > >
   < o(1) : Fanout[Data] |
             out : (< 1 ; null > < 2 ; null > < 3 ; null > < 4 ; null >),
             in : < 1 ; null >, cnt : 4 >
   < o(2) : Pipe[Data] | q : nil, from : < o(1) ; 1 >, to : < o(5) ; 1 > >
   < o(3) : Pipe[Data] | q : nil, from : < o(1) ; 2 >, to : < o(6) ; 1 > >
   < o(4) : Pipe[Data] | q : nil, from : < o(1) ; 3 >, to : < o(7) ; 1 > >
   < o(5) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                        model : (2 . 2 . 4 . 3 . 3 . 2) >
   < o(6) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                        model : (1 . 1) >
   < o(7) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                        model : (3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3) >
   < o(8) : Pipe[Data] | q : nil, from : < o(5) ; 1 >, to : < o(11) ; 1 > >
   < o(9) : Pipe[Data] | q : nil, from : < o(6) ; 1 >, to : < o(11) ; 2 > >
   < o(10) : Pipe[Data] | q : nil, from : < o(7) ; 1 >, to : < o(11) ; 3 > >
   < o(11) : Collector | out : < 1 ; null >,
              in : (< 1 ; null > < 2 ; null > < 3 ; null > < 4 ; null >),
              cnt : 4 >
   < o(12) : Discriminator | controller : o(13), threshold : 4,
                            collector : o(11) >
   < o(13) : Meta | cnt : null, collector : o(11), tag : 3,
                   state : ready, out-pipe : null, in-pipe : null,
                   recognizer : null, fanout : o(1) >
   < o(13) . 0 : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                             model : (2 . 2 . 2 . 2 . 2 . 2 . 2 . 3) >
   < o(13) . 1 : Pipe[Data] | q : nil, from : < o(1) ; 4 >,
                             to : < o(13) . 0 ; 1 > >
   < o(13) . 2 : Pipe[Data] | q : nil, from : < o(13) . 0 ; 1 >,
                             to : < o(11) ; 4 > >
   to o(13) at < 0 ; 0 ; 0 > evals < 2 . 2 . 2 . 2 . 2 . 2 . 2 . 3 ; 0 > ot
   to o(13) at < 1 ; 2 ; 3 > evals < 2 . 2 . 4 . 3 . 3 . 2 ; 1 > ot
   to o(13) at < 4 ; 5 ; 2 > evals < 1 . 1 ; 1 > ot
   to o(13) at < 7 ; 5 ; 1 > evals < 3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3 ; 2 > ot

Maude> (rew syst-conf .)
Rewrite in OVERALL-SYSTEM : syst-conf .
Result Configuration :
   < o(0) : Pipe[Data] | q : nil, from : null, to : < o(1) ; 1 > >
   < o(1) : Fanout[Data] | out : (< 1 ; null > < 2 ; null > < 3 ; null >),
                          in : < 1 ; null >, cnt : 4 >
```

```
< o(2) : Pipe[Data] | q : nil, from : < o(1) ; 1 >, to : < o(5) ; 1 > >
< o(3) : Pipe[Data] | q : nil, from : < o(1) ; 2 >, to : < o(6) ; 1 > >
< o(4) : Pipe[Data] | q : nil, from : < o(1) ; 3 >, to : < o(7) ; 1 > >
< o(5) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                      model : (2 . 2 . 4 . 3 . 3 . 2) >
< o(6) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                      model : (1 . 1) >
< o(7) : Recognizer | out : < 1 ; null >, in : < 1 ; null >,
                      model : (3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3) >
< o(8) : Pipe[Data] | q : nil, from : < o(5) ; 1 >, to : < o(11) ; 1 > >
< o(9) : Pipe[Data] | q : nil, from : < o(6) ; 1 >, to : < o(11) ; 2 > >
< o(10) : Pipe[Data] | q : nil, from : < o(7) ; 1 >, to : < o(11) ; 3 > >
< o(11) : Collector | out : < 1 ; null >,
                      in : (< 1 ; null > < 2 ; null > < 3 ; null >),
                      cnt : 4 >
< o(12) : Discriminator | controller : o(13), threshold : 4,
                          collector : o(11) >
< o(13) : Meta | cnt : null, collector : o(11), tag : 0,
                 state : ready, out-pipe : null, in-pipe : null,
                 recognizer : null, fanout : o(1) >
< o(14) : Controller |
          conf : (< q(0) : Commander | table : < sghtng ; air-act >,
                                       subordinate : q(1) >
                  < q(2) : Commander | table : < sghtng ; air-act >,
                                       subordinate : q(3) >) >
to o(13) at < 0 ; 0 ; 0 > evals unidentified-object ot
to o(13) at < 1 ; 2 ; 3 > evals < 2 . 2 . 4 . 3 . 3 . 2 ; 1 > ot
to o(13) at < 4 ; 5 ; 2 > evals < 1 . 1 ; 1 > ot
to o(13) at < 7 ; 5 ; 1 > evals < 3 . 3 . 3 . 3 . 3 . 4 . 3 . 3 . 3 ; 2 > ot
```