The Maude Specification of Full Maude

Francisco Durán and José Meseguer

Computer Science Laboratory
SRI International

May, 1999

Abstract

This document describes the executable formal specification of Full Maude—a version of Maude supporting a rich module algebra with module hierarchies, parameterization, views, theories, module expressions, and object-oriented modules—which is in fact its implementation in the Maude 1.00 system[1]. We have realized the design of Full Maude, which is described in detail in [14] and in [26], using the reflective capabilities of the rewriting logic language Maude itself.

The specification of Full Maude in the Core Maude sublanguage deals not only with module operations, but also with the functionality of a complete language, including parsing, pretty-printing, and an execution environment for such a language. The fact that Full Maude has Core Maude as a "sublanguage" simplifies somewhat the representation map between them. Nevertheless, the exact same methodology can be used for representing and executing in Maude many other languages and logics.

We emphasize the extensibility of the module algebra with which Maude has been endowed in this way. We present the specification of Full Maude in sufficient detail so that the different extensions of Full Maude become easily understandable. Note that by 'extensions' we not only mean those adding new module transformation operations in the style of the transformation from object-oriented modules to system modules, but also new module combining and/or manipulating operations. We illustrate some of the possibilities for extensions of this kind with several examples.

Contents

[1]Maude 1.00 is available free of charge in its web site at `maude.csl.sri.com`. The complete specification of Full Maude is available as part of the distribution package.

1 Introduction

The efforts required for building adequate tools for formal specification languages are considerable. Such efforts can be particularly intense when such languages are *executable*, since a good execution engine must also be developed to have a complete execution environment. The methods we propose to implement tools for formal specification languages in Maude (see also [15, 19]) can be used in practice to develop tools and environments for formal specification languages, including executable ones, and to endow such languages with a powerful module algebra of specification-combining operations.

We have applied these methods to the design and implementation of Maude itself. The most basic parts of the language—supporting module hierarchies of functional and system modules and some predefined modules—are implemented in C++, giving rise to a sublanguage called Core Maude. This is extended by special syntax for object-oriented specifications, and by a rich *module algebra* of parameterized modules and module composition in the Clear/OBJ style [9, 32], as described in [26, Chapter 3] and in [14], giving rise to the Full Maude language.

All of Full Maude has been formally specified in Core Maude [27, 15, 26]. This executable formal specification—a Maude specification of about 7,000 lines—is in fact its implementation in the Maude 1.00 system. Our experience in this regard is very encouraging in several respects. Firstly, because of how quickly we were able to develop Full Maude. Secondly, because of how easy it will be to maintain it, modify it, and extend it with new features and new module operations (see Section 11 and [26, Chapter 5]). Thirdly, because of the competitive performance with which we can carry out very complex module composition and module transformation operations, that makes the user interaction with Full Maude quite reasonable.

But the reflective methods described here, that underly our development of Full Maude, are much more general. They can equally be used to develop high-performance executable environments for other formal specification languages with much less effort and much greater flexibility, maintainability, and extensibility than what would be required in conventional implementations. For example, an executable environment for Action Semantics [52] is currently being considered, and it would be quite interesting to explore Maude implementations, for example, for a next-generation CafeOBJ [24] and for CASL [23]. Therefore, before discussing in detail the executable specification of Full Maude in Maude, we explain the more general methodology for building formal tools for different logics in Maude, of which the Full Maude specification in this work and the Church-Rosser checker in [26, Chapter 5] are particular applications.

1.1 Building Formal Tools in Maude

Suppose that we want to build a theorem prover for a logic, or an executable formal specification language. We can do so by representing the logic $\mathcal{L}$ of the theorem prover or specification language in question in rewriting logic by means of a representation map

$$\Psi : \mathcal{L} \longrightarrow RWLogic.$$

Such representations are typically very simple and natural. Using reflection we can, as further explained in Section 2.4, *internalize* such a map as an executable equationally defined function $\overline{\Psi}$. In Maude this is accomplished using the module `META-LEVEL` and its sort `Module`. Specifically, we can reify the above representation map Ψ by defining an abstract data type $\mathtt{Module}_{\mathcal{L}}$ representing theories in the logic $\mathcal{L}$. We can then internalize the representation map Φ as an equationally defined function

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

in a module extending `META-LEVEL`. Then, we can use the functions `meta-reduce`, `meta-apply`, and `meta-rewrite`, or more complex strategies that use such functions, to execute in Maude the metarepresentation $\overline{\Psi}(\overline{T})$ of a theory T in $\mathcal{L}$. In other words, we can in this way *execute* $\mathcal{L}$ in Maude.

But we need more. To build a usable formal tool we need to build an *environment* for it, including not only the execution aspect just described, but parsing, pretty printing, and input/output. If we had instead considered formal translator generator uses of Maude—that is, translations of the form $\Phi : \mathcal{L} \rightarrow \mathcal{L}'$ between two logics, that can be reified in Maude as functions $\bar{\Phi} : \texttt{Module}_{\mathcal{L}} \rightarrow \texttt{Module}_{\mathcal{L}'}$—we would have observed entirely similar needs, since we need to get the specifications in different logics—originating from, or going to, different tools—in and out of Maude by appropriate parsing, pretty printing, and input-output functions. In Maude, these additional metalanguage features are supported as follows:

- The *syntax definition* for $\mathcal{L}$ is accomplished by defining the data type $\texttt{Grammar}_{\mathcal{L}}$. In Maude this can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax of an existing tool supporting $\mathcal{L}$.

- Particularities at the *lexical* level of $\mathcal{L}$ can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier to the language in question (see Section 2.5). This is very useful when the theories in $\mathcal{L}$ have user-definable syntax, since then we can use such bubble sorts to represent strings of symbols corresponding to expressions that can only be analyzed after the user-defined syntax has been analyzed.

- Parsing and pretty printing for $\mathcal{L}$ is accomplished by the `meta-parse` and `meta-pretty-print` functions in `META-LEVEL`, in conjunction with the bubble sorts defined for $\mathcal{L}$ (see Section 2.3.5).

- Input/output of theory definitions, and of commands for execution in $\mathcal{L}$ is accomplished by the predefined module `LOOP-MODE`, that provides a generic read-eval-print loop (see Section 2.3.6).

As we shall see in Section 2.3.6, given any input enclosed in parentheses, the read-eval-print loop gives us access to a list of quoted identifiers $\mathcal{I}$ obtained automatically by putting a quote in front of each of the identifiers in the input. Similarly, any list of quoted identifiers placed in the "output channel" of the loop will be printed in the terminal after removing the quotes. Calling the function `meta-parse` with the input $\mathcal{I}$ and the metarepresentation of the signature $\texttt{Grammar}_{\mathcal{L}}$ in which we want to parse it, if there is a parse, we get the corresponding *parse tree* as a term $\mathcal{T}_{\mathcal{I}}$ of sort `Term`. The reverse process is accomplished by the `meta-pretty-print` function, which takes a term of sort `Term`, and returns its representation as a list of quoted identifiers.

This term $\mathcal{T}_{\mathcal{I}}$ can be seen as the representation of the input $\mathcal{I}$ as a term of sort `Term`. We could manipulate this term directly, applying any transformation to it and obtaining some other term representing the desired result, or we could apply some function to it to perform some kind of analysis or computation. Our experience, however, is that it is simpler and more flexible to transform this term into another term in some data type more appropriate for representing the input.

Suppose, for example, that we want to specify the transformation from modules in some language or logic $\mathcal{L}$ into modules in some language or logic $\mathcal{L}'$. First, we define abstract data types $\texttt{Module}_{\mathcal{L}}$ and $\texttt{Module}_{\mathcal{L}'}$. Then, we equationally define a function

$$\Lambda\colon \texttt{Term} \longrightarrow \texttt{Module}_{\mathcal{L}}$$

returning a term of sort $\texttt{Module}_{\mathcal{L}}$ for each term $\mathcal{T}_{\mathcal{I}}$ of sort `Term` corresponding to an input $\mathcal{I}$. Then, we can compose this function with the also equationally defined function

$$\overline{\Phi}\colon \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}_{\mathcal{L}'}$$

reifying a map of logics of the form

$$\Phi\colon \mathcal{L} \longrightarrow \mathcal{L}'.$$

The whole picture can be completed with a user-defined `meta-pretty-print` function extending the predefined one.

$$\texttt{QidList} \xrightarrow[\texttt{Grammar}_{\mathcal{L}}]{\texttt{meta-parse}} \texttt{Term} \xrightarrow{\Lambda} \texttt{Module}_{\mathcal{L}} \xrightarrow{\overline{\Phi}} \texttt{Module}_{\mathcal{L}'} \xrightarrow[\texttt{Grammar}_{\mathcal{L}}]{\texttt{meta-pretty-print}} \texttt{QidList}$$

In the case of an *executable* environment, this picture may have to be extended to meet additional requirements. In the case of Full Maude, for example, we do not only want to apply some transformation on some particular input, or to just give some command or module expression with some particular set of arguments and get some result. We also want to be able to interact with the system, entering modules, theories, views, and commands of different types.

In addition, such modules can be highly structured and parameterized. That is, there can be a rich collection of module composition operations endowing $\mathcal{L}$ with a *module algebra*. In such cases we typically have two data types of modules, a data type $\texttt{Module}_{\mathcal{L}}$ of *flat* or unstructured modules, and a more general data type $\texttt{StrModule}_{\mathcal{L}}$ of structured modules.

All the module algebra operations for $\mathcal{L}$ can be defined within Maude as an extension of the module `META-LEVEL`. In this way, the environment that we can build for $\mathcal{L}$ using Maude can also support all the module composition operations of $\mathcal{L}$. Among those module operations a common and important one is *flattening*, that is, the process of passing from a structured module to its unstructured flat form. This can be understood as a function

$$\texttt{StrModule}_{\mathcal{L}} \xrightarrow{(\_)^{\flat}} \texttt{Module}_{\mathcal{L}}.$$

Since modularity constructs can change from language to language, it may be simpler to represent $\mathcal{L}$ in rewriting logic by representing only its flat modules, that is, by a function $\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}$ which makes the language $\mathcal{L}$ executable on top of Maude. But using the function $(\_)^{\flat}$ we can also make structured modules in $\mathcal{L}$ executable by means of the function composition

$$\texttt{StrModule}_{\mathcal{L}} \xrightarrow{(\_)^{\flat}} \texttt{Module}_{\mathcal{L}} \xrightarrow{\overline{\Phi}} \texttt{Module}.$$

We may also want to be able to *refer by name* to modules, theories, and views in any other module, theory, view, or command. In Full Maude, for example, modules can be parameterized by theories, and both of them can import other modules or theories, or combinations of them given by *module expressions*. A module importing some module expressions can be seen as denoting a structured module with more or less complex relationships among its component submodules. We want to be able to store modules, theories, and views in a *database*, so that they can be referred to later by name in order to evaluate module expressions, evaluate commands, and so on. Given a data type `Database` for the database of modules, theories, and views, the processing of a term obtained from a call to `meta-parse` with an input corresponding, for example, to a module can be seen as a function

$$\texttt{Term} \times \texttt{Database} \xrightarrow{\texttt{processModule}} \texttt{Database}.$$

In the case of commands, we can see their processing as the following function.

$$\texttt{Term} \times \texttt{Database} \xrightarrow{\texttt{processCommand}} \texttt{Database} \times \texttt{QidList}.$$

Note that some module, theory, or view in the database can be referred to in a command. There might be commands changing the database itself, for example, a command to remove a module from the database. The execution of a command may also generate some output, which will have to be conveniently meta-pretty-printed, so that it can be passed to the loop.

1.2 The Full Maude Execution Environment

In the case of Full Maude, there are several data types and several transformations taking place in the system. The main data types are: `Unit`, for the different types of theories and modules; `View` for views; and `Database` for the database of module, theory, and view definitions. We also have subsorts of `Unit` for the different types of theories and modules. For example, we have sorts `StrOModule` and `StrSModule` for structured object-oriented modules and structured system modules, respectively, and the transformation from object-oriented modules to system modules, described in [14, Section 3.2.2] and in [26, Section 3.3.2], is reified as a function

```
objectOrientedToSystemModule : StrOModule Database -> StrSModule
```

We discuss the data types `Unit`, `View`, and `Database` in Sections 4, 5, and 6, respectively, and the details about the function `objectOrientedToSystemModule` in Section 7.6.

To be able to *execute* some command using functions such as `meta-rewrite` or `meta-reduce`, or to be able to use any of the other predefined functions in the `META-LEVEL` module on the modules entered to Full Maude, we need to map terms of sort `Unit` into terms of sort `Module`, which will then be compiled into the rewrite engine. Note, however, that terms of sort `Unit` are structured modules, where the structure is given by module expressions that refer by name to other modules, or combinations of modules, which are stored in the database. This function in fact takes a unit and a database and returns the database after including the resulting module in it.

```
evalUnit : Unit Database -> Database
```

Terms of sort `Module` can only contain importation declarations of predefined modules, or of modules entered at the object level of Core Maude. Therefore, structured modules have to be "partially flattened," leaving only importations of built-ins[2]. In our approach, transforming a module from its possibly complex structured version to its unstructured form is a two-step process. First, the evaluation of the module expressions results in an intermediate form, in which only simple inclusion relationships appear among the modules; this first step can be seen as the reduction of a structured specification to its *structured normal form*. The normalization of the structure is carried out by the `normalization` function. It takes place on

[2]Although still partially structured, we shall refer to this "partially flattened" module as a module in *unstructured form* or in *flattened form*.

the database, and gives as result a possibly modified version of the database state. Then, in a second step, this structured normal form is flattened into an unstructured specification. Note that, instead of flattening the modules before evaluating the expressions, as it is done in many systems, we prefer to define operations on structured modules that return new structured modules. Besides simplicity and extensibility, the main advantage of dealing with structured modules is that the sharing of sub-structures is made explicit, avoiding the introduction of duplicated modules, or the cost of checking for it. We explain this process in detail in Section 7.

Since units can contain "bubbles," corresponding to terms which have to be parsed using the signature of the unit itself, the function Λ mentioned in Section 1.1 can be split in three stages. As we shall see in Section 2.5, to be able to parse the bubbles in a module, we first need, in a first stage, to extract the signature. Then, in a second stage, a function Λ_1 replaces each bubble by its parsed term using the extracted signature. Then, in a third stage, we map the resulting term of sort `Term` without bubbles to a term of sort $\texttt{Module}_{\mathcal{L}}$ using a function Λ_2.

$$\texttt{Term} \xrightarrow{(1_{\texttt{Term}}\ \times\ \texttt{getSignature})} \texttt{Term} \times \texttt{Sign}_{\mathcal{L}} \xrightarrow{\Lambda_1} \texttt{Term} \xrightarrow{\Lambda_2} \texttt{Module}_{\mathcal{L}}$$

However, this process is computationally quite expensive. Note that building the signature can be almost as expensive as building the whole module. We shall see in detail in Section 7.7 how this process is efficiently accomplished. The idea is to build at the same time a term of sort $\texttt{Module}_{\mathcal{L}}$ with bubbles, which we call a *premodule*, and the signature, using finally the signature to parse the bubbles in the premodule and returning the module without bubbles in it. We can see the process as the functional composition

$$\texttt{Term} \longrightarrow \texttt{Sign}_{\mathcal{L}} \times \texttt{PreModule}_{\mathcal{L}} \longrightarrow \texttt{Module}_{\mathcal{L}}.$$

Notice that in the case of Full Maude we are dealing with structured modules, which are used to construct the signature and to evaluate a module expression in the database. In Full Maude, this process is accomplished by the `evalPreUnit` function. We will discuss it in Section 7.7 as well. In fact, we shall see that the functions `evalUnit` and `evalPreUnit` are quite similar.

The rest of the document is organized as follows. In Section 2 we give an overview of the Maude system, focusing on its most relevant aspects for our present purposes. For a detailed presentation of Maude see [14], and for a description of Full Maude see [14] or [26]. Section 3 presents the way in which the signature of Full Maude (discussed in [14, Chapter 3] and [26, Chapter 3], and given in Appendix A) is related to the rest of the specification. Sections 4.5, 5.3, and 6.2 explain, respectively, the data types `Unit`, `View`, and `Database`. The evaluation of units and views is described, respectively, in Sections 7 and 8. In Section 9 we discuss the processing of the input/output of the system, and in Section 10.1 we describe the top-level handling of the persistent database and its interaction with the read-eval-print loop presented in Section 2.3.6. Finally, in Section 11 we illustrate the flexibility and extensibility of the Full Maude system by introducing some new module expressions. Another extension of Full Maude to a theorem-proving tool for checking the Church-Rosser property is discussed in detail in [26, Chapter 5].

We do not intend to present here all the details of the executable specification of Full Maude. We present an overview of the whole system, focusing on its most important aspects and its main functionality. We present the different modules with their main declarations so that a general idea of the whole system can be gained. Declarations of operators not relevant for the general discussion, as well as equations describing the semantics of some functions and some rules, are omitted. We sometimes add dots "`...`" to indicate the missing equations and declarations.

All the declarations of the variables used are presented. However, some explanation of the conventions followed may ease the readability of the specification. We have followed very simple conventions for naming variables, sorts, and operators. As for the predefined modules, sort names are given starting with a capital letter. When the sort name is the concatenation of different words, or parts of words, each of the parts starts with a capital letter. Thus, we have, for example, sorts `UnitList` and `SubclassDecl` for lists of units and for declarations of subclass relations, respectively. Variables are named using the capital letters of the names of the sorts in which they are defined. For example, we have variables `B` of sort `Bool` or `VDS` of sort `VarDeclSet`. However, we have made some exceptions to this rule when there was a possibility of confusion, or seemed more natural. For example, `QI`, `F`, `V`, and `L` are used as names of variables of sort `Qid` when they are used as any quoted identifier, as an operator name, as a variable name, or as a label, respectively, and there are variables `Eq`, `Rl`, and `AtS` for elements of sorts `Equation`, `Rule`, and `AttrSet`, respectively.

2 The Maude Language

Maude [14, 16, 20] is a high-level reflective language and high-performance system supporting both membership equational logic [48, 5] and rewriting logic [42, 50, 35] specification and programming for a wide range of applications. Maude has been influenced in important ways by the OBJ3 language [32], which can be regarded as an equational logic sublanguage.

Rewriting logic [42, 50, 35] 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 [42, 46, 50, 8]. 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 [40, 39, 36, 37, 3, 57, 10, 22, 11, 17]. Moreover, object-oriented design languages, architectural description languages, and languages for distributed components also have a natural semantics in rewriting logic [61, 51, 58, 53, 54]. 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 present document describes in detail one such metalanguage application, namely the extension of Core Maude to Full Maude in Maude.

A key feature distinguishing Maude from other languages based on rewriting logic, such as ELAN [36, 4] and CafeOBJ [29, 24], is its systematic and efficient use of *reflection* [13, 15]—exploiting the fact that rewriting logic is reflective [21, 11]. This feature makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

The largest Maude application developed so far is Full Maude (see [14] and [26, Chapter 3]) whose executable specification is described in this document. Three other substantial applications are: an inductive theorem prover [17]; a Church-Rosser checker [17, 26] for equational theories, both part of a formal environment for Maude and for the CafeOBJ language; and a formal translator transforming theories in the HOL logic into Nuprl theories. 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 [45, 50, 35, 19]).

2.1 The Logical Basis of Maude

Maude's *functional modules* are theories in *membership equational logic* [49, 6], 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 [31], 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 [6].

Membership equational logic is a sublogic of *rewriting logic* [43]. 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 [47], and a general *logical framework* to represent and execute different logics [41].

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* [12] 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 [33] as well as *object-oriented modules* with convenient syntax for object-oriented applications.

2.2 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* [33], 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 (see Section 11).

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.

This document does *not* explain and illustrate in detail the features and constructs of the Full Maude language. In fact, it assumes basic familiarity with such features and constructs, which are explained in detail in [14, Chapter 3] and in [26, Chapter 3]. What we explain here is how Full Maude is specified and executed in Core Maude.

2.3 Core Maude 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. There are modules `BOOL`, `MACHINE-INT`, `QID`, `QID-LIST`, `META-LEVEL`, and `LOOP-MODE`. Predefined modules are explained in detail in [14].

The module `BOOL` defines the Boolean values and some predefined operators, like `_==_` and `_=/=_`, for equality and inequality, respectively, `if_then_else_fi`, and some of the usual Boolean predicates, like conjunction, disjunction, etc. The module `MACHINE-INT` provides sorts `MachineInt` and `NzMachineInt` for built-in integers and nonzero integers, respectively, and usual arithmetic operations on them. Built-in quoted identifiers in the LISP style are provided by the sort `Qid` in the `QID` module. This module also includes some built-in operations on quoted identifiers, as concatenation (`conc`) or indexing of a quoted identifier by a machine integer (`index`). In the rest of this section we explain in some detail the `META-LEVEL` module and the `LOOP-MODE` module, which are both used in essential ways in the specfication of Full Maude..

2.3.1 The Module `META-LEVEL`

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective. The functional module `META-LEVEL` efficiently implements key functionality of the universal theory $\mathcal{U}$. For example, it provides the following functionality.

- 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 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.

2.3.2 Representing Terms

Terms are reified as elements of the data type `Term` of terms, with the following signature[3]

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

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

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

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

The last declaration for the data type of terms is a constant `error*` to be used as an error element.

[3]The module `META-LEVEL` also includes constructors to represent sort tests and lazy sort tests that we do not present here.

2.3.3 Representing Modules

Functional and system modules are metarepresented 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 metarepresented as we have already explained; (2) in the metarepresentation 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 `_;_`.

Besides sorts `Term` and `Module`, the module `META-LEVEL` includes other sorts whose terms represent items appearing in a module or auxiliary data. There are sorts `Sort` (for sort identifiers[4]), `Attr` (for operator attributes) and `Assignment` (assignments of terms to variables in substitutions); sorts for the declaration of submodule inclusions (`Import`), sorts (`SortDecl`), subsort relations (`SubsortDecl`), operations (`OpDecl`), variables (`VarDecl`), equations (`Equation`), rules (`Rule`), and membership axioms (`MembAx`); sorts for sets[5] or lists of some of such declarations (`ImportList`, `SubsortDeclSet`, `OpDeclSet`, `VarDeclSet`, `EquationSet`, `RuleSet`, `MembAxSet`); and some other sorts, such as `QidList` or `Substitution`.

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 .
```

Sort declarations are represented using a constructor

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

where `QidSet` is the sort of sets of quoted identifiers:

```
subsort Qid < QidSet .
op none : -> QidSet .
op _;_ : QidSet QidSet -> QidSet [assoc comm id: none] .
```

Subsort declarations are represented with the syntax:

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

`META-LEVEL` also includes similar syntax declarations for representing the other elements that can appear in a module. Without going into all the details, let us for example focus in the representation of operator declarations. We have already mentioned the sorts `OpDecl` and `OpDeclSet`, which provide declarations of operators and sets of declarations of operators, respectively. Each operator declaration is given by a term of sort `OpDecl` using a constructor

[4] User defined sorts are of sort `Qid`. The sort `Sort` is declared as a supersort of `Qid`.

[5] Although they are called "sets", actually they are "multisets" because there is missing the idempotence attribute or equation.

```
op op_:_->_[_]. : Qid QidList Qid AttrSet -> OpDecl .
```

The sort `AttrSet` is declared as a supersort of `Attr`, and with a union operator `__`. The sort `Attr` is used for representing operator attributes, including attributes that are equational axioms used to rewrite modulo. For example, the commutativity attribute is represented by the constant

```
op comm : -> Attr .
```

Thus, the declaration of a commutative natural number addition operator can be represented by the term

```
op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
```

Note that names of sorts and operators (and also those of variables) are represented in `META-LEVEL` as quoted identifiers. For example, a "mixfix" operator like `_+_` is represented by the corresponding quoted identifier `'_+_`, with underbar characters indicating the argument positions.

If there are no operator declarations in a particular module, then the corresponding argument will be `none`. Otherwise, the set is given by the associative constructor `__`, which is declared also with identity element `none`.

```
op none : -> OpDeclSet .
op __ : OpDeclSet OpDeclSet -> OpDeclSet [assoc comm id: none] .
```

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 a commutative addition operator.

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

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

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

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 module `META-LEVEL` 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 Maude syntax, which is mirrored quite closely.

Note that—just as in the case of terms—terms of sort `Module` can be metarepresented 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 [17], where modules are metarepresented as terms in the inference rules for induction, but they have to be meta-metarepresented as terms of sort `Term` when used in strategies that control the application of the inductive inference rules.

2.3.4 Descent Functions

The descent functions [13, 14] provide useful and efficient ways of reducing metalevel computations to object-level ones. We explain here only two of the built-in descent functions in the module `META-LEVEL`, namely `meta-reduce` and `meta-rewrite`.

The function `meta-reduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module. It has syntax[6]

```
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. Such a metarepresentation has to be explicitly given, that is, considering NAT as the module given above, the previous example 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] .
                 var 'N : 'Nat .
                 var 'M : 'Nat .
                 none
                 eq '_+_[{'0}'Nat, 'N] = 'N .
                 eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
              endfm,
              '_+_['s_[{'0}'Nat], {'0}'Nat]) .
result Term: 's_[{'0}'Zero]
```

[6] We do not give the arguments of the `special` attributes; we use '`...`' instead.

[14, Section 3.3] and [26, Section 3.4] discuss how we can extend the Maude system using its metaprogramming capabilities to be able to refer to the metarepresentation of modules previously entered in the system. We discuss in Section 2.5.1 some alternatives to do it in Core Maude.

The function `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.

2.3.5 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 metarepresentation 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` above and the input `'s '0 '+ '0`, we get the following result.

```
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`, 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
```

The module `META-LEVEL` also provides some commonly needed functions on the poset of sorts of a module in a built-in way. For example, 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. Its syntax is as follows.

```
op leastSort : Module Term -> Sort [special (...)] .
```

Since it can return an error sort not defined by the user its result sort is `Sort`. For example, we can compute the least sort of the term $\overline{\texttt{N + s M}}$ in the module `NAT` as follows.

`Maude> red least-sort(`$\overline{\texttt{NAT}}$`,` $\overline{\texttt{N + s M}}$`) .`
`result Qid:` $\overline{\texttt{Nat}}$

2.3.6 The LOOP-MODE Module

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 the release 1.00 of Maude that may not be maintained in future versions. Maude will be endowed with general built-in support for objects in the future; 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 persistent state and the rewrite rules for changing that state and interacting with 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. 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 following output:

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

A somewhat more interesting example is a loop that echoes the input, but storing the last ten tokens, that is, it prints one token only if the number of tokens stored in the state is bigger than ten. 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—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 .
  var QI : Qid .
  var Counter : MachineInt .
  rl [init] : init => [nil, < nil ; 0 >, nil] .
  rl [in] :
     [QI Input, < StoredInput ; Counter >, Output]
     => [Input, < StoredInput QI ; Counter + 1 >, Output] .
  crl [out] :
     [Input, < QI StoredInput ; Counter >, Output]
     => [Input, < StoredInput ; Counter - 1 >, Output QI]
        if Counter > 10 .
endm
```

A possible interaction with the system can then be given as follows:

```
Maude> loop init .
Maude> (a b)
Maude> (c d e f g h i)
Maude> (j k l)
a b
```

The above examples are toy examples to illustrate the basic features of the module 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.

2.4 Metalanguage Uses of Maude

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* [39] 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

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

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

By being reflective, a rewriting logic language as Maude supports 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.

As already explained in Section 2.3.1, 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 $\texttt{Module}_{\mathcal{L}}$ 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

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

In fact, thanks to the general metaresult of Bergstra and Tucker [2], 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 [40, 11], and for structured Maude modules in Full Maude as explained in [14], [26, Chapter 3], and in the present paper. 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. The `LOOP-MODE` module introduced in Section 2.3 makes this 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.

2.5 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 defining a data type $\texttt{Grammar}_{\mathcal{L}}$ for $\mathcal{L}$-modules, and with auxiliary declarations for commands and other constructs. Maude provides great flexibility to do this thanks to its mixfix front-end and to the use of bubbles[7] (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 Maude parser (MSCP [56]) provides a basis for this flexible syntax definition, and an efficient treatment of what might be called *syntactic reflection*. MSCP is implemented using SCP as its formal kernel [55].

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, 1, 34, 38].

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.

[7] In the release 1.00 of Maude 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 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)
```

in its constructor declaration 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.

```
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)`. Note the form of the `id-hook` special attribute now. The symbols '`(`' and '`)`' in it indicate that parentheses have to be balanced inside a bubble. We could use any other pair of symbols instead of parentheses in the same way.

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 we give the following declarations to specify the syntax of a list of operator declarations in a module:

```
sorts Decl DeclList .
subsort Decl DeclList .
op op_: ->_. : Token Token -> Decl .
op op_:_->_. : Token Bubble Token -> Decl .
op __ : DeclList DeclList -> DeclList [assoc] .
```

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[8] the identifier '`.`' with the following declarations.

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

```
  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 .
  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] .
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 metarepresentation of the module `MINI-MAUDE-SYNTAX` and the previous module transformed into a list of quoted identifiers[9], 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 `'Nat3` and `'s 's 's '0` are represented, respectively, as `token('Nat3)` and `bubble('s 's 's '0)`. Maude returns them metarepresented. 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? .
```

[9]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`.

Notice that `extractSignature` is a partial function that is not well defined for metaterms of sort `Term` that do not metarepresent 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 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 metarepresenting 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.

```
Maude> 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
```

2.5.1 Extending `META-LEVEL`

From the examples in the previous section one can imagine that giving the metarepresentation of modules when calling functions taking them as arguments can become quite inconvenient. There are two very simple ways of avoiding it:

- One possibility is to declare in a module extending `META-LEVEL` a constant of sort `FModule` or `Module`, and then defining it by an equation identifying such a constant with the metarepresentation of the flattened version of the module we are interested in using. However, the module may be large, and this makes this solution unsatisfactory.

- It is possible to refer in the metarepresentation of a module to built-in modules and to modules previously entered in Core Maude's database. This makes possible a second alternative, namely, entering the module of interest in Core

Maude, and then referring to it in the metarepresentation of another module. Note that in this way we can use the module entered at the object level in calls to built-in functions, but we do not have access to its metarepresentation, that is, we cannot manipulate it as data. For example, we can define modules such as the following.

```
fmod FOO is
  sort Foo .
  op foo : -> Foo .
endfm

fmod BAR is
  including META-LEVEL .
  op M : -> FModule .
  eq M = (fmod 'M is
             including 'FOO .
             sorts none .
             none
             none
             none
             none
             none
          endfm) .
endfm
```

Note that, if desired, we could have added additional declarations to the metarepresentation of the module M. In the context of the BAR module, we can now refer to the metarepresentation of module M, which contains all the declarations in the module FOO, in any command.

Bubble sorts are not supported at the object level in the version 1.00 of Core Maude. The special attributes linking the constructors for the bubble sorts to the built-in ones are only supported at the metalevel, that is, the declarations of the constructor operators for bubble sorts have to be given in the metarepresentation of a module. But this does not mean that we have to give the metarepresentation of modules including them. We can write a module at the object level including all the declarations, except for those containing special attributes for bubble sorts. Such declarations involving special attributes can then be included in the metarepresentation of a module that includes the quoted name for the rest of the module of interest, which can be defined at the object level. We can illustrate these ideas by applying them to the module MINI-MAUDE-SYNTAX above, which can be made available at the metalevel by redefining at the object level the fragment not containing the special attributes, and then importing that fragment—together with the metarepresentation of the operations involving the special attributes—within a module term at the metalevel in the module PARSING.

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

  *** 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] .
endfm

fmod PARSING is
  including META-LEVEL .
  op META-MINI-MAUDE-SYNTAX : -> FModule .
  eq META-MINI-MAUDE-SYNTAX
    = (fmod 'META-MINI-MAUDE-SYNTAX is
          including 'MINI-MAUDE-SYNTAX .
          sorts none .
          none
          op 'token : 'Qid -> 'Token
                [special(
                   (id-hook('Bubble, '1 '1)
                    op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          op 'neTokenList : 'QidList -> 'NeTokenList
                [special(
                   (id-hook('Bubble, '1 '-1 '`( '`))
                    op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                    op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          op 'bubble : 'QidList -> 'Bubble
                [special(
                   (id-hook('Bubble, '1 '-1 '`( '`))
                    op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                    op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          none
          none
          none
       endfm) .
endfm
```

3 The Signature of Full Maude

The signature of Full Maude is given in its entirety in Appendix A, and is discussed in detail in [14, Chapter 3] and in [26, Chapter 3]. We focus here on the way in which the signature is used in the specification, and in the way in which it can be used in future extensions of Full Maude.

As explained in Section 2.5, to parse some input using the built-in function `meta-parse`, we need to give the metarepresentation of the signature in which the input is going to be parsed. But we do not need to give the complete metarepresentation of such a module. As we saw in Section 2.5, in modules including `META-LEVEL` it is possible to define terms of sort `Module` that import built-in modules or any module introduced at the "object level" of Core Maude. In this way, it is possible to get the equivalent effect of having the explicit metarepresentation of a module by declaring a constant and adding an equation identifying such a constant with the metarepresentation of an extended module that imports the original module at the object level.

As also mentioned in Section 2.5, the declaration of constructors for bubble sorts at the object level is not supported in the current version of Core Maude. The `special` attributes linking the constructors for the bubble sorts to the built-in ones are only supported at the metalevel, that is, the declarations of the constructor operators for bubble sorts have to be given in the metarepresentation of a module.

To allow the greatest generality and flexibility in future extensions of Full Maude, we have declared its signature as a module `FULL-MAUDE-SIGN` (see Appendix A). Then, in the following module `META-FULL-MAUDE-SIGN` we declare a constant `GRAMMAR` of sort `FModule`, and using the technique described in Section 2.5 to import the metarepresentation of a module by name, we give an equation identifying such constant with the metarepresentation of a module `GRAMMAR` in which there is a declaration importing `FULL-MAUDE-SIGN`. Declarations for the constructors of the bubble sorts are also included in this module. Note that the bubble sorts `Token`, `Bubble`, `ViewToken`, `SortToken`, and `NeTokenList` are declared in the module `SIGN&VIEW-EXPR`, which is imported by `FULL-MAUDE-SIGN`. These sorts are used in the declarations describing the syntax of the system.

```
fmod META-FULL-MAUDE-SIGN is
  including META-LEVEL .

  op GRAMMAR : -> FModule .

  eq GRAMMAR
    = (fmod 'GRAMMAR is
         including 'QID-LIST .
         including 'FULL-MAUDE-SIGN .
         sorts none .
         none
         op 'token : 'Qid -> 'Token
              [special(
                (id-hook('Bubble, '1 '1)
                 op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
         op 'viewToken : 'Qid -> 'ViewToken
              [special(
                (id-hook('Bubble, '1 '1)
                 op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
                 id-hook('Exclude, 'assoc 'associative 'comm
                                   'commutative 'idem 'idempotent)))] .
         op 'sortToken : 'Qid -> 'SortToken
              [special(
                (id-hook('Bubble, '1 '1)
                 op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
                 id-hook('Exclude, '`[ '`] '< 'to ': '`, '. '`( '`))))] .
         op 'neTokenList : 'QidList -> 'NeTokenList
              [special(
                (id-hook('Bubble, '1 '-1 '`( '`))
                 op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                 op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
         op 'bubble : 'QidList -> 'Bubble
              [special(
                (id-hook('Bubble, '1 '-1 '`( '`))
                 op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                 op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
```

```
        none
        none
        none
     endfm) .
endfm
```

The GRAMMAR module will be used in calls to the meta-parse function in order to get the input parsed in this signature. Notice that this module is not the data type in which we shall represent the inputs. From the call to meta-parse we shall get a term representing the parse tree of the input. This term will then be transformed into terms of other appropriate data types if necessary.

Future extensions to Full Maude will require extending the signature as well. The addition of new commands, new module expressions, or additions of any other kind will require adding new declarations to the present Full Maude signature and defining the corresponding extensions to the data types and functions to deal with the new cases introduced by the extensions. We shall see in Section 11 how the system can be easily extended to handle some new module expressions in addition to the renaming and instantiation module expressions presented in [14, Section 3.5.4] and in [26, Section 3.6.4], and further discussed in Sections 7.10 and 7.9. In [26, Chapter 5] it is explained how to combine a proving tool like a Church-Rosser checker [18, 17] with Full Maude.

4 The Abstract Data Type Unit

In this section we present the abstract data type Unit, which can be seen as an extension of the predefined sort Module in several ways. There are constructors for functional, system, and object-oriented modules and theories, which can be parameterized and can import module expressions. There can also be parameterized sorts in Full Maude modules, and therefore, the constructors for the different declarations that can appear in a module have to be appropriately extended.

The section is structured as follows. After introducing some modules defining some functions on the predefined sorts Bool and QidList in Section 4.1, we present in Sections 4.2.2 and 4.3 the data types for extended sorts and extended declarations. In Section 4.4 we introduce module expressions and module names, and in Section 4.5 the abstract data type Unit itself.

4.1 Extensions to BOOL and QID-LIST

The module EXT-BOOL extends the predefined BOOL module with operations _and-then_ and _or-else_. The semantics of _and-then_ and _or-else_ are, respectively, as the semantics of the boolean operators _and_ and _or_, but they are not declared to be associative, and their second arguments are evaluated lazily thanks to the evaluation strategy (1 0 2 0) (see [32, 28, 14] for more on evaluation strategies).

```
fmod EXT-BOOL is
  pr BOOL .

  op _and-then_ : Bool Bool -> Bool [strat (1 0 2 0) prec 55] .
  op _or-else_ : Bool Bool -> Bool [strat (1 0 2 0) prec 59] .

   ...

endfm
```

The conversion of lists of quoted identifiers into single quoted identifiers by concatenating them is heavily used in the coming modules. This is the task of the `qidListToQid` function, which is introduced in the following module `EXT-QID-LIST`. `EXT-QID-LIST` is defined as an extension of the predefined module `QID-LIST`.

```
fmod EXT-QID-LIST is
  pr QID-LIST .

  op qidListToQid : QidList -> Qid .

    ...

endfm
```

4.2 View Expressions and Extended Sorts

To allow the use of parameterized sorts, or sorts qualified by the view expression with which the parameterized module in which the given sorts appear is instantiated (see [14, Section 3.5.2] or [26, Section 3.6.2]), we add the sort `ESort` of "extended sorts" as a supersort of the predefined sort `Sort`. View expressions and extended sorts are introduced in the following modules.

4.2.1 View Expressions

A view expression is given by a single quoted identifier, by a sequence of view expressions (at the user level, separated by commas), or by the composition of view expressions. In the current version, the composition of view expressions is only used internally; we plan to make it available to the user with syntax `_;_` in the future. View expressions are used in the instantiation of parameterized modules and in parameterized sorts. We plan to support parameterized views in the future as well. We use operators `_|_` and `_;;_` to represent, respectively, sequences and composition of view expressions.

The `VIEW-EXPR` module below also includes declarations for sets of view expressions, which will be used in the database structure to hold the list of the names of the views in it. The elements of sort `ViewExpSet`, for sets of view expressions, are given using the constructor `_#_`, which is declared to be associative, commutative, and with identity element `noneViewExpSet`.

The function `_inViewExpSet_` checks whether the view expression given as first argument is in the set of view expressions given as second argument or not.

```
fmod VIEW-EXPR is
  pr EXT-BOOL .
  pr QID .

  sort ViewExp .
  subsort Qid < ViewExp .
  op nullViewExp : -> ViewExp .
  op _|_ : ViewExp ViewExp -> ViewExp [assoc id: nullViewExp] .
  op _;;_ : ViewExp ViewExp -> ViewExp [assoc] .

  sort ViewExpSet .
  subsort ViewExp < ViewExpSet .
  op noneViewExpSet : -> ViewExpSet .
  op _#_ : ViewExpSet ViewExpSet -> ViewExpSet
        [assoc comm id: noneViewExpSet] .
```

```
  op _inViewExpSet_ : ViewExp ViewExpSet -> Bool .

   ...

endfm
```

Since the Core Maude engine does not know about view expressions, or, as we shall see, about extended sorts, extended module expressions, extended modules, and other declarations that we introduce, to be able to use them with built-in functions such as `sameComponent`, `leastSort`, `meta-reduce`, etc., we shall have to convert them into terms which only use the built-in constructors. Thus, for example, view expressions in sort `ViewExp` will be converted into quoted identifiers of sort `Qid` by means of function `viewExpToQid`, or, similarly, elements of sorts `ESort`, `ESortList`, and `ESortSet` are transformed into elements of sorts `Qid`, `QidList`, and `QidSet`, respectively, with functions `eSortToQid` defined on the appropriate sorts.

```
fmod VIEW-EXPR-TO-QID is
  pr VIEW-EXPR .
  pr EXT-QID-LIST .

  op viewExpToQid : ViewExp -> Qid .
  op viewExpToQidList : ViewExp -> QidList .

   ...

endfm
```

4.2.2 Parameterized Sorts

In addition to the `ESort` sort, in the following module `EXT-SORT` we also define sorts `ESortList` and `ESortSet`.

The operator `eSort` is declared to be a constructor for extended sorts.

As for lists and sets of quoted identifiers, we declare `__` and `_;_` as constructors for sorts `SortList` and `ESortList`, and `ESortSet`, respectively.

```
fmod EXT-SORT is
  pr META-LEVEL .
  pr VIEW-EXPR-TO-QID .

  sort ESort .
  subsort Sort < ESort .
  op eSort : ESort ViewExp -> ESort .

  sort SortList .
  subsort SortList ESort < ESortList .
  op __ : SortList SortList -> SortList [assoc id: nil] .

  sort ESortList .
  subsort QidList Sort < SortList .
  op __ : ESortList ESortList -> ESortList [assoc id: nil] .

  sort ESortSet .
  subsort QidSet ESort < ESortSet .
  op _;_ : ESortSet ESortSet -> ESortSet [assoc comm id: none] .

endfm
```

4.2.3 Transforming Extended Sorts into Quoted Identifiers

Since eventually all units will have to be given as arguments in some of the calls to the built-in functions, and since the engine does not know about parameterized sorts, we need to transform all parameterized sorts appearing in any of the declarations that can appear in a unit into quoted identifiers. This is the task of the `eSortToQid` function, which is defined in the module `EXT-SORT-TO-QID`. These functions will be used with terms not containing error sorts[10]. Although the user cannot use error sorts, calls to built-in functions may generate such error sorts, and some built-in functions can handle them. The function `eSortToSort` presented in the previous module will be used in the calls to the built-in functions that may use such error sorts.

```
fmod EXT-SORT-TO-QID is
  pr EXT-SORT .

  op eSortToQid : ESort -> Qid .
  op eSortToQid : ESortSet -> QidSet .
  op eSortToQid : ESortList -> QidList .

   ...

endfm
```

4.3 Extended Declarations

In this section we discuss modules `EXT-DECL` and `O-O-DECL` which introduce, respectively, the declarations extending the sorts and constructors for declarations of the predefined data type `Module` in the `META-LEVEL` module to allow the use of extended sorts in them, and the declarations appearing in object-oriented units, namely class declarations, subclass relation declarations, and message declarations.

4.3.1 Declarations of Functional and System Units

In the following module `EXT-DECL`, we introduce the declarations extending those in `META-LEVEL` to allow the use of extended sorts in declarations of sorts, subsort relations, operators, variables, and membership axioms. The extension is accomplished by adding new supersorts for each of the sorts in `META-LEVEL` involved, and by adding new constructors for these new sorts.

We start introducing the declarations for the supersorts and their corresponding constructors. The `EXT-DECL` module also contains the declarations for sets of such declarations.

```
fmod EXT-DECL is
  pr EXT-SORT .

  sort ESortDecl .
  subsort SortDecl < ESortDecl .
  op sorts_. : ESortSet -> ESortDecl .
```

[10] The Core Maude system automatically adds 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 [48, 7].

```
  sorts ESubsortDecl ESubsortDeclSet .
  subsort SubsortDecl < ESubsortDecl .
  subsorts SubsortDeclSet ESubsortDecl < ESubsortDeclSet .
  op subsort_<_. : ESort ESort -> ESubsortDecl .
  op __ : ESubsortDeclSet ESubsortDeclSet -> ESubsortDeclSet
       [comm assoc id: none] .

  sorts EOpDecl EOpDeclSet .
  subsort OpDecl < EOpDecl .
  subsorts OpDeclSet EOpDecl < EOpDeclSet .
  op op_:_->_[_]. : Qid ESortList ESort AttrSet -> EOpDecl .
  op __ : EOpDeclSet EOpDeclSet -> EOpDeclSet [assoc comm id: none] .

  sorts EVarDecl EVarDeclSet .
  subsort VarDecl < EVarDecl .
  subsorts VarDeclSet EVarDecl < EVarDeclSet .
  op var_:_. : Qid ESort -> EVarDecl .
  op __ : EVarDeclSet EVarDeclSet -> EVarDeclSet [assoc comm id: none] .

  sorts EMembAx EMembAxSet .
  subsort MembAx < EMembAx .
  subsorts EMembAx MembAxSet < EMembAxSet .
  op mb_:_. : Term ESort -> EMembAx .
  op cmb_:_if_=_. : Term ESort Term Term -> EMembAx .
  op __ : EMembAxSet EMembAxSet -> EMembAxSet[assoc comm id: none] .
```

The `sortSet` function returns the set of sorts from a declaration of sorts.

```
  op sortSet : ESortDecl -> ESortSet .
```

There are also some other functions to deal with variables. The function `varDeclSetInTerm` returns the set of the variable declarations in the given module for those variables that appear in the term given as second argument. The function `varDeclSetDiff` computes the set-theoretic difference between the two sets of variables given as arguments. Note that the difference is not between the declarations themselves, but just between the names of the variables in the declarations; the sorts with which they are declared are not taken into consideration.

```
  op varDeclSetInTerm : EVarDeclSet Term -> EVarDeclSet .

  op varDeclSetDiff : EVarDeclSet EVarDeclSet -> EVarDeclSet .

   ...

endfm
```

4.3.2 Declarations for Object-Oriented Units

In the `O-O-DECL` module we introduce the sorts and constructors for declarations of classes, subclass relations, and messages in object-oriented units.

Note that we follow the same naming conventions for classes as for extended sorts (see [14, Section 3.5.2] or [26, Section 3.6.2]), and therefore we use the sort `ESort` for class identifiers, and `ESortList` and `ESortSet` for lists and sets of class identifiers, respectively. We use the operator `attr_:_` as a constructor for declarations of attributes.

```
fmod O-O-DECL is
  pr EXT-SORT .

  sorts AttrDecl AttrDeclSet .
  subsort AttrDecl < AttrDeclSet .
  op attr_:_ : Qid ESort -> AttrDecl .
  op none : -> AttrDeclSet .
  op _,_ : AttrDeclSet AttrDeclSet -> AttrDeclSet [assoc comm id: none] .

  sorts ClassDecl ClassDeclSet .
  subsort ClassDecl < ClassDeclSet .
  op class_|_. : ESort AttrDeclSet -> ClassDecl .
  op none : -> ClassDeclSet .
  op __ : ClassDeclSet ClassDeclSet -> ClassDeclSet [assoc comm id: none] .

  sorts SubclassDecl SubclassDeclSet .
  subsort SubclassDecl < SubclassDeclSet .
  op subclass_<_. : ESort ESort -> SubclassDecl .
  op none : -> SubclassDeclSet .
  op __ : SubclassDeclSet SubclassDeclSet -> SubclassDeclSet
        [assoc comm id: none] .

  sorts MsgDecl MsgDeclSet .
  subsort MsgDecl < MsgDeclSet .
  op msg_:_->_. : Qid ESortList ESort -> MsgDecl .
  op none : -> MsgDeclSet .
  op __ : MsgDeclSet MsgDeclSet -> MsgDeclSet [assoc comm id: none] .

   ...

endfm
```

4.3.3 'Desugaring' of Extended Sorts in Declarations

In the following module we define the `eSortToQid` function on the sorts for the different declarations introduced in the previous modules.

```
fmod DECL-EXT-SORT-TO-QID is
  pr EXT-SORT-TO-QID .
  pr EXT-DECL .
  pr O-O-DECL .

  op eSortToQid : ESubsortDeclSet -> SubsortDeclSet .
  op eSortToQid : EOpDeclSet -> OpDeclSet .
  op eSortToQid : EVarDeclSet -> VarDeclSet .
  op eSortToQid : EMembAxSet -> MembAxSet .
  op eSortToQid : ClassDeclSet -> ClassDeclSet .
  op eSortToQid : SubclassDeclSet -> SubclassDeclSet .
  op eSortToQid : MsgDeclSet -> MsgDeclSet .
  op eSortToQid : AttrDeclSet -> AttrDeclSet .

   ...

endfm
```

4.4 Module Expressions and Module Names

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. All module expressions will be evaluated generating modules with such module expressions as names. In the case of parameterized modules, each of the parameters in an interface will be used as the name of a new module created as a renamed copy of the parameter theory.

4.4.1 Module Expressions

In the following `MOD-EXPR` module the sort `ModExp` is introduced as a supersort of `Qid`. Module expressions for module renaming and for module instantiation will be introduced, respectively, in the modules `RENAMING-EXPR` and `INST-EXPR`, in Sections 7.10 and 7.9, which will be defined as extensions of `MOD-EXPR`. We shall see in Section 11 how new module combining and/or transforming operators can be easily added, showing the flexibility and extensibility of the module algebra being defined.

```
fmod MOD-EXPR is
  pr QID .

  sort ModExp .
  subsort Qid < ModExp .

endfm
```

4.4.2 Module Names

As we shall see in the coming sections, the evaluation of module expressions may produce the creation of new modules, whose *names* are given by the module expressions themselves. If there is already a module in the database with the module expression being evaluated as name, the evaluation of such module expression does not produce any change in the database. However, the evaluation of a module expression may involve the evaluation of some other module expressions contained in the modules involved, which in turn may generate new modules.

Given a parameterized module $\mathtt{N[L_1\ ::\ T_1,\ \ldots,\ L_n\ ::\ T_n]}$, with $\mathtt{L}_1 \ldots \mathtt{L}_n$ labels and $\mathtt{T}_1 \ldots \mathtt{T}_n$ theory identifiers, we say that `N` is the name of the module and that $\mathtt{[L_1\ ::\ T_1,\ \ldots,\ L_n\ ::\ T_n]}$ is its *interface*. As we shall see in Sections 7.9 and 9.3.1, for each parameter $\mathtt{L}_i$ `::` $\mathtt{T}_i$ in the interface of a module, a new module is generated with such a parameter expression as its name, and a declaration importing it in the parameterized module is added. We regard the relationship between the body of a parameterized module and the parameters in its interface, not as an inclusion, but as mediated by a module constructor that generates renamed copies of the parameters, which are then included. Therefore, the sort `Parameter` is declared as a subsort of `ModName`, that is, terms of sort `Parameter` are considered to be module names. The constructor operator for the sort `Parameter` is `par_::_`.

The `MOD-NAME` module also includes a sort `ModNameSet`, for sets of module names. The constructors of sort `ModNameSet` are `noneModNameSet` and `_&_`. The function `_inModNameSet_` checks whether the module name given as first argument is in the set of module names given as second argument.

```
fmod MOD-NAME is
```

```
  pr MOD-EXPR .
  pr EXT-BOOL .

  sort Parameter .
  op par_::_ : Qid ModExp -> Parameter .

  sorts ModName ModNameSet .
  subsorts Parameter ModExp < ModName < ModNameSet .
  op noneModNameSet : -> ModNameSet .
  op _&_ : ModNameSet ModNameSet -> ModNameSet
        [assoc comm id: noneModNameSet] .
  op nullModName : -> ModName .

  op _inModNameSet_ : ModName ModNameSet -> Bool .

   ...

endfm
```

Since the Core Maude engine assumes that module names are identifiers and does not know about term-structured module names (such as parameterized module interfaces or module expressions), for evaluation purposes we need to transform them into quoted identifiers. The functions `modNameToQid` and `modNameToQidList` in the module `MOD-NAME-TO-QID` below accomplish this transformation. In any language extensions, new equations for the function `modNameToQidList` should be added for each new module expression constructor introduced. In Sections 7.10 and 7.9 we shall see how the corresponding equalities are added for renaming and instantiation expressions, and in Section 11 for other new module expressions in extensions of Full Maude.

```
fmod MOD-NAME-TO-QID is
  pr MOD-NAME .
  pr EXT-QID-LIST .

  op modNameToQid : ModName -> Qid .
  op modNameToQidList : ModName -> QidList .

  vars QI X : Qid .
  var ME : ModExp .
  var MN : ModName .

  eq modNameToQidList(QI) = QI .
  eq modNameToQidList(par X :: ME) = (X ':: modNameToQidList(ME)) .
  eq modNameToQidList(nullModName) = ' .

  eq modNameToQid(MN) = qidListToQid(modNameToQidList(MN)) .

endfm
```

4.5 Units

We handle six different types of units: functional, system, and object-oriented modules, and functional, system, and object-oriented theories. Modules and theories of any kind are considered to be elements in specific subsorts of the sort `Unit`. A constructor `unitError` is also included to represent incorrect units. `unitError` has a list of quoted identifiers as argument, which is used to report the error. Besides considering functional and system theories and object-oriented theories and modules, the declarations presented in the following module extend the declarations for sort `Module` in the `META-LEVEL` module in three different ways:

- the name of a module can be any term of sort `ModName`,
- parameterized modules are handled, for which a list of parameters is added to the constructors of modules,
- the importation declaration is extended to module names, and
- parameterized sorts are supported.

```
fmod UNIT is
  pr EXT-DECL .
  pr O-O-DECL .
  pr MOD-NAME-TO-QID .
  inc META-LEVEL .
```

We start by introducing declarations for sorts `EImport` and `EImportList`, which are declared as supersorts of `Import` and `ImportList`, respectively. We declare the constructor `including_.` for importation declarations on module names. We shall see in Section 9.1.1 that importations in both `protecting` and `including` modes are represented using this constructor. The sort for lists of parameters is `ParameterList`, which is defined with constructors `parameterList` and `nilParameterList`.

```
  sorts EImport EImportList .
  subsort Import < EImport  .
  subsorts ImportList EImport < EImportList .
  op including_. : ModName -> EImport .
  op __ : EImportList EImportList -> EImportList [assoc id: nil] .

  sort ParameterList .
  subsort Parameter < ParameterList .
  op nilParameterList : -> ParameterList .
  op parameterList : ParameterList ParameterList -> ParameterList
        [assoc id: nilParameterList] .
```

Next, we introduce the different sorts for the different types of modules and theories, with the subsort relation among them, and their constructors. The structure of the hierarchy of sorts can be seen in Figure 1.

```
  sorts Unit FUnit SUnit OUnit
        StrFModule StrSModule StrOModule StrModule
        StrFTheory StrSTheory StrOTheory StrTheory .
  subsorts FModule < StrFModule < FUnit StrSModule .
  subsorts Module < StrSModule < SUnit StrOModule .
  subsorts StrFTheory < StrSTheory FUnit .
  subsorts StrSTheory < StrOTheory SUnit .
  subsort FUnit < SUnit .
  subsort StrOTheory < StrTheory .
  subsorts StrOTheory SUnit StrOModule < OUnit .
  subsort StrOModule < StrModule .
  subsorts StrTheory OUnit StrModule < Unit .

  op noModule : -> Module .
  op unitError : QidList -> Unit .
```

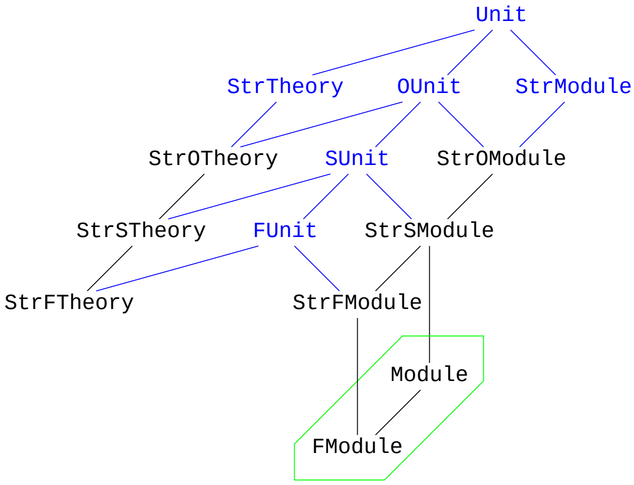

Figure 1: Hierarchy of Unit Sorts.

```
op fmod_is________endfm :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet -> StrFModule
      [gather (& & & & & & & & &)] .
op fth_is________endfth :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet -> StrFTheory
      [gather (& & & & & & & & &)] .
op mod_is_________endm :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet RuleSet -> StrSModule
      [gather (& & & & & & & & & &)] .
op th_is_________endth :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet RuleSet -> StrSTheory
      [gather (& & & & & & & & & &)] .
op omod_is____________endom :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      ClassDeclSet SubclassDeclSet EOpDeclSet MsgDeclSet EVarDeclSet
      EMembAxSet EquationSet RuleSet -> StrOModule
      [gather (& & & & & & & & & & & & &)] .
op oth_is____________endoth :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      ClassDeclSet SubclassDeclSet EOpDeclSet MsgDeclSet EVarDeclSet
      EMembAxSet EquationSet RuleSet -> StrOTheory
      [gather (& & & & & & & & & & & & &)] .
```

In addition to the constructor operators, the following functions are introduced in the UNIT module:

- A function `_in_` to check whether a given importation declaration is in a set of importation declarations or not.

  ```
  op _in_ : EImport EImportList -> Bool .
  ```

- Selector functions for the different components of a unit.

```
op name : Unit -> ModName .
op importList : Unit -> EImportList .
op parameterList : Unit -> ParameterList .
op sortDecl : Unit -> ESortDecl .
op subsortDeclSet : Unit -> ESubsortDeclSet .
op opDeclSet : Unit -> EOpDeclSet .
op varDeclSet : Unit -> EVarDeclSet .
op membAxSet : Unit -> EMembAxSet .
op equationSet : Unit -> EquationSet .
op ruleSet : Unit -> RuleSet .
op classDeclSet : Unit -> ClassDeclSet .
op subclassDeclSet : Unit -> SubclassDeclSet .
op msgDeclSet : Unit -> MsgDeclSet .
```

- Functions to change the value of each of the components of a unit.

```
op setName : Unit ModName -> Unit .
op setParameterList : Unit ParameterList -> Unit .
op setImportList : Unit EImportList -> Unit .
op setSortDecl : Unit ESortDecl -> Unit .
op setSubsortDeclSet : Unit ESubsortDeclSet -> Unit .
op setOpDeclSet : Unit EOpDeclSet -> Unit .
op setVarDeclSet : Unit EVarDeclSet -> Unit .
op setMembAxSet : Unit EMembAxSet -> Unit .
op setEquationSet : Unit EquationSet -> Unit .
op setRuleSet : Unit RuleSet -> Unit .
op setClassDeclSet : Unit ClassDeclSet -> Unit .
op setSubclassDeclSet : Unit SubclassDeclSet -> Unit .
op setMsgDeclSet : Unit MsgDeclSet -> Unit .
```

- Functions to add new declarations to the set of declarations already in a unit.

```
op addImportList : EImportList Unit -> Unit .
op addSortSet : ESortSet Unit -> Unit .
op addSubsortDeclSet : ESubsortDeclSet Unit -> Unit .
op addOpDeclSet : EOpDeclSet Unit -> Unit .
op addVarDeclSet : EVarDeclSet Unit -> Unit .
op addMembAxSet : EMembAxSet Unit -> Unit .
op addEquationSet : EquationSet Unit -> Unit .
op addRuleSet : RuleSet Unit -> Unit .
op addClassDeclSet : ClassDeclSet Unit -> Unit .
op addSubclassDeclSet : SubclassDeclSet Unit -> Unit .
op addMsgDeclSet : MsgDeclSet Unit -> Unit .
```

- There are functions and constants to create empty units of the different types. For example, the function `emptyStrFTheory` returns an empty structured functional theory. There is also a function `empty` which takes a unit as argument and returns an empty unit of the same type.

```
op emptyFModule : ModName -> FModule .
op emptyStrFModule : -> StrFModule .
op emptyStrSModule : -> StrSModule .
op emptyStrOModule : -> StrOModule .
op emptyStrFTheory : -> StrFTheory .
op emptyStrSTheory : -> StrSTheory .
op emptyStrOTheory : -> StrOTheory .
op empty : Unit -> Unit .
```

- A function addDecls which returns the unit resulting from adding all the declarations in the unit passed as second argument to the unit passed as first argument.

```
op addDecls : Unit Unit -> Unit .
```

Note that some of the 'set' and 'add' functions are partial functions.

```
...
endfm
```

5 The Abstract Data Type View

In this section we present the data type View for views. Basically, the data elements of sort View are composed by the name of the view, the names of the source and target units, and a set of maps representing the maps asserting how the given target unit is claimed to satisfy the source theory [14, Section 3.5.3].

Internally, renaming maps are considered to be a particular case of view maps. The sort ViewMap is declared as a supersort of Map. The only kind of maps in sort ViewMap not in sort Map are maps of operators going to derived operators. We start introducing the declarations for renaming maps and sets of renaming maps in Section 5.1, we then introduce view maps and sets of view maps in Section 5.2, and finally we introduce the sort View, its constructor, and some operations on it in Section 5.3.

5.1 Renaming Maps

We introduce the different types of renaming maps in the following module MAP. A sort is introduced for each of these types of maps, with the appropriate constructors for each sort (see [14, Section 3.5.4] or [26, Section 3.6.4]). All these sorts are declared to be subsorts of the sort Map. A sort for sets of maps (MapSet) is then declared as supersort of Map with constructors none and _,_.

```
fmod MAP is
  pr EXT-SORT .

  sorts OpMap SortMap LabelMap ClassMap MsgMap AttrMap Map .
  subsorts OpMap SortMap LabelMap ClassMap MsgMap AttrMap < Map .
  op op_to_[_] : Qid Qid AttrSet -> OpMap .
  op op_:_->_to_[_] : Qid ESortList ESort Qid AttrSet -> OpMap .
  op sort_to_ : ESort ESort -> SortMap .
  op label_to_ : Qid Qid -> LabelMap .
  op class_to_ : ESort ESort -> ClassMap .
  op attr_._to_ : Qid ESort Qid -> AttrMap .
  op msg_to_ : Qid Qid -> MsgMap .
  op msg_:_->_to_ : Qid ESortList ESort Qid -> MsgMap .

  sort MapSet .
  subsort Map < MapSet .
  op none : -> MapSet .
  op _,_ : MapSet MapSet -> MapSet [assoc comm id: none] .

endfm
```

5.2 View Maps

In addition to the maps of sort `Map` declared in the `MAP` module, in views there can also be maps from operators to derived operators, that is, terms with variables (Cf. [14, Section 3.5.3]). Maps of this kind are given with the constructor `termMap`, which, in addition to the source and target terms, takes the set of variable declarations for the variables used in the map. The source term must be of the form $\mathtt{F}(\mathtt{X}_1, \ldots, \mathtt{X}_n)$, where `F` is an operator name declared with n arguments of sorts in the connected components of the variables $\mathtt{X}_1 \ldots \mathtt{X}_n$, respectively. We will see in Section 9.3.2 how in the initial processing of a view the variables declared in it are associated to each of the maps in which they are used.

```
fmod VIEW-MAP is
  pr MAP .
  pr EXT-DECL .

  sort TermMap .
  op termMap : EVarDeclSet Term Term -> TermMap .

  sorts ViewMap ViewMapSet .
  subsorts Map TermMap < ViewMap .
  subsorts ViewMap MapSet < ViewMapSet .
  op _,_ : ViewMapSet ViewMapSet -> ViewMapSet [assoc comm id: none] .

endfm
```

5.3 Views

The `View` sort is introduced in the following module `VIEW`. In addition to the constructor for views (`view`), selector functions are added for each of the components of a view (`name`, `source`, `target`, and `mapSet`), and a constant `emptyView`, which is identified in an equation with the empty view, is defined.

Although the declaration of the constructor for views includes an argument for the list of parameters, parameterized views are not handled yet, so at present this argument must be set to the `nilParameterList`.

```
fmod VIEW is
  pr UNIT .
  pr VIEW-MAP .

  sort View .
  op view : ViewExp ParameterList ModExp ModExp ViewMapSet -> View .

  op name : View -> ViewExp .
  op source : View -> ModExp .
  op target : View -> ModExp .
  op mapSet : View -> ViewMapSet .

  op emptyView : Qid ModExp ModExp -> View .

  eq emptyView(QI, ME, ME')
    = view(QI, nilParameterList, ME, ME', none) .

   ...

endfm
```

6 The Abstract Data Type Database

In this section we present the data type Database, which will be used to store information about the units and views in the system. Before discussing this data type in Section 6.2, we present the predefined units added in Full Maude to those already available in Core Maude.

6.1 Non-Built-In Predefined Modules

As we shall see in the following section, except for the LOOP-MODE module, all the predefined modules that are available in Core Maude are also available in Full Maude. In addition to these Core Maude predefined modules, in Full Maude we have some additional predefined units. In the present system, the only units with which the database is initialized (see Section 10.2) are the functional theory TRIV, the module CONFIGURATION (see [14, Section 3.2] and [26, Section 3.3]), and the module UP, which will be used to evaluate the up functions (see [14, Section 3.3], [26, Section 3.4], and 7.4). We shall see in Section 10.2 how new predefined modules can be added to the initial database.

```
fmod PREDEF-UNITS is
  pr UNIT .

  op TRIV : -> StrFTheory .
  eq TRIV
    = (fth 'TRIV is
          nilParameterList
          nil
          sorts 'Elt .
          none
          none
          none
          none
          none
       endfth) .

  op CONFIGURATION : -> StrFModule .
  eq CONFIGURATION
    = (fmod 'CONFIGURATION is
          nilParameterList
          nil
          sorts 'Oid ; 'Cid ; 'Attribute ; 'AttributeSet ;
                'Configuration ; 'Object ; 'Msg .
          subsort 'Attribute < 'AttributeSet .
          subsort 'Object < 'Configuration .
          subsort 'Msg < 'Configuration .
          op 'none : nil -> 'AttributeSet [none] .
          op '_`,_ : 'AttributeSet 'AttributeSet -> 'AttributeSet
                [comm assoc id({'none}'AttributeSet)] .
          op 'none : nil -> 'Configuration [none] .
          op '__ : 'Configuration 'Configuration -> 'Configuration
                [comm assoc id({'none}'Configuration)] .
          op '<_:_|_> : 'Oid 'Cid 'AttributeSet -> 'Object [none] .
          op '<_:_|`> : 'Oid 'Cid -> 'Object [none] .
          var 'O : 'Oid .
          var 'C : 'Cid .
          none
```

```
    eq '<_:_|`>['O, 'C] = '<_:_|_>['O, 'C, {'none}'AttributeSet] .
  endfm) .
```

The following module UP contains the necessary declarations to be able to parse the up functions presented in [14, Section 3.3] and [26, Section 3.4]. We shall see in Section 7.7 how a declaration importing the following module UP is added to all the modules importing the predefined module META-LEVEL. With this declaration, it is possible to parse the up commands in the bubbles of such modules or in commands being evaluated in such modules. We shall see in Section 7.4 how these commands are then evaluated.

```
op UP : -> StrFModule .
eq UP
  = (fmod 'UP is
        nilParameterList
        including 'QID-LIST .
        including 'META-LEVEL .
        sorts 'Token ; 'Bubble .
        none
        op 'token : 'Qid -> 'Token
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
        op 'bubble : 'QidList -> 'Bubble
             [special(
               (id-hook('Bubble, '1 '-1 '`( '`))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
        op 'up : 'Token 'Bubble -> 'Term [none] .
        op 'up : 'Token -> 'Module [none] .
        none
        none
        none
     endfm) .

endfm
```

6.2 The Database

In order to be able to refer to modules by name, which is extremely useful for module definition purposes at the user level, the evaluation of module expressions takes place in the context of a database, in which we keep information about the modules already introduced in the system, and also about those modules generated internally. This information is stored as a set of elements of sort UnitInfo and ViewInfo, in which we hold, respectively, the information concerning units and views. For each unit we save:

- Its original form, as introduced by the user, or, in case of an internally generated unit, as generated from the original form of some other unit.
- Its internal representation, in which variables have been renamed to avoid collisions with the names of variables in other units in the same hierarchy. In the case of object-oriented units, we store its equivalent system module, that is, the result of transforming it into a system module.
- Its signature, which is given as a functional module of sort FModule with no axioms, ready to be used in calls to meta-parse. There can only be

importation declarations including built-in modules in this module. These are the only inclusions handled by the Core Maude engine.

- Its flattened version, for which, as for signatures, only the importation of built-in modules is left unevaluated.

For each view we keep its name and the view itself.

As a simple mechanism to keep the database consistent, for each unit we maintain the list of names of all the units and views 'depending' on it. Similarly, for each view we maintain the list of names of all the units 'depending' on it. The idea is that if a unit or view is redefined or removed, all those units and/or views depending on it will also be removed. This dependency does not only mean direct importation. For example, the module resulting from the renaming of some module also depends on the module being renamed; the instantiation of a parameterized module also depends on the parameterized module and on all the views used in its instantiation; a view depends on its source and target units, etc. This dependency is transitive: if a module, theory, or view has to be removed, all the units and/or views depending on them will be removed as well. The dependencies derived from the module expressions themselves are established by the function `setUpModExpDependencies`. The function `setUpUnitDependencies` calls `setUpModExpDependencies`, and then `setUpImportSetDependencies` to add the *back references* in the modules being imported. The function `setUpViewDependencies` sets up the back references for the views being introduced.

In addition to this set of information cells for units and views, we also keep lists with the names of all the units and views in the database, and a list of quoted identifiers in which we store the messages generated during the process of treatment of the inputs in order to simplify the communication with the read-eval-print loop process.

```
fmod DATABASE is
  pr VIEW .

  sort UnitInfo .
  op <_;_;_;_;_;_;_> :
         ModName Unit Unit Unit Module ModNameSet ViewExpSet -> UnitInfo .

  sort ViewInfo .
  op <_;_;_> : ViewExp View ModNameSet -> ViewInfo .

  sort InfoSet .
  subsort UnitInfo ViewInfo < InfoSet .
  op emptyInfoSet : -> InfoSet .
  op __ : InfoSet InfoSet -> InfoSet [assoc comm id: emptyInfoSet] .

  sort Database .
  op database : InfoSet ModNameSet ViewExpSet QidList -> Database .

  vars QI X : Qid .
  vars QIL QIL' : QidList .
  var VE : ViewExp .
  vars VES VES' : ViewExpSet .
  var IS : InfoSet .
  vars MNS MNS' MNS'' : ModNameSet .
  var PL : ParameterList .
  vars ME ME' : ModExp .
  var VI : View .
```

```
var VMAPS : ViewMapSet .
vars U U' U'' : Unit .
var DB : Database .
vars M M' M'' : Module .
var EIL : EImportList .
vars MN MN' : ModName .
```

The constant `emptyDatabase` denotes the empty database, and there are predicates `viewInDatabase` and `unitInDatabase` to check, respectively, whether a view and a unit are in a database or not.

```
op emptyDatabase : -> Database .
eq emptyDatabase
  = database(emptyInfoSet, noneModNameSet, noneViewExpSet, nil) .

op unitInDatabase : ModName Database -> Bool .
op viewInDatabase : ViewExp Database -> Bool .
```

If a module, theory, or view is being redefined, that is, if there was already in the database a module, theory, or view with the same name, then all the units and/or views depending on it are removed using the functions `deleteUnits` and `deleteViews`. Removing a view or a unit from the database means removing its info cell from the set of cells in the database.

```
op deleteUnits : ModNameSet Database -> Database .
op deleteViews : ViewExpSet Database -> Database .
```

The `warning` function allows us to place messages (warning, error, or any other kind of messages) in the last argument of the database constructor. These messages are given in the form of quoted identifier lists, and will be passed to the third argument of the read-eval-print loop, to be printed in the terminal.

```
op warning : Database QidList -> Database .
eq warning(database(IS, MNS, VES, QIL), QIL')
  = database(IS, MNS, VES, (QIL QIL')) .
```

The constant `builtIns` denotes the set of identifiers of the predefined modules of Core Maude. It will be used to check whether a particular module is a built-in Core Maude module or not.

```
op builtIns : -> ModNameSet .
eq builtIns
  = 'TRUTH-VALUE & 'TRUTH & 'BOOL & 'IDENTICAL & 'MACHINE-INT &
    'QID & 'QID-LIST & 'META-LEVEL & 'LOOP-MODE .
```

Core Maude built-in modules are handled in a special way in the current version of the system. They are not explicitly defined in the Full Maude database; their importation is directly handled by Core Maude. This has some drawbacks: Core Maude built-in modules cannot be renamed; they cannot be directly used with built-in functions, such as `meta-reduce` or `sameComponent`, although they can be imported in modules being used in the calls to these functions; and, in general, any function taking as argument or returning as result the metarepresentation of a module cannot take one of these built-in modules as argument. This is the case, for example, for the `up` function presented in Section 7.2, or for functions or commands in which the name of a module has to be specified, as the `select` or `down` commands, or the `up` function presented in [14, Section 3.3] and in [26, Section 3.4].

Nevertheless, there are also some advantages: The flattening of the built-in part of the structure is accomplished more efficiently, and, since these modules do not have to be stored in the database of Full Maude, the size of the database is reduced.

Our plan is to have in the future a hybrid solution. Once we have some way of storing the modules entered to Full Maude in Core Maude's database, it will be enough to keep in the Full Maude database just the original form of the top of all the modules, including built-ins, leaving all the importation declarations to be resolved by the engine. The structures will be normalized as they are now, so that the engine will have to deal just with inclusions, but it will be possible to use the predefined modules as any other module. Moreover, the Full Maude database will be relatively smaller and the flattening will be computed more efficiently.

When a new module or theory is entered, the names of all the modules, theories, and views depending on it are included in its lists of dependencies with functions `setUpUnitDependencies` and `setUpViewDependencies`. Notice that if new module expressions are defined, the function `setUpModExpDependencies` will have to be extended accordingly.

```
op setUpUnitDependencies : Unit Database -> Database .
op setUpModExpDependencies : ModName Database -> Database .
op setUpImportSetDependencies : ModName EImportList Database -> Database .

eq setUpUnitDependencies(U, DB)
  = setUpImportSetDependencies(name(U), importList(U),
      setUpModExpDependencies(name(U), DB)) .

eq setUpModExpDependencies((par X :: ME),
     database((< ME ; U ; U' ; M ; M' ; MNS ; VES > IS), MNS', VES', QIL))
  = database((< ME ; U ; U' ; M ; M' ; MNS & (par X :: ME) ; VES > IS),
         MNS', VES', QIL) .
ceq setUpModExpDependencies((par X :: ME), DB)
  = warning(DB,
      ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
    if not unitInDatabase(ME, DB) .
eq setUpModExpDependencies(QI, DB) = DB .

eq setUpImportSetDependencies(MN, ((including MN' .) EIL),
     database((< MN' ; U ; U' ; M ; M' ; MNS' ; VES > IS), MNS'', VES', QIL))
  = setUpImportSetDependencies(MN, EIL,
       database((< MN' ; U ; U' ; M ; M' ; MNS' & MN ; VES > IS),
         MNS'', VES', QIL)) .
ceq setUpImportSetDependencies(MN, ((including MN' .) EIL), DB)
  = if MN' inModNameSet builtIns
    then DB
    else warning(DB,
           ('\n 'ERROR: 'module modNameToQidList(MN') 'not 'in 'database '.))
    fi
    if not unitInDatabase(MN', DB) .
eq setUpImportSetDependencies(MN, nil, DB) = DB .

op setUpViewDependencies : ModExp ViewExp Database -> Database .

eq setUpViewDependencies(ME, VE,
      database((< ME ; U ; U' ; M ; M' ; MNS' ; VES > IS), MNS'', VES', QIL))
  = database((< ME ; U ; U' ; M ; M' ; MNS' ; VES # VE > IS),
       MNS'', VES', QIL) .
ceq setUpViewDependencies(ME, VE, DB)
```

```
  = if ME inModNameSet builtIns
    then DB
    else warning(DB,
           ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
    fi
    if not unitInDatabase(ME, DB) .
```

There are functions to insert and extract a view in the database.

```
op insertView : View Database -> Database .
op getView : ViewExp Database -> View .

eq insertView(view(VE, PL, ME, ME', VMAPS),
     database((< VE ; VI ; MNS > IS), MNS', VES, QIL))
  = setUpViewDependencies(ME, VE,
      setUpViewDependencies(ME', VE,
        deleteUnits(MNS,
          database(
            (< VE ; view(VE, PL, ME, ME', VMAPS) ; noneModNameSet > IS),
            MNS', VES, QIL)))) .
ceq insertView(view(VE, PL, ME, ME', VMAPS), database(IS, MNS, VES, QIL))
  = setUpViewDependencies(ME, VE,
      setUpViewDependencies(ME', VE,
        database((< VE ; view(VE, PL, ME, ME', VMAPS) ; noneModNameSet > IS),
          MNS, (VE # VES), QIL)))
    if not (VE inViewExpSet VES) .

eq getView(VE, database((< VE ; VI ; MNS > IS), MNS', VES, QIL)) = VI .
```

There are functions to insert the different versions of a unit, and to extract them. We only give here the equations for the insertion of top units to illustrate the way in which the consistency of the database is maintained. We assume that when the internal version, the signature, or the flat version of a module is entered in the database, its corresponding top module is already present in it.

```
op insertTopUnit : Unit Database -> Database .
op insertInternalTopUnit : Unit Database -> Database .
op insertSignature : ModName Module Database -> Database .
op insertFlatUnit : ModName Module Database -> Database .
op getTopUnit : ModName Database -> Unit .
op getInternalTopUnit : ModName Database -> Unit .
op getSignature : ModName Database -> Unit .
op getFlatUnit : ModName Database -> Unit .

ceq insertTopUnit(U, database(IS, MNS, VES, QIL))
  = setUpUnitDependencies(U,
      database(
        (< name(U) ; U ; noModule ; noModule ;
           noModule ; noneModNameSet ; noneViewExpSet > IS),
        (name(U) & MNS), VES, QIL))
    if not (name(U) inModNameSet MNS) .
ceq insertTopUnit(U,
      database((< MN ; U' ; U'' ; M ; M' ; MNS ; VES > IS), MNS', VES', QIL))
  = setUpUnitDependencies(U,
      deleteUnits(MNS,
        deleteViews(VES,
           database(
```

```
          (< MN ; U ; noModule ; noModule ;
             noModule ; noneModNameSet ; noneViewExpSet > IS),
          MNS', VES',
          (QIL ('\n 'ADVISORY:
                 'Module modNameToQidList(MN) 'redefined)))))
    if name(U) == MN .

  ...

endfm
```

7 The Evaluation of Units

The general principle for the evaluation of units in our design consists in first evaluating any module expression, reducing it to a canonical form in which only unit inclusions appear, that is, to a unit hierarchy, which can be seen as a partial order of unit inclusions. The design of the Full Maude system has been based upon the principle of evaluating all module expressions to irreducible structured units, and on using the flat version of the units only for execution purposes. We have then two different processes clearly distinguished: a first step in which the structured unit is evaluated and reduced to its normal form, and a second step in which this normal form is flattened. We illustrate both stages of this process in Section 7.1 with a very simple example, namely the importation of a renamed module.

As explained in Section 1.1, the process of evaluation to normal form is also responsible for the parsing of the bubbles in the premodules, which is accomplished once the signature has been built. The parsing of bubbles is discussed in Section 7.4. To be able to handle the `up` function and the `down` command presented in [14, Section 3.3] and in [26, Section 3.4], it is necessary to be able to move terms and modules from one level of reflection to another. The functionality to move between levels is presented in Section 7.2, where functions `up` and `down` on sorts `Module` and `Term` are defined. The transformation of object-oriented modules into system modules in discussed in Section 7.6. The evaluation of module expressions is discussed in Sections 7.5, 7.8, 7.9, and 7.10.

7.1 Overview of the Unit Evaluation Process

Let us consider the following specification `NAT` of natural numbers, as a functional module with a sort `Nat` and operations `zero` and `suc` with the usual meaning, and a specification `NAT3` of the natural numbers modulo 3, given by importing a renamed copy of the module `NAT` and adding the corresponding congruence equation. Note that this module is imported in *including* mode, that is, nothing is guaranteed about the relationship between the initial models of `NAT` and `NAT3`.

```
(fmod NAT is
   sort Nat .
   op zero : -> Nat .
   op suc : Nat -> Nat .
 endfm)

(fmod NAT3 is
   including NAT *(sort Nat to Nat3) .
   eq suc(suc(suc(zero))) = zero .
 endfm)
```

The renaming process is summarized in the following picture. Basically, we create a copy of the module NAT, with name NAT *(sort Nat to Nat3), in which the sort Nat has been renamed to Nat3. This amounts to a module transformation denoted by ϕ in the picture below. Then, the renamed module is included as a submodule of NAT3, in which the congruence equation is added. It is important to highlight the fact that there is no inclusion relationship between the modules NAT and NAT3. The "confusion" associated to the congruence modulo 3 equation has only been introduced in the sort Nat3, which has nothing to do with the sort Nat.

$$\begin{tikzcd} & \texttt{NAT3} \\ \texttt{NAT} \arrow[r, dashed, "\phi"'] & \texttt{NAT *(sort Nat to Nat3)} \arrow[u, hook] \end{tikzcd}$$

The metarepresentations of the modules NAT and NAT3 before being evaluated are given by the following terms of sort StrFModule:

```
fmod 'NAT is
  nilParameterList
  nil
  sorts 'Nat .
  none
  op 'zero : nil -> 'Nat [none] .
  op 'suc : 'Nat -> 'Nat [none] .
  none
  none
  none
endfm

fmod 'NAT3 is
  nilParameterList
  including 'NAT *(sort 'Nat to 'Nat3) .
  sorts none .
  none
  none
  none
  none
  eq 'suc['suc['suc[{'zero}'Nat3]]] = {'zero}'Nat3 .
endfm
```

7.1.1 Normalization of Extended Modules

For NAT3, the first stage of the module evaluation process consists in the reduction of the module to its structured normal form. This normalization step can be understood as the evaluation of all module expressions in it. Since the normal forms of all previously defined modules have been computed and have been stored in the database, the module expressions have to be evaluated only in the module at the top of the hierarchy. However, some of the operations may affect the whole structure, or part of it. This happens, for example, with the renaming operation when some of the renamings affect sorts or operators in a submodule. We will discuss the details about the evaluation of the renaming module expression in Section 7.10. Note that in this case, as in general for all the module operations we deal with, given a structured module as input, we evaluate it without flattening, and return a structured module as result.

The evaluation of the only module expression appearing in the module `NAT3`, namely `NAT *(sort Nat to Nat3)`, results in a new module, whose *name* is given by the module expression itself. This module is then introduced in the database, in such a way that if the module expression appears again, it will not be recomputed. The representation of this module at the metalevel is

```
fmod 'NAT *(sort Nat to Nat3) is
  nilParameterList
  nil
  sorts 'Nat3 .
  none
  op 'zero : nil -> 'Nat3 [none] .
  op 'suc : 'Nat3 -> 'Nat3 [none] .
  none
  none
  none
endfm
```

Note that the name of the module is the module expression itself. Thus, the normalization process does not change the text of the module being normalized, such as `NAT3` in our example. Instead, each imported module expression is evaluated and a new module with that expression as name is entered into the database. In this way, the normalization yields a structured module in which only inclusions appear. The following picture shows the form of this structure for the example.

$$\begin{tikzcd} \texttt{NAT3} \\ \texttt{NAT *(sort Nat to Nat3)} \arrow[u, hook] \end{tikzcd}$$

7.1.2 Flattening

The flattening of the normalized structure is accomplished following the tradition of the Clear/OBJ family of languages, in which specification structuring is based on the categorical concept of *colimit* [9, 25, 30]. However, instead of considering the category of specifications and specification morphisms [9], flattening is understood as a colimit in the category of specifications and inclusions of specifications. The colimit of a diagram in this category coincides with the set-theoretic union of the theories in the diagram. This is accomplished by the function

$$\texttt{evalUnit} : \texttt{Unit Database} \to \texttt{Database}$$

that takes a unit U and a database and returns the database after having evaluated all the module expressions appearing in U. The corresponding flat form of the module is also included in the database.

The resulting flat specification for `NAT3` is:

```
fmod 'NAT3 is
  nilParameterList
  nil
  sorts 'Nat3 .
  none
  op 'zero : nil -> 'Nat3 [none] .
  op 'suc : 'Nat3 -> 'Nat3 [none] .
  none
```

```
  none
  eq 'suc['suc['suc[{'zero}'Nat3]]] = {'zero}'Nat3 .
endfm
```

Note that in the flattening of a structured module we are assuming that sort names are unique in each structure, although not necessarily in the entire database. This proposal is simpler than the solution taken, for example, in OBJ3 [32], in which sorts and operations are qualified by module names. However, although simpler, this solution would not be fully satisfactory without some mechanism to help the user avoid the systematic renaming of repeated sorts, which—as discussed in [14, Section 3.5.2] and [26, Section 3.6.2]—is a common occurrence in the presence of parameterized modules. This is accomplished by parameterized sorts [14, 26]

7.2 Changing Levels

Moving terms of sorts `Term` and `Module` from one level of reflection to another is possible thanks to the `up` and `down` functions, which are defined, respectively, in the following modules `MOVE-UP` and `MOVE-DOWN`.

7.2.1 The up Function

Given a term of sort `Module` or `Term`, the `up` function, defined in the following module `MOVE-UP`, returns the term metarepresenting it. The function is also defined on the different components of a unit. The `up` function on units is defined recursively by applying it to the different components of a unit.

The semantics of the `up` functions can be inferred from the following example. The result of applying the `up` function to the metarepresentation of the flat form of the module `NAT3` presented in Section 7.1 is the following.

```
red up(fmod 'NAT3 is
          nil
          sorts 'Nat3 .
          none
          op 'zero : nil -> 'Nat3 [none] .
          op 'suc : 'Nat3 -> 'Nat3 [none] .
          none
          none
          eq 'suc['suc['suc[{'zero}'Nat3]]] = {'zero}'Nat3 .
        endfm) .

result Term:
  'fmod_is_______endfm[
     {''NAT3}'Qid,
     {'nil}'ImportList,
     'sorts_.[{''Nat3}'Qid],
     {'none}'SubsortDeclSet,
     '__['op_:_->_`[_`].[{''zero}'Qid, {'nil}'QidList, {''Nat3}'Qid,
           {'none}'AttrSet],
         'op_:_->_`[_`].[{''suc}'Qid, {''Nat3}'Qid, {''Nat3}'Qid,
           {'none}'AttrSet]],
     {'none}'VarDeclSet,
     {'none}'MembAxSet,
     'eq_=_.['_`[_`][{''suc}'Qid,
              '_`[_`][{''suc}'Qid,
                '_`[_`][{''suc}'Qid,
                  '`{_`}_[{''zero}'Qid, {''Nat3}'Qid]]]],
```

```
            ''{_'}_[{''zero}'Qid, {''Nat3}'Qid]]]
```

We shall see in Section 7.4 how the up function is used to evaluate the homonymous function discussed in [14, Section 3.3] and in [26, Section 3.4]. In Section 7.9 we shall discuss how the up function is used to evaluate the META-LEVEL module expression [14, 26].

```
fmod MOVE-UP is
  pr UNIT .

  op up : Module -> Term .
  op up : Term -> Term .

   ...

endfm
```

7.2.2 The down Function

Given a term of sort Term metarepresenting a term of sort Term or Module, the down function can be seen as the inverse of the up function discussed in the previous section, that is, it returns the original term that had been metarepresented. There are also down functions for terms metarepresenting terms in other sorts. We present here only some of them.

We assume that the down functions are called with valid metarepresentations. In fact, these functions should be declared as partial functions going to error sorts when their arguments are invalid.

The main application of the down functions is in the evaluation of the down command (see [14, Section 3.3] or [26, Section 3.4]). However, they are also used in other tasks, as for example in the parsing of some inputs.

```
fmod MOVE-DOWN is
  pr UNIT .

  op downTerm : Term -> Term .
  op downModule : Term -> Module .
  op downQid : Term -> Qid .
  op downQidList : Term -> QidList .
  op downQidSet : Term -> QidSet .
  op downModName : Term -> ModName .
  op downMachineInt : Term -> MachineInt .

   ...

endfm
```

7.3 Renaming of Variables

To avoid the clash of names of variables appearing in different units in the same unit hierarchy, which can appear very easily when such a hierarchy is flattened to a single unit, variables are systematically renamed. The function renameVars takes a unit and returns it with all the variables in it renamed following the convention of adding the character '@' plus the name of its sort after being converted to a Qid. Thus, for example, a variable 'N of sort 'Nat is renamed to 'N@Nat, and a variable 'L of sort 'List['X] is renamed to 'L@List'['X'].

```
fmod VBLE-RENAMING is
  pr UNIT .
  pr EXT-SORT-TO-QID .

  op renameVars : Unit -> Unit .

   ...

endfm
```

7.4 Parsing of Bubbles

As discussed in Section 1, in Full Maude, the parsing process is split into two phases. In a first stage, the input is parsed using the top-level grammar for Full Maude modules, theories, views, and commands. Once this first stage is completed, we get a term with bubbles in it, which is converted into a module, theory, or view. This unit or view may still have the bubbles in it. We say that a module with bubbles is a premodule, a view with bubbles a preview, and so on. The second stage of the process consists in taking this preunit or preview and converting the bubbles in it into terms by parsing them in the appropriate signatures, obtaining a 'valid' unit or view out of it, or otherwise a parsing error. In the case of commands, if they contain any bubble, the same will have to be done. All bubbles have to be parsed in the appropriate signature before any further processing can be done with the module, view, or command in which they appear.

7.4.1 Parsing of Bubbles

In the following module `BUBBLE-PARSING`, the definitions for the basic processing of bubbles are introduced. In it we declare a function `solveBubbles` which takes a bubble and some other arguments and returns the term resulting from parsing it.

```
fmod BUBBLE-PARSING is
  pr DATABASE .
  pr MOVE-UP .
  pr MOVE-DOWN .
  pr EXT-SORT-TO-QID .

  var T : Term .
  var M : Module .
  var B : Bool .
  var QIL : QidList .
  var DB : Database .
```

As we shall see in Section 7.7, a declaration importing the predefined module `UP` (see Section 6.1) is added to all modules importing the `META-LEVEL` module. The `solveBubbles` function is called with a 'flag' indicating whether the module can contain calls to the `up` function or not. Thus, when we call `meta-parse` with some bubble and the module in which such bubble has to be parsed, if there are occurrences of the function `up` in it, they will be of the form `'up['token[T]]` or `'up['token[T], 'bubble[T']]` for terms `T` and `T'`. The function `solveUps` will evaluate them.

```
  op solveBubbles : Term Module Bool Database -> Term .
  op solveBubbles1 : QidList Module Bool Database -> Term .
  op solveUps : TermList Database -> TermList .
```

The function solveBubbles calls the function solveBubbles1 after lowering the list of quoted identifiers of the bubble.

```
  eq solveBubbles('bubble[T], M, B, DB)
    = solveBubbles1(downQidList(T), M, B, DB) .
```

The solveBubbles1 function is in charge of calling the function meta-parse. The flag indicating the inclusion of the module META-LEVEL in the module in which the term appears decides whether the function solveUps is called or not, so the extra price of searching for calls to the up function is paid only when an occurrence of the function is possible. This function takes care of the occurrences of the up function that may exist in such bubbles.

The function solveUps goes through the term looking for a term with 'up as top operator and 'token as top operator of its unique argument if there is only one argument, or with 'token and 'bubble as top operators of its first and second arguments, respectively, if there are two. If a term of the form 'up['token[T]] is reached, it is replaced by the metarepresentation of the flat version of the module in the database with the name given by the token. If a term of form 'up['token[T], 'bubble[T']] is reached, the metarepresentation of the result of parsing the bubble in the signature of the module with the name given by the token, after solving possible nested calls to the up function, is returned.

```
  eq solveBubbles1(QIL, M, true, DB) = solveUps(meta-parse(M, QIL), DB) .
  eq solveBubbles1(QIL, M, false, DB) = meta-parse(M, QIL) .

   ...

endfm
```

7.4.2 Parsing the Bubbles in a Unit

The solveBubbles function defined in the UNIT-BUBBLE-PARSING module takes a term of sort Unit (a preunit in fact) and a signature, and returns the unit resulting from the evaluation (parsing) of all the bubbles in it.

```
fmod UNIT-BUBBLE-PARSING is
  pr BUBBLE-PARSING .
  pr DATABASE .
  pr MOVE-UP .
  pr MOVE-DOWN .
  pr EXT-SORT-TO-QID .

  op solveBubbles : Unit Module Bool Database -> Unit .

   ...

endfm
```

7.5 Module Expression Evaluation

So far we have not introduced more module expressions than those given by simple quoted identifiers. We will introduce some later, but the scheme followed for evaluating them is very simple and can be presented in a generic way. Given a module expression and a database state, the evaluation of a module expression results in the generation of a new module, which is introduced in the database, with the module expression as its name. The resulting database is then returned. If there is already a module in the database with that name, the function returns the original database without any change. The evaluation of a module expression may produce the evaluation of other module expressions contained in the modules involved in the process. This is the case, for example, for the renaming of modules, in which not only the top module is renamed but, perhaps, some of its submodules as well; it is also the case for the instantiation of parameterized modules, where the module being instantiated may contain submodules which are parameterized by some of the parameter theories of the parameterized module in which are imported. We shall discuss in more detail the renaming and instantiation of module expressions in Sections 7.10 and 7.9, respectively.

It is possible to import a module expression in which a parameterized module is instantiated by some of the formal parameters of the parameterized module into which it is imported (Cf. [14, Section 3.5.4] or [26, Section 3.6.4]). To be able to evaluate this kind of module expression, the list of parameters of the module in which the module expression appears has to be given.

```
fmod MOD-EXPR-EVAL is
  pr DATABASE .

  op evalModExp : ModExp Database -> Database .
  op evalModExp : ModExp ParameterList Database -> Database .

  var QI : Qid .
  var ME : ModExp .
  var PL : ParameterList .
  var DB : Database .

  eq evalModExp(ME, DB)
    = evalModExp(ME, nilParameterList, DB) .

  ceq evalModExp(QI, PL, DB)
    = if QI inModNameSet builtIns
      then DB
      else warning(DB, ('ERROR: 'module QI 'not 'in 'database))
      fi
      if not unitInDatabase(QI, DB) .
  ceq evalModExp(ME, PL, DB)
    = DB
      if unitInDatabase(ME, DB) .

endfm
```

7.6 The Transformation of Object-Oriented Modules to System Modules

A discussion on the transformation of object-oriented modules into system modules can be found in [44, 27, 14, 26]. We focus here on the part of the process accomplished by each of the main functions involved in the transformation.

The transformation discussed in [27] assumed that object-oriented modules were flattened before being transformed into system modules. However, doing it in this way, the transformation already made for the modules in the structure were not reused. In the current system, the transformation is done only for the module being introduced, the top of the structure, and using the 'internal' representations of the submodules stored in the database for the rest of the structure.

This approach requires gathering all class and subclass relation declarations in the structure before starting with the transformation process itself. The function `prepareClasses` collects all these declarations in the structure, and completes all the declarations of classes with the attributes inherited from their superclasses.

Once all the class declarations in the structure have been collected and completed, the transformation is accomplished in two stages. First, the function `objectOrientedToSystemModuleA` carries out the following tasks:

- For each class declaration of the form `class` C `|` $a_1 : S_1$ `,` ... `,` $a_n : S_n$, the following items are 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 (the function `attributeOpDeclSet` creates these declarations).

- For each subclass relation of the form `subclass` C `<` C', a subsort declaration `subsort` C `<` C' is introduced.

- For each message declaration of the form `msg F : ESL -> ES`, an operator declaration `op F : ESL -> ES` is added.

When this process has been completed, the function `prepareAxioms` is called. This function applies to the membership axioms, equations, and rewriting rules in the module the transformations indicated in [14, Section 3.2.2] and in [26, Section 3.3.2], so that they become applicable to all the objects of the given class and of their subclasses. The set of attributes of the objects appearing in the membership axioms, equations, and rewriting rules are completed, so that the default convention of not having to exhaustively mention the set of attributes of a class is supported.

Note that in Meseguer's paper [44] a parallel hierarchy of sorts was defined to deal with objects in different classes, and membership axioms constraining the objects to their corresponding sorts were added. The transformation could be easily completed with sorts, subsort relations, and membership constraints as indicated there. In fact, these declarations were added in an initial version and were then removed because they were computationally expensive. However, there are examples in which it would be interesting to have them; when needed, these declarations can be explicitly added by the user in the current version.

```
fmod O-O-TO-SYSTEM-MOD-TRANSF is
  pr DATABASE .
  pr VBLE-RENAMING .

  var DB : Database .
  var I : MachineInt .
  var MN : ModName .
  var F : Qid .
  vars ES C C' : ESort .
  var ESS : ESortSet .
  var ESL : ESortList .
  vars T T' T'' T''' : Term .
  var PL : ParameterList .
  vars EIL EIL' : EImportList .
  var ESD : ESortDecl .
  vars CDS CDS' : ClassDeclSet .
  var ADS : AttrDeclSet .
  var ESSDS : ESubsortDeclSet .
  var SCDS : SubclassDeclSet .
  var EODS : EOpDeclSet .
  var MDS : MsgDeclSet .
```

```
var EVDS : EVarDeclSet .
vars EMAS EMAS' : EMembAxSet .
vars EqS EqS' : EquationSet .
vars RlS RlS' : RuleSet .
```

The function prepareClasses completes all classes in the module with all the attributes they inherit from their superclasses.

```
op prepareClasses :
     ClassDeclSet SubclassDeclSet EImportList Database -> ClassDeclSet .
```

Given a set of attribute declarations, the attributeOpDeclSet function returns a set of operator declarations as indicated above. That is, for each attribute a:S, an operator of the form a `:_ :` S `-> Attribute` is declared.

```
op attributeOpDeclSet : AttrDeclSet -> OpDeclSet .
```

The function prepareLHS takes the term in the lefthand side of a rule, equation, or membership axiom, and replaces each object `<` O `:` C `|` ADS `>` in it—with O of sort Oid, C the name of a class, and ADS a set of attributes with their corresponding values—by an object `<` O `:` V `|` $ADS\ ADS'\ Atts$ `>` where the identifier of the class is replaced by a variable V of sort C, which is not used in the axiom, and where the set of attributes is completed with attributes ADS' as indicated in [14, Section 3.2.2] and in [26, Section 3.3.2], so that all attributes declared in class C or in any of its superclasses is added with a new variable as value. *Atts* is a new variable of sort AttributeSet, which is used to range over the additional attributes that may appear in objects of a subclass.

The function prepareLHS takes as arguments a term (in the initial call, the term in the lefthand side of a rule, equation, or membership axiom), the set of variable declarations of those variables declared in the module that are not used in the axiom—new variables are created only if there are no variables in the module with the appropriate sort—, the set of class declarations in the module—to be able to complete the set of attributes in the occurrences of the objects—, and an index—to make sure that the variables being added have not been added previously. In the initial call this index is set to zero. prepareLHS gives as result a tuple composed by the resulting term, the set of objects in the term (so that the modification of the objects in the righthand side of the rule is simplified), the set of variable declarations corresponding to the new added variables, the set of variable declarations of the variables in the module that have not been used, and the index for the creation of new variables.

The set of objects in the lefthand side will be given as a set of terms. The sort TermSet is defined as a supersort of the sort Term, and with constructors emptyTermSet and termSet as follows.

```
sort TermSet .
subsort Term < TermSet .
op emptyTermSet : -> TermSet .
op termSet : TermSet TermSet -> TermSet [comm assoc id: emptyTermSet] .
```

Terms of sort PrepareLHSResult are built with constructor `<_;_;_;_;_>`, and have selectors for their different components.

```
sort PrepareLHSResult .
op <_;_;_;_;_> : TermList TermSet EVarDeclSet EVarDeclSet MachineInt
     -> PrepareLHSResult .
```

```
op term : PrepareLHSResult -> TermList .
op objects : PrepareLHSResult -> TermSet .
op oldVbles : PrepareLHSResult -> EVarDeclSet .
op newVbles : PrepareLHSResult -> EVarDeclSet .
op index : PrepareLHSResult -> MachineInt .

op prepareLHS :
     TermList EVarDeclSet ClassDeclSet MachineInt -> PrepareLHSResult .
```

In the case of equations and rules, the function `prepareAxioms` calls the function `prepareLHS` with the term in the lefthand side of the axiom, and then use the generated set of objects to call the `prepareRHS` function. For conditional equations, rules, and membership axioms, this set of terms representing the objects in the lefthand side is also used in the calls to `prepareRHS` with each of the terms in the conditions. The term in the lefthand side of the equation, rule, or membership axiom is replaced by the term returned by `prepareLHS`. The variables generated are added to the variables in the module, and the index is used in the recursive calls to `prepareAxioms`.

`prepareLHS` returns as second argument the set of objects (as a set of terms) appearing in it. These objects are returned after extending their set of attributes by those of the class to which they belong not already specified.

```
op prepareAxioms :
     Unit MembAxSet EquationSet RuleSet ClassDeclSet MachineInt -> Unit .
```

We illustrate the process presenting the equation corresponding to the case of conditional equations.

```
eq prepareAxioms(
     mod MN is PL EIL ESD ESSDS EODS EVDS EMAS EqS RlS endm,
     EMAS', ((ceq T = T' if T'' = T''' .) EqS'), RlS', CDS, I)
  = prepareAxioms(
      mod MN is
         PL EIL ESD ESSDS EODS
         (EVDS newVbles(
                 prepareLHS(T,
                   varDeclSetDiff(EVDS,
                     (varDeclSetInTerm(EVDS, T)
                      varDeclSetInTerm(EVDS, T'))),
                   CDS, I)))
         EMAS
         ((ceq term(
                 prepareLHS(T,
                   varDeclSetDiff(EVDS,
                     (varDeclSetInTerm(EVDS, T)
                      varDeclSetInTerm(EVDS, T'))),
                   CDS, I))
             = prepareRHS(
                 objects(
                   prepareLHS(T,
                     varDeclSetDiff(EVDS,
                       (varDeclSetInTerm(EVDS, T)
                        varDeclSetInTerm(EVDS, T'))),
                     CDS, I)),
                 T')
             if prepareRHS(
                  objects(
```

```
                 prepareLHS(T,
                   varDeclSetDiff(EVDS,
                     (varDeclSetInTerm(EVDS, T)
                      varDeclSetInTerm(EVDS, T'))),
                   CDS, I)), T'')
             = prepareRHS(
                 objects(
                   prepareLHS(T,
                     varDeclSetDiff(EVDS,
                       (varDeclSetInTerm(EVDS, T)
                        varDeclSetInTerm(EVDS, T'))),
                     CDS, I)), T''') .) EqS)
       RlS
     endm,
     EMAS', EqS', RlS', CDS,
     index(
       prepareLHS(T,
         varDeclSetDiff(EVDS,
           (varDeclSetInTerm(EVDS, T) varDeclSetInTerm(EVDS, T'))),
         CDS, I))) .
```

After completing the set of classes in the module with the attributes from their superclasses, the function `objectOrientedToSystemModule` calls the function `objectOrientedToSystemModuleAux` with the same module and the set of class declarations. The definition of the `objectOrientedToSystemModule` function is given by the five equations below.

```
op objectOrientedToSystemModule :
     StrOModule Database -> StrSModule .
op objectOrientedToSystemModuleAux :
     StrOModule ClassDeclSet -> StrSModule .

eq objectOrientedToSystemModule(
     omod MN is PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS endom,
     DB)
  = objectOrientedToSystemModuleAux(
      omod MN is
         PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS
      endom,
      prepareClasses(CDS, SCDS, EIL, DB)) .

eq objectOrientedToSystemModuleAux(
     omod MN is
        PL EIL (sorts ESS .) ESSDS ((class C | ADS .) CDS)
        SCDS EODS MDS EVDS EMAS EqS RlS endom,
     CDS')
  = objectOrientedToSystemModuleAux(
      omod MN is
         PL EIL (sorts (ESS ; C) .)
         (subsort C < 'Cid . ESSDS)
         CDS SCDS
         ((op eSortToQid(C) : nil -> C [none] .)
          attributeOpDeclSet(ADS) EODS)
         MDS EVDS EMAS EqS RlS
      endom,
      CDS') .
eq objectOrientedToSystemModuleAux(
```

```
      omod MN is
         PL EIL ESD ESSDS CDS ((subclass C < C' .) SCDS)
         EODS MDS EVDS EMAS EqS RlS
      endom,
      CDS')
   = objectOrientedToSystemModuleAux(
       omod MN is
          PL EIL ESD ((subsort C < C' .) ESSDS)
          CDS SCDS EODS MDS EVDS EMAS EqS RlS
       endom,
       CDS') .
 eq objectOrientedToSystemModuleAux(
       omod MN is
          PL EIL ESD ESSDS CDS SCDS EODS ((msg F : ESL -> ES .) MDS)
          EVDS EMAS EqS RlS
       endom,
       CDS')
   = objectOrientedToSystemModuleAux(
        omod MN is
           PL EIL ESD ESSDS CDS SCDS ((op F : ESL -> ES [none] .) EODS)
           MDS EVDS EMAS EqS RlS
        endom,
        CDS') .
 eq objectOrientedToSystemModuleAux(
      omod MN is
         PL EIL ESD ESSDS none none EODS none EVDS EMAS EqS RlS
      endom, CDS)
   = prepareAxioms(
       mod MN is PL EIL ESD ESSDS EODS EVDS none none none endm,
       EMAS, EqS, RlS, CDS, 0) .

  ...

endfm
```

7.7 Evaluation of Modules and Theories

As explained in Section 7.1, in our approach transforming a module from its possibly complex structured version to its unstructured form is a two-step process. First, all module expressions are evaluated, generating an intermediate form in which there are only simple inclusion relationships among the modules. This first step can be seen as the reduction of a structured specification to its structured *normal form*. Then, in a second step, this structured normal form is flattened into an unstructured specification. Note, however, that the importation of built-in modules is left explicit in the flattened form. The function `normalization` is in charge of normalizing the structure.

The process of evaluation of a preunit has to take into account the possibility of containing bubbles in it. Depending on whether it is dealing with a preunit or with a unit, the evaluation process is accomplished by two different functions, namely, `evalPreUnit` and `evalUnit`. One function or the other will be called in each case. Evaluating a module already in the database, which is done by `evalUnit`, does not require bubble handling. Besides this difference, both functions proceed in a similar way. Before presenting the functions `evalPreUnit` and `evalUnit` we introduce some auxiliary declarations.

```
fmod EVALUATION is
```

```
pr O-O-TO-SYSTEM-MOD-TRANSF .
pr MOD-EXPR-EVAL .
pr UNIT-BUBBLE-PARSING .
pr DECL-EXT-SORT-TO-QID .

sort UnitList .
subsort Unit < UnitList .

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

var QIL : QidList .
var M : Module .
vars PU U U' U'' : Unit .
var UL : UnitList .
var DB : Database .
var ME : ModExp .
var P : Parameter .
var PL : ParameterList .
var EIL : EImportList .
var ESD : ESortDecl .
var CDS : ClassDeclSet .
var ESSDS : ESubsortDeclSet .
var SCDS : SubclassDeclSet .
var EODS : EOpDeclSet .
var MDS : MsgDeclSet .
var EVDS : EVarDeclSet .
var EMAS : EMembAxSet .
var EqS : EquationSet .
var RlS : RuleSet .
```

The `subunitImportList` function returns the list of all the subunits of a given unit. It is called with the list of importations of the given unit as first argument, and proceeds recursively through its structure collecting all the subunits in it.

```
op subunitImportList : EImportList Database -> EImportList .
```

The function `getNonBuiltInUnitList` returns the list of units in the list of importations given as argument which are not built-in.

```
op getNonBuiltInUnitList : EImportList Database -> UnitList .
```

The function `selectBuiltInImports` returns the sublist of the importations of built-in modules.

```
op selectBuiltInImports : EImportList -> EImportList .
```

The normalization of a structure consists in evaluating each of the module expressions appearing in it. Note that, if the `evalModExp` function generates new modules, they will be evaluated using the `evalUnit` function, producing recursive calls on the part of the structure not previously normalized.

```
op normalization : EImportList ParameterList Database -> Database .
eq normalization(((including P .) EIL), PL, DB)
  = normalization(EIL, PL, DB) .
eq normalization(((including ME .) EIL), PL, DB)
  = normalization(EIL, PL, evalModExp(ME, PL, DB)) .
eq normalization(nil, PL, DB) = DB .
```

`checkSortClashes` checks whether the intersection of the two sets of sorts given as arguments is empty or not. If it is nonempty, then there is a clash of names, and a warning message is passed to the database. The check is very simple, and only reports the name of one of the modules from which the sorts come from. Only the name of the module from which the sorts given as second argument come from is known at this point. This is the module name given as first argument.

```
op checkSortClashes : ModName ESortSet ESortSet Database -> Database .
```

In the current system, the only transformation handled by the `transform` function is the one from object-oriented modules to system modules, which is accomplished by the `objectOrientedToSystemModule` function presented in Section 7.6. However, `transform` has been defined as a general transformation that could affect other kinds of modules in a future extension.

```
op transform : Unit Database -> Unit .

eq transform(
     omod MN is
        PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS
     endom, DB)
  = objectOrientedToSystemModule(
      omod MN is
         PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS
      endom, DB) .
```

The function `signature` generates a functional module of sort `FModule`, without equations, by "forgetting" the appropriate declarations and converting extended sorts and module names into quoted identifiers.

```
op signature : Unit -> FModule .
eq signature(unitError(QIL)) = unitError(QIL) .
ceq signature(U)
  = fmod modNameToQid(name(U)) is
       importList(U)
       sorts eSortToQid(sortSet(sortDecl(U))) .
       eSortToQid(subsortDeclSet(U))
       eSortToQid(opDeclSet(U))
       eSortToQid(varDeclSet(U))
       none
       none
    endfm
    if (U : StrTheory) or (U : StrModule) .
```

The function `flatUnit` generates a module of sort `Module` "forgetting" declarations and converting extended sorts and module identifiers into quoted identifiers.

```
op flatUnit : Unit -> Module .
eq flatUnit(unitError(QIL)) = unitError(QIL) .
ceq flatUnit(U)
  = if U : FUnit
    then (fmod modNameToQid(name(U)) is
             importList(U)
             sorts eSortToQid(sortSet(sortDecl(U))) .
             eSortToQid(subsortDeclSet(U))
             eSortToQid(opDeclSet(U))
             eSortToQid(varDeclSet(U))
```

```
            eSortToQid(membAxSet(U))
            equationSet(U)
         endfm)
   else (mod modNameToQid(name(U)) is
            importList(U)
            sorts eSortToQid(sortSet(sortDecl(U))) .
            eSortToQid(subsortDeclSet(U))
            eSortToQid(opDeclSet(U))
            eSortToQid(varDeclSet(U))
            eSortToQid(membAxSet(U))
            equationSet(U)
            ruleSet(U)
         endm)
   fi
   if (U : StrTheory) or (U : StrModule) .
```

The evaluation process for units without bubbles is as follows. After normalizing the structure, the function `evalUnit` calls `evalUnit1` with an empty copy of the module to which the list of declarations of importations of built-in modules is added, and with the list of its nonbuilt-in subunits.

`evalUnit1` accumulates all the declarations in all the nonbuilt-in submodules in the copy of the module passed as second argument. The top module is then introduced in the database, and, after calling the `transform` function and renaming all the variables in it, the internal version of such a module is entered in the database as well.

Finally, `evalUnit2` generates the signature and the flat version of the module and enters them in the database.

```
op evalUnit : Unit Database -> Database .
op evalUnit1 : Unit Unit UnitList Database -> Database .
op evalUnit2 : Unit Unit Database -> Database .

eq evalUnit(U, DB)
  = evalUnit1(U,
      setImportList(empty(U),
        selectBuiltInImports(
          subunitImportList(importList(U),
            normalization(importList(U), parameterList(U), DB)))),
      getNonBuiltInUnitList(
        subunitImportList(importList(U),
          normalization(importList(U), parameterList(U), DB)),
        normalization(importList(U), parameterList(U), DB)),
      normalization(importList(U), parameterList(U), DB)) .

eq evalUnit1(U, U', (U'' UL), DB)
  = evalUnit1(U, addDecls(U', setImportList(U'', nil)), UL,
      checkSortClashes(
        name(U), sortSet(sortDecl(U')), sortSet(sortDecl(U'')), DB)) .
eq evalUnit1(U, U', nil, DB)
  = evalUnit2(setImportList(transform(U, DB), nil), U',
      checkSortClashes(
        name(U'), sortSet(sortDecl(U)), sortSet(sortDecl(U')),
        insertInternalTopUnit(renameVars(transform(U, DB)),
          insertTopUnit(U, DB)))) .

eq evalUnit2(U, U', DB)
  = insertFlatUnit(name(U),
```

```
      flatUnit(addDecls(U, U')),
      insertSignature(name(U),
        signature(addDecls(U, U')), DB)) .
```

The function evalPreUnit has to take care of the bubbles in the unit. As we explained in Section 7.1, both the signature and the flattened version of the module are created simultaneously, completing the parsing of the bubbles once the signature has been built, and then completing the flattened module.

The evalPreUnit function takes as arguments two copies of the module and a database. We shall see in Section 9.3.1 how these two modules are generated; the one passed as first argument has still bubbles in it, while the other one, which will be used to build the signature, does not contain any bubbles. This module without bubbles is the result of removing the bubbles from the declarations in it, or of removing the declarations themselves when they contain bubbles, as in the case of equations, for example.

The evalPreUnit function is quite similar to the function evalUnit. First, the structure is normalized by calling the normalization function, and then all the subunits in the structure are collected (accomplished by subunitImportList and getNonBuiltInUnitList) and the list of importations is updated with the sublist of importations of built-in modules (selectBuiltInImports). Second, the structure of all the subunits below the top is flattened to a single unit. This unit is used to create a first version of the signature (without identity elements of operators) in which all the bubbles in the top preunit are parsed (solveBubbles). The final version of the signature and the flat unit are generated once the bubbles have been parsed. The 'internal' version of the module is also generated by renaming the variables in it (renameVars). All these versions of the module are finally entered in the database.

Note that if the META-LEVEL module is imported in the module being evaluated, a declaration importing the predefined module UP (see Section 6.1) is added. With the declarations in this module it will be possible to parse bubbles containing calls to the up functions [14, 26] in them.

```
op evalPreUnit : Unit Unit Database -> Database .
op evalPreUnit1 : Unit Unit UnitList Unit Database -> Database .
op evalPreUnit2 : Unit Unit Module Database -> Database .
op evalPreUnit3 : Unit Unit Module Database -> Database .

eq evalPreUnit(PU, U, DB)
  *** PU : top module with bubbles
  *** U  : top module without bubbles
  = evalPreUnit1(
      (if ((including 'META-LEVEL .) in
              subunitImportList(importList(PU),
                normalization(importList(PU), parameterList(PU), DB)))
       then addImportList((including 'UP .), PU)
       else PU
       fi),
      setName(empty(U), name(U)),
      getNonBuiltInUnitList(
        subunitImportList(
          importList(
            if ((including 'META-LEVEL .) in
                   subunitImportList(importList(PU), nil,
                     normalization(importList(PU), parameterList(PU), DB)))
            then addImportList((including 'UP .), PU)
```

```
              else PU
              fi),
            normalization(importList(PU), parameterList(PU), DB)),
          normalization(importList(PU), parameterList(PU), DB)),
        setImportList(U,
          selectBuiltInImports(
            subunitImportList(importList(PU),
              normalization(importList(PU), parameterList(PU), DB)))),
        normalization(importList(PU), parameterList(PU), DB)) .

  eq evalPreUnit1(PU, U, (U' UL), U'', DB)
    = evalPreUnit1(PU, addDecls(U, U'), UL, U'',
        checkSortClashes(
          name(U''), sortSet(sortDecl(U)), sortSet(sortDecl(U')), DB)) .
  ceq evalPreUnit1(PU, U, nil, U', DB)
    = evalPreUnit2(PU, U,
        signature(addDecls(transform(U', DB), setImportList(U, nil))),
        checkSortClashes(
          name(U'), sortSet(sortDecl(U)), sortSet(sortDecl(U')), DB))
      if (U : StrTheory) or (U : StrModule) .
  eq evalPreUnit1(PU, unitError(QIL), UL, U', DB) = warning(DB, QIL) .

  eq evalPreUnit2(PU, U, M, DB)
    *** PU : top module with bubbles
    *** U  : everything below
    *** M  : complete signature (with variables of the top module)
    = evalPreUnit3(
        solveBubbles(PU, addImportList((including 'BOOL .), M),
          ((including 'META-LEVEL .) in importList(M)), DB),
        U, M,
        insertTopUnit(
          solveBubbles(PU, addImportList((including 'BOOL .), M),
            ((including 'META-LEVEL .) in importList(M)), DB), DB)) .

  eq evalPreUnit3(PU, U, M, DB)
    *** PU : top module without bubbles
    *** U  : everything below
    *** M  : complete signature (with variables of the top module)
    = insertFlatUnit(name(PU),
        flatUnit(
          setImportList(addDecls(transform(PU, DB), U), importList(M))),
        insertSignature(name(PU),
          setOpDeclSet(M,
            opDeclSet(
              flatUnit(
                setImportList(
                  addDecls(transform(PU, DB), U), importList(M))))),
          insertInternalTopUnit(renameVars(transform(PU, DB)), DB))) .

   ...

endfm
```

Note that in both `evalUnit` and `evalPreUnit`, the function `transform` has to be invoked to transform the module into a functional or system module. In the current system, the only transformation available is from object-oriented modules to system modules.

7.8 Application of Map Sets

The following two modules deal with the application of a set of renaming maps to a module. Except for the proof obligations and additional checks associated with views—almost none of these checks is performed, and none of these proof obligations is generated in the current version—the way of applying a renaming map and a view map on a module is the same. Internally, they are treated in the same way; the only difference between them consists in the way of calling the function to accomplish this application.

Note that there might be some 'interference' between sort maps, and operator maps and message maps when they are applied. Let us consider for example a module with an operator declaration

```
op f : Foo -> Foo .
```

and a renaming map set

```
(sort Foo to Bar, op f : Foo -> Foo to g)
```

These renamings have to be applied carefully to avoid unintended behaviors. Depending on which of the maps is applied first, the other will be applicable or not. All the maps must be applied to the original module. To avoid the interference between the sort maps and other maps, the map set is divided into two sets: The first one contains the sort maps, and the second one contains the other maps.

We assume that there are no ambiguous mappings, that is, that we do not have, for example, maps `op f to g` and `op f to h`. In case of such ambiguity, one of the maps will be arbitrarily chosen.

7.8.1 Map Sets on Terms

The application of a set of view maps to a term is defined in the following module `VIEW-MAP-SET-APPL-ON-TERM`. The function `applyMapSetToTerm` is used to apply a given view map set to terms appearing in equations, rules, identity element declarations, and membership axioms, as part of the process of applying a map set to a unit.

```
fmod VIEW-MAP-SET-APPL-ON-TERM is
  pr UNIT .
  pr VIEW-MAP .
  pr EXT-SORT-TO-QID .
```

Note that all maps introduced in Sections 5.1 and 5.2, except for label maps, may affect a term. For example, sort maps will be applied to the qualifications of terms, and class and attribute maps have to be applied to the objects appearing in the term. Operator and message maps in which an explicit arity and coarity is given, and operator maps going to derived operators (see [14, Section 3.5.3]) must be applied to the complete family of subsort-overloaded operators.

The function `applyMapSetToTerm` takes as arguments two sets of view maps (the first set for sort maps, and the second for the other maps), the term to which the maps will be applied, and a module to be used in the matching of terms, sort comparisons, etc. Its declaration is as follows.

```
  op applyMapSetToTerm : ViewMapSet ViewMapSet Term Unit -> Term .

   ...

endfm
```

We do not include here the equations defining the semantics of the function `applyMapSetToTerm`. Instead, we present an example illustrating the meaning of the function. Renaming maps and view maps are discussed in [14, 26].

Let us consider the following configuration in the module `STACK2[Accnt]` presented in [14, Section 3.5.4] and in [26, Section 3.6.4]. In this configuration we have objects in the class `Accnt` which represent the accounts of different clients of a bank, which is represented as an object `'bank` of class `Stack[Accnt]`. The object `'bank` in the example configuration below keeps a stack with the accounts of the bank represented as a linked list of nodes, each of which corresponds to the account of one of the clients.

```
('bank push 'john)
('peter elt 2000)
< 'bank : Stack[Accnt] | first : o ('bank, 1) >
< 'paul : Accnt | bal : 5000 >
< 'peter : Accnt | bal : 2000 >
< 'mary : Accnt | bal : 7200 >
< 'john : Accnt | bal : 100 >
< o('bank, 0) : Node[Accnt] | node : 'peter, next : null >
< o('bank, 1) : Node[Accnt] | node : 'mary, next : o('bank, 0) > .
```

Let us apply the following renaming to the previous term.

```
(op o to id,
 class Stack[Accnt] to Bank,
 msg _push_ : Oid Oid -> Msg to open`account`in_to_,
 msg _pop to close`account`of_,
 msg _elt_ to _owns_dollars,
 attr node . Node[Accnt] to client,
 attr bal . Accnt to balance)
```

The resulting term is as follows.

```
(open account in 'bank to 'john)
('peter owns 2000 dollars)
< 'bank : Bank | first : id('bank, 1) >
< 'paul : Accnt | balance : 5000 >
< 'peter : Accnt | balance : 2000 >
< 'mary : Accnt | balance : 7200 >
< 'john : Accnt | balance : 100 >
< id('bank, 0) : Node[Accnt] | client : 'peter, next : null >
< id('bank, 1) : Node[Accnt] | client : 'mary, next : id('bank, 0) >
```

The function `applyMapSetToTerm` treats the object constructor `<_:_|_>` in a special way. It cannot be renamed, and, when an occurrence of such a constructor is found, class and attribute maps require a particular handling. Inside terms these maps are only triggered when this constructor is found, and they are applied in a very restricted way, according to the general pattern for objects. We assume that the operator `<_:_|_>` is only used for objects and that objects constructed using it are well-formed.

7.8.2 Map Sets on Units

The application of view maps to modules and theories of the different types is defined in the following module `VIEW-MAP-SET-APPL-ON-UNIT`. The function `applyMapSetToUnit` is defined recursively by applying it to the different components of a unit. When the terms in the different declarations are reached, the function `applyMapSetToTerm` is called. This call is made with the set of maps split conveniently, as explained above.

```
fmod VIEW-MAP-SET-APPL-ON-UNIT is
  pr VIEW-MAP-SET-APPL-ON-TERM .

  op applyMapSetToUnit : ViewMapSet Unit Module -> Unit .

   ...

endfm
```

7.9 Instantiation of Parameterized Modules and the `META-LEVEL` Module Expression

A parameterized module `M[`L_1` :: `T_1`, ..., `L_n` :: `T_n`]`, with $L_1 \ldots L_n$ labels and $T_1 \ldots T_n$ theory identifiers, is represented as a module with name `M` which contains parameter declarations `par` L_i `::` T_i for $1 \leq i \leq n$, and an importation declaration `including par` L_i `::` T_i `.` for each parameter L_i `::` T_i in its interface. Note that all modules are handled in a uniform way: nonparameterized modules and theories have their list of parameters set to `nilParameterList`.

The instantiation of the formal parameters of a parameterized module with actual modules or theories requires a view from each formal parameter theory to its corresponding actual unit. The process of instantiation results in the replacement of each interface theory by its corresponding actual parameter, using the views to bind actual names to formal names.

The naming conventions for sorts have to be taken into account in the instantiation process: every occurrence of a sort coming from a theory in the interface of a module must be qualified by its theory's label, and sorts defined in the body of a parameterized module can be parameterized by the labels in the interface of the module (see [14, Section 3.5.2] or [26, Section 3.6.2]).

The labeling convention for theories and for the sorts coming from them is very useful to avoid collisions of sort names coming from the parameter theories, and also to allow different uses of the same theory several times in the interface of a module. We assume that all sorts coming from the theory part of the parameter theories are used in their qualified form to manipulate the maps defined in the views before being applied to the body of the module being instantiated. If the target of a view is a theory, the sorts from the theory part of the target theory appearing in the targets of the maps in the view will be qualified as well, following the same convention.

When a parameterized module `M[`L_1` :: `T_1`, ..., `L_n` :: `T_n`]` is instantiated with views $V_1 \ldots V_n$, each parameterized sort `S[`L_1`,...,`L_n`]` in the body of the parameterized module is renamed to `S[`V_1`,...,`V_n`]`.

The discussion on the qualification of sorts in views before being used in the instantiation process applies in a completely similar way to class names in parameterized object-oriented modules.

It is possible to import a module expression in which a parameterized module is instantiated by some of the formal parameters of the parameterized module in which it is imported (see [14, Section 3.5.4] and [26, Section 3.6.4]). This is done by using the label of some of the parameters in the interface of a module, say L_k `::` T_k, in a module expression in which some parameterized module `N` with formal parameter T_k is instantiated with L_k, that is, we have the module expression `N[`$\ldots L_k \ldots$`]`. In this case, L_k is considered as the identity view for the theory T_k with L_k as name. Note that to be able to check whether a label in the interface of a module is used in an instantiation of this form, in the evaluation of a module expression the list of parameters of the module in which the module expression appears must be available. This is the reason why the `evalModExp` function was defined with `ParameterList` as one of the sorts in its arity (see Section 7.5). For module expressions appearing outside of any module, that is, in commands, etc., this list will be set to `nilParameterList`.

Note that this kind of instantiation may produce a 'cascade' effect. The module being instantiated may itself import module expressions in which labels of some of its parameter theories are used in the instantiation of some module expression in its body. This is handled by 'preparing' the module expressions appearing in the importation declarations of the module (`prepareImportList`). This process consists in changing the labels of the interface of the module being instantiated which are used in the importations of module expressions by the corresponding view names (`prepareModName`). After completing the generation of the module resulting from the evaluation of the module expression, this module will be evaluated with the `evalUnit` function, producing the evaluation of these new module expressions. In any extension of the language, new equations for the function `prepareModName` will have to be added for each new kind of module expression being defined.

In Sections 7.10 and 11 we shall see how new equations completing the semantics of `prepareModName` are added for each new module expression being defined. In the case of the renaming module expression, the renaming maps will have to be prepared as well, to adjust the sort names being renamed to the conventions discussed above.

As for any other module expression being defined, in addition to the operator declaration for the constructor of the instantiation module expression, equations completing the semantics of operators `evalModExp`, `modNameToQidList`, and `setUpModExpDependencies` have to be given.

```
fmod INST-EXPR is
  pr EVALUATION .
  pr VIEW-MAP-SET-APPL-ON-UNIT .
  inc MOD-EXPR .
  inc MOD-NAME .
```

We start by giving the new constructor for sort `ModExp`. Note that the modules `MOD-EXPR` and `MOD-NAME` have been imported in `including` mode.

```
  op _<_> : ModExp ViewExp -> ModExp .

  vars QI X Y W Z : Qid .
  var QIL : QidList .
  vars M M' : Module .
  vars U U' : Unit .
  vars ME ME' : ModExp .
  var MN : ModName .
  vars MNS MNS' : ModNameSet .
  vars VE VE' VE'' : ViewExp .
  vars VES VES' : ViewExpSet .
  var IS : InfoSet .
  var DB : Database .
  vars PL PL' PL'' : ParameterList .
  vars EIL EIL' EIL'' : EImportList .
  var VMAPS : ViewMapSet .
```

In the input given by the user, the operator `_[_]` is used both for the instantiation of module expressions, and for expressions parameterizing the module `META-LEVEL` with a list of module names. The function `evalModExp` distinguishes these two cases, calling the function `instantiation` in the former and the function `prepareMetalevel` in the latter.

```
  op instantiation : ModName ViewExp ParameterList Database -> Database .
  op prepareMetalevel : ViewExp Database -> Database .

  ceq evalModExp(_<_>(ME, VE), PL, DB)
    = if ME == 'META-LEVEL
      then prepareMetalevel(VE, DB)
      else instantiation(ME, VE, PL, evalModExp(ME, PL, DB))
      fi
      if not unitInDatabase(_<_>(ME, VE), DB) .
```

The function `prepareMetalevel` creates a new module with the module expression being evaluated as name, which imports the predefined `META-LEVEL` module. For each module name `I` in the list given as parameter of the expression, the declaration of a constant `I` of sort `Module` and an equation identifying such constant with the metarepresentation of the module with such name in the database are added to the module being created.

```
  op prepareMetalevelAux : ViewExp Unit Database -> Database .

  eq prepareMetalevel(VE, DB)
    = prepareMetalevelAux(VE,
        addImportList((including 'META-LEVEL .),
          setName(emptyStrFModule, _<_>('META-LEVEL, VE))), DB) .

  eq prepareMetalevelAux(_|_(QI, VE), U, DB)
    = if QI inModNameSet builtIns
      then prepareMetalevelAux(VE, U,
             warning(DB,
               ('WARNING: 'META-LEVEL '`[ QI '`] 'not 'supported)))
      else prepareMetalevelAux(VE,
             addOpDeclSet((op QI : nil -> 'Module [none] .),
               addEquationSet(
                 (eq {QI}'Module = up(getFlatUnit(QI, DB)) .), U)),
             DB)
      fi .
  eq prepareMetalevelAux(nullViewExp, U, DB) = evalUnit(U, DB) .
```

The function `labelInParameterList` checks whether the quoted identifier given as first argument is used as a label in the list of parameters given as second argument.

```
  op labelInParameterList : Qid ParameterList -> Bool .
```

The function `classDeclSetESortSet` returns the set of the names of the classes in a set of class declarations.

```
  op classDeclSetESortSet : ClassDeclSet -> ESortSet .
```

The following 'getTheory' functions return the corresponding elements in the theory part of the structure of the given unit. For example, the function `getTheorySortSet` returns the set of sorts declared in the "loose part" of the structure of the unit in the database having the name indicated as first argument.

```
  op getTheorySortSet : ModName Database -> ESortSet .
  op getTheoryClassSet : ModName Database -> ESortSet .
```

The 'get' functions return the corresponding elements in the structure of the given unit. For example, `getSortSet` returns all the sorts declared in the structure of the unit in the database having the name given as first argument.

```
op getSortSet : ModName Database -> ESortSet .
op getClassSet : ModName Database -> ESortSet .
```

In a parameterized module all occurrences of sorts or classes coming from the parameter theories have to be qualified [14, 26]. `createCopy` is the function used for creating these renamed copies of the parameters. As also explained in [14, 26], if a parameter theory is structured, the renaming is carried out not only at the top level, but for the entire "theory part" in the structure.

The function `createCopy` calls an auxiliary function, `prepareParameter`, which recursively proceeds through all the subtheories of the given theory. For each theory in the structure, the required set of maps is generated and applied to such a theory using the `applyMapSetToUnit` function discussed in Section 7.8.2, which is then evaluated and entered into the database. Note that the renamings to which a theory is subjected must also be applied to the theories importing it. The new database and the renaming maps applied to the theory will have to be returned by the function.

The function `prepareParameter` makes a copy of the theory specified by the name given as first argument and of all its subtheories (only theories, no modules), and qualifies all the sorts appearing in it with the label given in the declaration of the parameter, which is given as second argument.

```
op createCopy : ModExp Parameter Database -> Database .
op prepareParameter : ModName Qid Database -> Database .

eq createCopy(ME, (par X :: ME'), DB)
  = if ME =/= ME' or-else unitInDatabase((par X :: ME), DB)
    then DB
    else prepareParameter(ME, X, DB)
    fi .
```

When one of the labels of the interface of a module is being used in a module expression to instantiate some formal parameter of a module, then, in the evaluation of such module expression the qualification of all sorts and class names coming from the theory part of the parameter theory have to be changed according to such a label. In the evaluation of an instantiation module expression this is done by generating the corresponding renaming maps, which are then applied to the module being instantiated. Given labels `L` and `L'`, for each sort or class name `S` in the set given as argument, a map of the form `S.L to S.L'` is generated.

Since for the time being we do not handle parameterized theories, we assume that there can only be non-qualified sorts and classes in theories.

```
op generateMapSetQualifiedSortSet : Qid Qid QidSet -> MapSet .
op generateMapSetQualifiedClassSet : Qid Qid QidSet -> MapSet .
```

The function `prepareViewMapSet` takes the map set of a view and prepares it to be used in an instantiation by transforming sort and class names into their qualified form, if required (sorts and class names in a view have to be qualified only if they were defined in a theory).

The `prepareViewMapSet` function takes five arguments: The set of maps to be prepared, the label with which the sorts to be renamed have to be qualified, the set of sorts in the theory part of the source of the view, and the set of sorts and class names in the theory part of the target of the view.

Note that we assume that there is a sort map and a class map for each sort and class in the theory part of the source of the view. Therefore, sorts and class names appearing as sources of sort and class maps are systematically qualified. The sorts or class names used in the targets of the maps will be qualified only if they were declared in a theory. In maps for operators in which the arity and coarity are specified, or for those going to derived terms, the sorts appearing in the arity or coarity of an operator and those used to qualify terms, or in sort tests in terms, must also be qualified. However, in these cases the qualification cannot be done on all sorts, but only on those defined in the theory parts. This is the reason why the sets of sorts in the theory parts of the source and target and the set of class names in the target of the view are given when calling `prepareViewMapSet`.

```
op prepareViewMapSet :
      ViewMapSet Qid ESortSet ESortSet ESortSet -> ViewMapSet .
```

For each parameterized sort `S[`$\mathtt{L}_1$`,...,`$\mathtt{L}_n$`]` in the body of a parameterized module with $\mathtt{L}_1 \ldots \mathtt{L}_n$ the labels of the parameters in the interface of the module, a map of the form `sort S[`$\mathtt{L}_1$`,...,`$\mathtt{L}_n$`] to S[`$\mathtt{V}_1$`,...,`$\mathtt{V}_n$`]` is generated, where $\mathtt{V}_i$ is the name of the view associated to the label $\mathtt{L}_i$ in the set of pairs given as argument.

As we shall see below, in the instantiation process a set of pairs is created. Each of these pairs is composed of the label of a parameter theory and the view expression with which it is being instantiated. The sorts `QidViewExp`, `QidViewExpPair`, and `QidViewExpPairSet` are defined with their respective constructors. This set of pairs is used by the functions `generateMapSetSortSet` and `generateMapSetClassSet` to generate the maps mentioned above.

```
sort QidViewExpPair .
op <_,_> : Qid ViewExp -> QidViewExpPair .
op qid : QidViewExpPair -> Qid .
op viewExp : QidViewExpPair -> ViewExp .
eq qid(< QI , VE >) = QI .
eq viewExp(< QI , VE >) = VE .

sort QidViewExpPairSet .
subsort QidViewExpPair < QidViewExpPairSet .
op none : -> QidViewExpPairSet .
op __ : QidViewExpPairSet QidViewExpPairSet -> QidViewExpPairSet
     [assoc comm id: none] .

vars QIVEPS QIVEPS' : QidViewExpPairSet .

op generateMapSetSortSet : ESortSet QidViewExpPairSet -> MapSet .
op generateMapSetClassSet : ESortSet QidViewExpPairSet -> MapSet .
```

The function `prepareImportList` takes a list of importation declarations and a set of pairs composed of a label and a view name, and returns the list of importations resulting from changing in each of the module expressions the occurrences of the labels of the interface of the module being instantiated by the names of the views associated to them in the list of pairs.

```
op prepareImportList : EImportList QidViewExpPairSet -> EImportList .
op prepareModName : ModName QidViewExpPairSet -> ModName .
op prepareModNameAux : ModName ViewExp ViewExp ViewExp QidViewExpPairSet
      QidViewExpPairSet -> ModName .

eq prepareImportList(((including MN .) EIL), QIVEPS)
  = ((including prepareModName(MN, QIVEPS) .)
     prepareImportList(EIL, QIVEPS)) .
eq prepareImportList(nil, QIVEPS) = nil .
```

```
eq prepareModName(_<_>(ME, VE), QIVEPS)
  = prepareModNameAux(ME, VE, nullViewExp, nullViewExp, QIVEPS, QIVEPS) .
eq prepareModName(QI, QIVEPS) = QI .
eq prepareModName((par X :: ME), QIVEPS) = (par X :: ME) .

eq prepareModNameAux(ME, _|_(X, VE), VE', VE'', (< Y , Z > QIVEPS), QIVEPS')
  = if X == Y
    then prepareModNameAux(ME, VE, _|_(VE', Z), VE'', QIVEPS', QIVEPS')
    else prepareModNameAux(ME, _|_(X, VE), VE', VE'', QIVEPS, QIVEPS')
    fi .
eq prepareModNameAux(
      ME, _|_(X, VE), VE', VE'', (< Y , _;;_(W, Z) > QIVEPS), QIVEPS')
  = if X == Y
    then prepareModNameAux(
           ME, VE, _|_(VE', W), _|_(VE'', Z), QIVEPS', QIVEPS')
    else prepareModNameAux(ME, _|_(X, VE), VE', VE'', QIVEPS, QIVEPS')
    fi .
eq prepareModNameAux(ME, _|_(X, VE), VE', VE'', none, QIVEPS')
  = prepareModNameAux(ME, VE, _|_(VE', X), VE'', QIVEPS', QIVEPS') .
eq prepareModNameAux(ME, nullViewExp, VE, VE', QIVEPS, QIVEPS')
  = if VE' == nullViewExp
    then _<_>(ME, VE)
    else _<_>(_<_>(ME, VE), VE')
    fi .
```

The function `instantiation` calls the auxiliary function `instantiationAux`, which proceeds recursively on each of the parameters in the interface of the module being instantiated. For each view, a set of maps to be applied to the module is generated, which are accumulated in the third argument of the function.

In the base case, when there are no more parameters and no more views, the maps for the parameterized sorts are also generated, and all maps are then applied.

`instantiationAux` proceeds accumulating also the list of parameters being modified, the list of importations, and a list of label-view pairs (`QidViewExpPair`) associating each label in the interface to the view used in the instantiation of the theory with such label. This list of pairs is used to generate the set of maps of the parameterized sorts and to 'prepare' the list of importations as indicated above.

```
op instantiationAux : Unit Unit ViewMapSet ParameterList ParameterList
      EImportList EImportList ViewExp QidViewExpPairSet ParameterList
      Database -> Database .

eq instantiation(ME, VE, PL, DB)
  = instantiationAux(setName(getTopUnit(ME, DB), _<_>(ME, VE)),
      getSignature(ME, DB), none,
      parameterList(getTopUnit(ME, DB)), nilParameterList,
      importList(getTopUnit(ME, DB)), nil, VE, none, PL, DB) .

eq instantiationAux(U, M, VMAPS,
     parameterList((par X :: ME), PL), PL',
     (EIL (including (par X :: ME) .) EIL'), EIL'',
     _|_(QI, VE), QIVEPS, PL'', DB)
  = if labelInParameterList(QI, PL'')
    then instantiationAux(U, M,
           (VMAPS,
            generateMapSetQualifiedSortSet(X, QI,
              getTheorySortSet(ME, DB)),
```

```
             generateMapSetQualifiedClassSet(X, QI,
               getTheoryClassSet(ME, DB))),
            PL, parameterList(PL', (par QI :: ME)), (EIL EIL'),
            (EIL'' (including (par QI :: ME) .)),
            VE, (< X , QI > QIVEPS), PL'',
            createCopy(ME, (par QI :: ME), DB))
    else if viewInDatabase(QI, DB)
         then if getTopUnit(target(getView(QI, DB)), DB) : StrTheory
              then instantiationAux(U, M,
                     (VMAPS,
                      prepareViewMapSet(
                        mapSet(getView(QI, DB)), X,
                        getTheorySortSet(ME, DB),
                        getTheorySortSet(target(getView(QI, DB)), DB),
                        getTheoryClassSet(target(getView(QI, DB)), DB))),
                     PL,
                     parameterList(PL', (par X :: target(getView(QI, DB)))),
                     (EIL EIL'),
                     (EIL''
                      (including (par X :: target(getView(QI, DB))) .)),
                     VE,
                     (< X , _;;_(QI, X) > QIVEPS),
                     PL'',
                     createCopy(target(getView(QI, DB)),
                       (par X :: target(getView(QI, DB))), DB))
              else instantiationAux(U, M,
                     (VMAPS,
                      prepareViewMapSet(
                        mapSet(getView(QI, DB)), X,
                        getTheorySortSet(ME, DB), none, none)),
                     PL,
                     parameterList(
                       parameterList(
                         getTopUnit(target(getView(QI, DB)), DB)),
                       PL'),
                     (EIL EIL'),
                     (EIL'' (including target(getView(QI, DB)) .)),
                     VE, (< X , QI > QIVEPS),
                     PL'', DB)
              fi
         else warning(DB, ('\n 'ERROR: 'view QI 'not 'in 'database))
         fi
    fi .
eq instantiationAux(U, M, VMAPS, nilParameterList, PL, EIL, EIL',
     nullViewExp, QIVEPS, PL', DB)
  = evalUnit(
      setImportList(
        setParameterList(
          applyMapSetToUnit(
            (VMAPS,
             generateMapSetSortSet(
               (sortSet(sortDecl(U)) ;
                getSortSetAux(importList(U), DB)), QIVEPS),
             generateMapSetClassSet(
               (classDeclSetESortSet(classDeclSet(U)) ;
                getClassSetAux(importList(U), DB)), QIVEPS)),
            U, M),
```

```
          PL),
        (prepareImportList(EIL, QIVEPS) EIL')),
      DB) .
  eq instantiationAux(
       unitError(QIL), U, VMAPS, PL, PL', EIL, EIL', VE, QIVEPS, PL'', DB)
    = warning(DB, QIL) .
  eq instantiationAux(U, M, VMAPS, parameterList((par X :: ME), PL),
       PL', EIL, EIL', nullViewExp, QIVEPS, PL'', DB)
  = warning(DB, ('\n 'ERROR: 'incorrect 'module modNameToQidList(name(U)))) .
  eq instantiationAux(U, M, VMAPS, nilParameterList, PL, EIL, EIL',
       _|_(QI, VE), QIVEPS, PL', DB)
  = warning(DB, ('\n 'ERROR: 'incorrect 'module modNameToQidList(name(U)))) .
```

As pointed out in Section 4.4.2, for each new module expression constructor being introduced, we need to add equations for the operator `modNameToQid`. Since the function to transform view expressions into lists of quoted identifiers was already defined in Section 4.2.1, we just need to add the following equation.

```
  eq modNameToQidList(_<_>(ME, VE))
    = (modNameToQidList(ME) '`[ viewExpToQidList(VE) '`]) .
```

Similarly, we need to give equations for the `setUpModExpDependencies` function, presented in Section 6.2, on the instantiation module expression. Given a module expression of the form `ME < VE >` such that `ME` is in the database, we need to add `ME < VE >` to the set of names of the modules depending on `ME` and on `VE`. Since `VE` may be a composed view expression, we have to add the name of the module to each of the views in it. In this way, if `ME` or any of the views in `VE` is redefined or removed from the database, `ME < VE >` will be removed as well.

```
  eq setUpModExpDependencies(_<_>(ME, VE),
       database((< ME ; U ; U' ; M ; M' ; MNS ; VES > IS), MNS', VES', QIL))
    = viewExpDependencies(_<_>(ME, VE), VE,
        database((< ME ; U ; U' ; M ; M' ; MNS & _<_>(ME, VE) ; VES > IS),
          MNS', VES', QIL)) .
  ceq setUpModExpDependencies(_<_>(ME, VE), DB)
    = if ME inModNameSet builtIns
      then DB
      else warning(DB,
             ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
      fi
      if not unitInDatabase(ME, DB) .

  op viewExpDependencies : ModName ViewExp Database -> Database .
  ceq viewExpDependencies(MN, _|_(QI, VE),
        database((< QI ; VI ; MNS > IS), MNS', VES, QIL))
    = viewExpDependencies(MN, VE,
         database((< QI ; VI ; MNS & MN > IS), MNS', VES, QIL))
      if QI =/= nullViewExp .
  ceq viewExpDependencies(MN, _|_(QI, VE), DB)
    = DB
      if not viewInDatabase(QI, DB) .
  eq viewExpDependencies(MN, nullViewExp, DB) = DB .

   ...

endfm
```

7.10 Renaming of Units

In addition to the declaration of the constructor for renaming module expressions, the following module RENAMING-EXPR introduces equations to treat this new case in the definition of functions evalModExp, modNameToQidList, prepareModName, and setUpUnitDependencies.

A renaming expression is evaluated by applying the renaming maps, not only to the top unit, but also to the part of the structure *affected* by the maps. The renaming process propagates downwards in the unit hierarchy while the units in the structure are affected by the renamings. We say that a unit is affected by a set of maps (checked by the moduleAffected function) when any of the maps is applicable to any of the declarations in the unit, or in any of its subunits. The application of a set of maps to a single unit is accomplished by the applyMapSetToUnit function, discussed in Section 7.8.2.

```
fmod RENAMING-EXPR is
  pr DATABASE .
  pr VIEW-MAP-SET-APPL-ON-UNIT .
  pr EVALUATION .
  inc MOD-EXPR .
  pr MOD-EXPR-EVAL .
  pr INST-EXPR .
  pr MAP .
  pr DECL-EXT-SORT-TO-QID .

  op _*<_> : ModExp MapSet -> ModExp .

  vars ME ME' : ModExp .
  var MNS MNS' : ModNameSet .
  vars M M' : Module .
  vars U U' : Unit .
  var DB : Database .
  vars ES ES' : ESort .
  var QIL : QidList .
  vars VES VES' : ViewExpSet .
  var IS : InfoSet .
  var PL : ParameterList .
  vars EIL EIL' : EImportList .
  var VMAP : ViewMap .
  var VMAPS : ViewMapSet .
  var QIVEPS : QidViewExpPairSet .
```

The function createCopyRenaming creates a copy of the part of the structure of the specified module which is affected by the renaming, applying to each of the generated modules in the new structure the subset of maps affecting each one of them. The equation extending the evalModExp function to the renaming module expression is then reduced to a call to createCopyRenaming with the appropriate arguments.

```
  op createCopyRenaming : ModExp ModExp Unit Database -> Database .

  ceq evalModExp(_*<_>(ME, VMAPS), PL, DB)
    = createCopyRenaming(ME, _*<_>(ME, VMAPS),
        getSignature(ME, evalModExp(ME, PL, DB)),
        evalModExp(ME, PL, DB))
      if not unitInDatabase(_*<_>(ME, VMAPS), DB) .
```

The predicate `moduleAffected` checks whether the module with the name given as first argument in the database is affected by the set of maps given as second argument. A module is affected by a map set if any of the maps is applicable to the module or to any of its submodules.

```
op moduleAffected : ModName ViewMapSet Module Database -> Bool .
```

The function `mapSetRestriction` returns the subset of the view maps given as second argument that affect the given module.

```
op mapSetRestriction : Unit ViewMapSet Module Database -> ViewMapSet .

eq mapSetRestriction(U, (VMAP, VMAPS), M, DB)
  = if moduleAffectedAux(U, VMAP, M, DB)
    then (VMAP, mapSetRestriction(U, VMAPS, M, DB))
    else mapSetRestriction(U, VMAPS, M, DB)
    fi .
eq mapSetRestriction(U, none, M, DB) = none .
```

We proceed downwards while the set of maps affects the module, but we do so restricting the set of maps to the subset affecting the module. Since operator and message maps in which arity and coarity are specified must be applied to the whole subsort-overloaded family of operators or messages, we have to carry along the signature of the module at the top to make all the calls to the engine. Note that we may have maps of operations or messages with the domain given by sorts that are not in the submodules but which have other sorts in the submodules in the same connected components.

```
op applyMapSetRecursively :
      MapSet EImportList EImportList Unit Module Database -> Database .

eq createCopyRenaming(ME, _*<_>(ME, VMAPS), M, DB)
  = if not unitInDatabase(_*<_>(ME, VMAPS), DB)
    then applyMapSetRecursively(VMAPS, importList(getTopUnit(ME, DB)), nil,
           setName(applyMapSetToUnit(VMAPS, getTopUnit(ME, DB), M),
             _*<_>(ME, VMAPS)),
           M, DB)
    else DB
    fi .
eq createCopyRenaming(ME, ME', unitError(QIL), DB) = warning(DB, QIL) .

eq applyMapSetRecursively(VMAPS, ((including MN .) EIL), EIL', U, M, DB)
 = if MN inModNameSet builtIns or-else
      not moduleAffectedAux(getTopUnit(MN, DB), VMAPS,
            setVarDeclSet(M, eSortToQid(varDeclSet(getTopUnit(MN, DB)))),
            DB)
   then applyMapSetRecursively(VMAPS, EIL, ((including MN .) EIL'), U, M, DB)
   else applyMapSetRecursively(
          VMAPS, EIL,
          ((including
              (MN *< mapSetRestriction(getTopUnit(MN, DB), VMAPS,
                       setVarDeclSet(M,
                         eSortToQid(
                           varDeclSet(getTopUnit(MN, DB)))), DB) >) .)
           EIL'),
          U, M,
          createCopyRenaming(MN,
```

```
           (MN *< mapSetRestriction(getTopUnit(MN, DB), VMAPS,
                    setVarDeclSet(M,
                      eSortToQid(
                        varDeclSet(getTopUnit(MN, DB)))), DB) >),
           setVarDeclSet(M, eSortToQid(varDeclSet(getTopUnit(MN, DB)))),
           DB))
    fi .
  eq applyMapSetRecursively(VMAPS, nil, EIL, U, M, DB)
    = evalUnit(setImportList(U, EIL), DB) .
  eq applyMapSetRecursively(VMAPS, EIL, EIL', unitError(QIL), M, DB)
    = warning(DB, QIL) .
```

The definition of the function modNameToQidList on the renaming module expression has to take care of transforming into a quoted identifier list the set of view maps given in the module expression.

```
  op mapSetToQidList : MapSet -> QidList .

  eq modNameToQidList(_*<_>(ME, MAPS))
    = ((modNameToQidList(ME) '* '`( mapSetToQidList(MAPS) '`))) .
```

Let us now give the equations for setUpModExpDependencies on the renaming module expression. Given a module expression of the form ME *< VMAPS > such that ME is in the database, we just need to add ME *< VMAPS > to the set of names of the modules depending on ME. In this way, if ME is redefined or removed from the database, ME *< VMAPS > will be removed as well.

```
  eq setUpModExpDependencies(_*<_>(ME, VMAPS),
       database((< ME ; U ; U' ; M ; M' ; MNS ; VES > IS), MNS', VES', QIL))
    = database(
        (< ME ; U ; U' ; M ; M' ; MNS & _*<_>(ME, VMAPS) ; VES > IS),
        MNS', VES', QIL) .
  ceq setUpModExpDependencies(_*<_>(ME, VMAPS), DB)
    = warning(DB,
        ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
      if not unitInDatabase(ME, DB) .
```

The definition of the prepareModName function on a renaming module expression must take into account the possibility of having parameterized sorts or parameterized class names in the maps of a renaming module expression. The preparation of a renaming module expression must take into account this fact and prepare accordingly all parameterized sorts and classes appearing in it.

```
  op prepareMapSet : ViewMapSet QidViewExpPairSet -> ViewMapSet .

  eq prepareModName(_*<_>(ME, VMAPS), QIVEPS)
    = _*<_>(prepareModName(ME, QIVEPS), prepareMapSet(VMAPS, QIVEPS)) .
```

For example, for sort maps the equation is as follows.

```
  eq prepareMapSet(((sort ES to ES'), VMAPS), QIVEPS)
    = ((sort prepareESort(ES, QIVEPS) to prepareESort(ES', QIVEPS)),
       prepareMapSet(VMAPS, QIVEPS)) .

   ...

endfm
```

8 The Evaluation of Views

Before being entered into the database, besides containing bubbles, views have a somewhat different structure from that of the views given in Section 5.3. We introduce in the following module a sort `PreView` with constructor `view`, which is declared as the constructor for views of sort `View`, but with an additional argument, namely, a set of variable declarations to hold the declarations of variables in the view. During the processing of views (see Section 9.3.2), which takes place once the parsing process has concluded, these variables are associated with the corresponding maps where they are used, generating a term of sort `View`.

We start by introducing in the following module `PRE-VIEW-MAP` the sorts `TermPreMap`, `PreViewMap`, and `PreViewMapSet`. A preview map is a view map with bubbles. Note that the bubbles can only appear in term maps. Elements of sort `TermPreMap` are built with the constructor `preTermMap`, which takes two terms of sort `Term`, that is, two bubbles. In the processing of views (see Section 9.3.2), elements of sort `PreTermMap` will be converted into elements of sort `TermMap` by parsing the bubbles in them, and by associating to them the variables in them defined in the view in which the maps appear.

```
fmod PRE-VIEW-MAP is
  pr VIEW-MAP .

  sort TermPreMap .
  op preTermMap : Term Term -> TermPreMap .

  sorts PreViewMap PreViewMapSet .
  subsorts Map TermPreMap < PreViewMap .
  subsorts PreViewMap ViewMapSet < PreViewMapSet .
  op _,_ : PreViewMapSet PreViewMapSet -> PreViewMapSet
        [assoc comm id: none] .
```

Given a set of maps, the function `sortMapSet` returns the subset of sort maps in it.

```
  op sortMapSet : PreViewMapSet -> MapSet .

   ...

endfm

fmod PRE-VIEW is
  pr UNIT .
  pr PRE-VIEW-MAP .

  sort PreView .
  op view : ViewExp ParameterList ModExp ModExp EVarDeclSet PreViewMapSet
          -> PreView .

  op name : PreView -> ViewExp .
  op source : PreView -> ModExp .
  op target : PreView -> ModExp .
  op varDeclSet : PreView -> EVarDeclSet .
  op mapSet : PreView -> PreViewMapSet .

  var QI : Qid .
  vars ME ME' : ModExp .
```

The following functions can be used to add new declarations to the set of declarations already in a preview.

```
  op addMapSet : PreViewMapSet PreView -> PreView .
  op addVarDeclSet : EVarDeclSet PreView -> PreView .

  op emptyPreView : Qid ModExp ModExp -> PreView .
  eq emptyPreView(QI, ME, ME')
    = view(QI, nilParameterList, ME, ME', none, none) .

   ...

endfm
```

The function `solveBubbles` defined in the following `VIEW-BUBBLE-PARSING` module parses the bubbles in a set of preview maps. It takes two modules, the signature of the view's source theory, with the variables declared in the view, to parse the source term in the term maps, and the target theory, with the mappings of the variable declarations in the view, to parse the target terms.

```
fmod VIEW-BUBBLE-PARSING is
  pr BUBBLE-PARSING .
  pr PRE-VIEW .

  op solveBubbles : PreViewMapSet EVarDeclSet Unit Unit -> ViewMapSet .

   ...

endfm
```

9 Input/Output Processing

In this section we discuss how the preterm resulting from the call to the function `meta-parse` with the input and the top-level signature of Full Maude is transformed into a term of sort `Unit`, representing a preunit or a term of sort `PreView`. In the case of commands, they are evaluated giving the corresponding results in the appropriate form.

9.1 Input Parsing

Let us recall here the example presented in Section 2.5. Calling `meta-parse` with the module `NAT3` given there and the signature of Full Maude presented in Section 3, we obtain the following 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]]]]]]
```

Given each one of the subterms representing declarations in terms representing modules as the previous one, the function `parseDecl` generates the corresponding declaration, with no bubbles in it, and the corresponding predeclaration, with the bubbles appearing in the term. For example, for the term

```
'op_:_->_.['token[{''s_}'Qid],
           'neTokenList[{''Nat3}'Qid],
           'token[{''Nat3}'Qid]]
```

the following operator declaration is generated:

```
op 's_ : 'Nat3 -> 'Nat3 [none] .
```

Note that in this case, since the operator is declared without identity element, the declaration and the predeclaration generated by `parseDecl` coincide.

In the following sections we shall see how this approach is followed for declarations appearing in units and in views.

9.1.1 Parsing of Unit Declarations

The `parseDecl` function takes a term (which corresponds to a declaration that has to be parsed), a preunit (to which the parsed declaration with its bubbles in it will be added), and a unit (to which the parsed declaration without bubbles will be added to build up the signature). For example, a term corresponding to an unconditional equation, that is, a term of the form `'eq_=_.[T, T']` will be added to the set of equations of the preunit as `eq T = T' .`, but nothing will be added to the unit. Note that according to the signature used in the call to `meta-parse` (see Sections 3 and 10.2), `T` and `T'` are bubbles. Declarations of sorts, subsort relations, operators, classes, subclass relations, messages, and variables will be added to both of them. In the case of operator declarations, identity element attributes, which in general can be terms, are not included in the added declaration. As in Core Maude, declarations in a module can be given in any order, and therefore we follow a two-step approach consisting in first building the signature to parse the bubbles, and then generating the unit without bubbles in it. It could be different for other languages. For example, in some languages we may be able to assume that each operator and sort has been defined before being used, allowing then an incremental processing of the input.

```
fmod UNIT-DECL-PARSING is
  pr DATABASE .
  pr MOVE-DOWN .
  pr INST-EXPR .
  pr RENAMING-EXPR .

  vars PU U : Unit .
  vars T T' : Term .
```

Before introducing the `parseDecl` function, we present some auxiliary functions. For example, the following functions `parseSort`, `parseSortSet`, and `parseSortList` return, respectively, the sort, set of sorts, and list of sorts represented by the term given as argument. Notice that these functions, as most of the functions in this module, are partial functions. We assume that the term given as argument is in fact the representation of, for example, a valid sort, or set of sorts, etc. In the case of `parseDecl` we assume that the term is the representation of a predeclaration.

```
op parseSort : Term -> ESort .
op parseSortSet : Term -> ESortSet .
op parseSortList : Term -> ESortList .
op parseViewExp : Term -> ViewExp .

eq parseSort('sortToken[T]) = downQid(T) .
eq parseSort('_`[_`][T, T'])
  = eSort(parseSort(T), parseViewExp(T')) .

eq parseSortSet('__[T, T'])
  = (parseSortSet(T) ; parseSortSet(T')) .
eq parseSortSet('sortToken[T]) = parseSort('sortToken[T]) .
eq parseSortSet('_`[_`][T, T']) = parseSort('_`[_`][T, T']) .

eq parseSortList('__[T, T'])
  = (parseSortList(T) parseSortList(T')) .
eq parseSortList('sortToken[T]) = parseSort('sortToken[T]) .
eq parseSortList('_`[_`][T, T']) = parseSort('_`[_`][T, T']) .

eq parseViewExp('viewToken[T]) = downQid(T) .
eq parseViewExp('_`,_[T, T'])
  = _|_(parseViewExp(T), parseViewExp(T')) .
```

The function parseModExp takes a term representing a module expression and returns the corresponding term in sort ModExp. In case of adding new constructors for module expressions, as it will be done in Section 11, new equations defining the semantics of the function on them will have to be given.

```
op parseModExp : Term -> ModExp .
op parseMapSet : Term -> MapSet .

eq parseModExp('token[T]) = downQid(T) .
eq parseModExp('`(_`)[T]) = parseModExp(T) .
eq parseModExp('_`[_`][T, T'])
  = _<_>(parseModExp(T), parseViewExp(T')) .
eq parseModExp('_*`(_`)[T, T'])
  = _*<_>(parseModExp(T), parseMapSet(T')) .
```

Given a term representing a machine integer, the function parseMachineInt returns the corresponding integer.

```
op parseMachineInt : Term -> MachineInt .
```

Given a term representing a declaration or a predeclaration, the function parseDecl must generate and update both the unit and the preunit that it takes as arguments. Note that in the case of rules, for example, only a prerule is generated.

Since the preunit and the unit may be modified, they have to be returned as a pair, which will be used to extract the corresponding arguments for the following calls. Note that the parseDecl functions are in fact partial functions. Each parsing function assumes that it is possible to parse the given term.

We only include here some of the equations needed.

```
sort parseDeclResult .
op <_;_> : Unit Unit -> parseDeclResult .
op preUnit : parseDeclResult -> Unit .
op unit : parseDeclResult -> Unit .
```

```
  eq preUnit(< PU ; U >) = PU .
  eq unit(< PU ; U >) = U .

  op parseDecl : Term Unit Unit -> parseDeclResult .
```

In the case of importation declarations, since internally only the `including` mode is handled, all importations are generated in this mode, independently of the keyword used in the input.

```
  eq parseDecl('including_.[T], PU, U)
    = < addImportList((including parseModExp(T) .), PU) ; U > .
  eq parseDecl('protecting_.[T], PU, U)
    = < addImportList((including parseModExp(T) .), PU) ; U > .

  eq parseDecl('sorts_.[T], PU, U)
    = < addSortSet(parseSortSet(T), PU) ;
        addSortSet(parseSortSet(T), U) > .

  eq parseDecl('eq_=_.[T, T'], PU, U)
    = < addEquationSet((eq T = T' .), PU) ; U > .
  eq parseDecl('ceq_=_if_.[T, T', T''], PU, U)
    = < addEquationSet((ceq T = T' if T'' = {'true}'Bool .), PU) ; U > .

   ...

endfm
```

9.1.2 Parsing of View Declarations

A similar approach is followed for the parsing of declarations in views.

```
fmod VIEW-DECL-PARSING is
  pr PRE-VIEW .
  pr VIEW .
  pr UNIT .
  pr UNIT-DECL-PARSING .

  vars T T' : Term .
  var EODS : EOpDeclSet .
  var MDS : MsgDeclSet .
  var M : Module .
```

Operator and message name maps of the form `F to F'` are substituted by an equivalent set of maps of the form `F : ESL -> ES to F'`. One of these maps is added for each family of subsort-overloaded operators in the source theory of the view.

The following functions `generateOpMapSet` and `generateMsgMapSet` take, respectively, an operator and a message map of the form `F to F'`, a set of operator or message declarations, and a term of sort `Module`, and return, respectively, a set of operator maps and a set of message maps, with each of the members of those sets having the general form `F : ESL -> ES to F'`. One of these maps is generated for each family of subsort-overloaded operators or messages with name `F` in the module given as argument.

```
  op generateOpMapSet : Map EOpDeclSet Module -> MapSet .
  op generateMsgMapSet : Map MsgDeclSet Module -> MapSet .
```

In the case of views, the parseDecl function takes the term representing the corresponding declaration and a preview in which the declarations are introduced. Note that in the case of views, the approach followed in the evaluation is somewhat different. The only predeclarations in a preview correspond to the term premaps of sort PreTermMap, for which, in addition to solving the bubbles in them, we have to convert them into term maps of sort TermMap associating to them the set of declarations of variables in the view which are used in them (see Section 9.3.2).

The function parseDecl for declarations in views takes then the term representing such declaration and a preview in which the result of adding the declaration will be returned. To be able to generate the sets of equivalent operator and message maps as indicated above, the function takes also as parameters the sets of declarations of operators and messages in the theory part of the source theory of the view in question, and the signature of such theory to make the necessary sort comparisons.

```
  op parseDecl : Term PreView EOpDeclSet MsgDeclSet Module -> PreView .

  eq parseDecl('op_to`term_.[T, T'], PV, EODS, MDS, M)
    = addMapSet(preTermMap(T, T'), PV) .

  eq parseDecl('sort_to_.[T, T'], PV, EODS, MDS, M)
    = addMapSet((sort parseSort(T) to parseSort(T')), PV) .

   ...

endfm
```

9.2 Meta Pretty Printing

To be able to show to the user the modules, theories, views, and terms resulting from the different commands, the built-in function meta-pretty-print is extended in the modules in this section to deal with units and views.

9.2.1 Meta Pretty Printing of Declarations

The predefined function meta-pretty-print is extended in the following module DECL-META-PRETTY-PRINT to handle any declaration that can appear in a unit. Note that the following meta-pretty-print functions, as the built-in one, return a list of quoted identifiers, although only for the declarations containing terms—such as equations, rules, operator declarations with an identity attribute, etc.—they have been defined with a term of sort Module as argument. In the other cases the module is not necessary.

```
fmod DECL-META-PRETTY-PRINT is
  pr EXT-DECL .
  pr O-O-DECL .
  pr EXT-SORT-TO-QID .

  op meta-pretty-print : ESort -> QidList .
  op meta-pretty-print : ESortSet -> QidList .
  op meta-pretty-print : ESortList -> QidList .
  op meta-pretty-print : ESubsortDeclSet -> QidList .
  op meta-pretty-print : ClassDeclSet -> QidList .
  op meta-pretty-print : SubclassDeclSet -> QidList .
  op meta-pretty-print : Module EOpDeclSet -> QidList .
  op meta-pretty-print : MsgDeclSet -> QidList .
```

```
  op meta-pretty-print : EVarDeclSet -> QidList .
  op meta-pretty-print : Module EMembAxSet -> QidList .
  op meta-pretty-print : Module EquationSet -> QidList .
  op meta-pretty-print : Module RuleSet -> QidList .

   ...

endfm
```

9.2.2 Meta Pretty Printing of Units

In the following module, the `meta-pretty-print` function is defined on sort `Unit`.

```
fmod UNIT-META-PRETTY-PRINT is
  pr UNIT .
  pr DECL-META-PRETTY-PRINT .

  op meta-pretty-print : Unit Unit -> QidList .

   ...

endfm
```

The function `meta-pretty-print` on units is defined recursively, calling the `meta-pretty-print` functions for the different declarations in the unit defined in module `DECL-META-PRETTY-PRINT`.

9.2.3 Meta Pretty Printing of Maps and Views

We define in the following module the function `meta-pretty-print` on maps.

```
fmod MAP-SET-META-PRETTY-PRINT is
  pr DECL-META-PRETTY-PRINT .
  pr MAP .
  pr UNIT .

  op meta-pretty-print : MapSet -> QidList .

   ...

endfm
```

Finally, in the `VIEW-META-PRETTY-PRINT` module, the `meta-pretty-print` function is defined on views.

```
fmod VIEW-META-PRETTY-PRINT is
  pr DATABASE .
  pr MAP-SET-META-PRETTY-PRINT .
  pr VIEW-MAP-SET-APPL-ON-UNIT .
  pr DECL-EXT-SORT-TO-QID .

  op meta-pretty-print : Database View -> QidList .

   ...

endfm
```

9.3 Input Processing

The processing functions presented in the following modules are in charge of taking each term generated by the meta-parse function and, after transforming it into an element of the data types Unit or View, or generating some output, returning the database resulting from introducing in it such a term. We shall see in Section 10.1 how the appropriate function is called after having performed a first analysis of the term, in which it is detected whether the input corresponds to a unit, view, or command. In the cases of units and views the processing is quite similar. After a preprocessing of the term, the function parseDecl is called with each of the subterms representing declarations, resulting in units or views with the parsed declarations in it.

9.3.1 Unit Processing

The processing of a term resulting from the parsing of some input corresponding to a unit is accomplished by the processUnit function. This function takes as arguments a term of sort Term, which represents some preunit, and a database. The function then enters into the given database the unit obtained from the transformation of such term into a term of sort Unit.

```
fmod UNIT-PROCESSING is
  pr DATABASE .
  pr UNIT-DECL-PARSING .
  pr EVALUATION .
  pr VIEW-MAP-SET-APPL-ON-UNIT .

  vars F X : Qid .
  vars PU U U' : Unit .
  var DB : Database .
  vars T T' T'' : Term .
  var TL : TermList .
  vars PL PL' : ParameterList .
  var EIL : EImportList .
  var ME : ModExp .
```

The parseParameterList takes a term representing a list of parameters and returns the corresponding list.

```
  op parseParameterList : Term -> ParameterList .
  eq parseParameterList('_::_['token[T], T'])
    = (par downQid(T) :: parseModExp(T')) .
  eq parseParameterList('_`,_[T, T'])
    = parameterList(parseParameterList(T), parseParameterList(T')) .
```

The function processParameters takes a preunit and a database and returns the resulting premodule and database, after processing each of the parameters in the parameter list of the preunit. For each of the parameters an importation declaration is added to the module, and the module expression used as parameter theory is evaluated in the database. The function returns a pair composed by a preunit and a database (UnitDatabasePair) as its result.

The function processParametersAux proceeds recursively on each of the parameters in the list of parameters of the given module. For each parameter of the form par X :: ME, the module expression ME is evaluated using the evalModExp function, which was discussed in Section 7.5, and then a copy of such a theory is created using the createCopy function, which applies to the unit the appropriate transformation, as explained in Section 7.9.

```
sort UnitDatabasePair .
op <_;_> : Unit Database -> UnitDatabasePair .
op preUnit : UnitDatabasePair -> Unit .
op database : UnitDatabasePair -> Database .

eq database(< PU ; DB >) = DB .
eq preUnit(< PU ; DB >) = PU .

op processParameters : Unit Database -> UnitDatabasePair .
op processParametersAux : Unit ParameterList ParameterList EImportList
     Database -> UnitDatabasePair .

eq processParameters(PU, DB)
  = processParametersAux(PU, parameterList(PU), parameterList(PU), nil, DB) .

eq processParametersAux(PU, PL, parameterList((par X :: ME), PL'), EIL, DB)
  = processParametersAux(PU, PL, PL', ((including (par X :: ME) .) EIL),
      createCopy(ME, (par X :: ME), evalModExp(ME, PL, DB))) .
eq processParametersAux(PU, PL, nilParameterList, EIL, DB)
  = < addImportList(EIL, PU) ; DB > .
```

All the operators declared as constructors of sort PreUnit in the signature of Full Maude, given in Appendix A, are declared with two arguments, namely, the name, or name and interface, of the unit, and the list of declarations of such units. The function processUnit2 is called with the term corresponding to the name, or name and interface, of the module as first argument, the term corresponding to the set of declarations as second argument, and an empty module of the appropriate type, in which the different declarations will be accumulated, as third argument.

The task of the function processUnit3 is then to make a second level parsing of the input, building up, simultaneously, the preunit represented in the term passed as argument, and the unit resulting from the declarations without bubbles. This unit without bubbles will be used by the evalPreUnit function to build the signature with which to analyze the bubbles in the preunit (see Section 7.7).

The case of parameterized modules requires a special treatment of the parameters. These parameters are evaluated and are added as submodules in the appropriate way.

When the last declaration is parsed, the function evalPreUnit is called with the preunit (the top module with bubbles) as first argument, the empty copy of it as second argument, the top module without bubbles as third argument, and the database.

Note that the processUnit function adds a declaration importing the module CONFIGURATION, presented in Section 6.1, to the object-oriented modules, and that processUnit3 adds a declaration importing the built-in module BOOL to all modules.

```
op processUnit : Term Database -> Database .
op processUnit2 : Term Term Unit Database -> Database .
op processUnit3 : Term Unit Unit Database -> Database .

eq processUnit('fmod_is_endfm[T, T'], DB)
  = processUnit2(T, T', emptyStrFModule, DB) .
eq processUnit('mod_is_endm[T, T'], DB)
  = processUnit2(T, T', emptyStrSModule, DB) .
eq processUnit('omod_is_endom[T, T'], DB)
  = processUnit2(T, T',
      addImportList((including 'CONFIGURATION .), emptyStrOModule), DB) .
eq processUnit('fth_is_endfth[T, T'], DB)
```

```
    = processUnit2(T, T', emptyStrFTheory, DB) .
  eq processUnit('th_is_endth[T, T'], DB)
    = processUnit2(T, T', emptyStrSTheory, DB) .
  eq processUnit('oth_is_endoth[T, T'], DB)
    = processUnit2(T, T',
        addImportList((including 'CONFIGURATION .), emptyStrOTheory), DB) .

  eq processUnit2('token[T], T', U, DB)
    = if downQid(T) inModNameSet builtIns
      then warning(DB, ('ERROR: 'built-in 'modules 'can 'not 'be 'redefined))
      else processUnit3(T',
             setName(U, downQid(T)),
             setName(U, modNameToQid(downQid(T))), DB)
      fi .
  eq processUnit2('_`[_`]['token[T], T'], T'', U, DB)
    = if downQid(T) inModNameSet builtIns
      then warning(DB, ('ERROR: 'built-in 'modules 'can 'not 'be 'redefined))
      else processUnit3(T'',
             preUnit(
               processParameters(
                 setParameterList(setName(U, downQid(T)),
                   parseParameterList(T')), DB)),
             setName(U, modNameToQid(downQid(T))),
             database(
               processParameters(
                 setParameterList(setName(U, downQid(T)),
                   parseParameterList(T')), DB)))
      fi .

  eq processUnit3('__[T, T'], PU, U, DB)
    = processUnit3(T', preUnit(parseDecl(T, PU, U)),
        unit(parseDecl(T, PU, U)), DB) .
  ceq processUnit3(F[TL], PU, U, DB)
    = evalPreUnit(
        preUnit(
          parseDecl(F[TL], addImportList((including 'BOOL .), PU), U)),
        unit(parseDecl(F[TL], addImportList((including 'BOOL .), PU), U)),
        DB)
      if F =/= '__ .

endfm
```

9.3.2 View Processing

A similar process is followed for views. Note that in case of operator maps going to derived terms we have bubbles, which will have to be treated using the signatures of the appropriate modules.

```
fmod VIEW-PROCESSING is
  pr UNIT-PROCESSING .
  pr VIEW-DECL-PARSING .
  pr VIEW-BUBBLE-PARSING .

  var F : Qid .
  vars T T' T'' T''' : Term .
  var M : Module .
  var VE : ViewExp .
```

```
var PV : PreView .
vars ME ME' : ModExp .
var DB : Database .
var EODS : EOpDeclSet .
var MDS : MsgDeclSet .
var TL : TermList .
```

As the functions getTheorySortSet and getTheoryClassSet presented in Section 7.9, the functions getTheoryOpDeclSet and getTheoryMsgDeclSet return, respectively, the set of declarations of operators, and the set of declarations of messages in the theory part of the structure of the module given as argument.

```
op getTheoryOpDeclSet : ModName Database -> EOpDeclSet .
op getTheoryMsgDeclSet : ModName Database -> MsgDeclSet .
```

The processing of terms representing previews accomplished by the function processView is quite similar to the one accomplished by processUnit on terms representing preunits. The algorithms followed are also quite similar. Both proceed recursively on the list of declarations, accumulating them in a preunit or in a preview.

The solving of bubbles in views requires the signatures of the source and target units extended, respectively, with the declarations of variables in the view and with the mappings of these declarations. As we shall see in Section 6.2, the signatures of the built-in modules are not accesible at the metalevel, and thus built-in modules cannot be used directly as arguments of built-in functions. Thus, to be able to use them as targets of views, a 'dummy' module is created importing the corresponding predefined module. The source and target module expressions of the view are evaluated before the view processing itself starts.

As we saw in Section 9.1.2, parsing of terms representing operator and message maps requires the set of operator and message declarations in the theory part of the source theory.

```
op processView : Term Database -> Database .
op processView : Term ViewExp ModExp ModExp Database -> Database .
op processViewAux :
      Term PreView EOpDeclSet MsgDeclSet Module Database -> Database .

eq processView('view_from_to_is_endv['viewToken[T], T', T'', T'''], DB)
  = processView(T''',  downQid(T), parseModExp(T'), parseModExp(T''), DB) .

eq processView(T, VE, ME, ME', DB)
  = processViewAux(T,
      emptyPreView(VE, ME, ME'),
      getTheoryOpDeclSet(ME, evalModExp(ME, DB)),
      getTheoryMsgDeclSet(ME, evalModExp(ME, DB)),
      getSignature(ME, evalModExp(ME, DB)),
      evalModExp(ME', evalModExp(ME, DB))) .

eq processViewAux('__[T, T'], PV, EODS, MDS, M, DB)
*** - EODS and MDS are, respectively, the set of operation and message
***   declarations in the theory part of the source.
*** - M is the signature of the source theory.
  = processViewAux(T', parseDecl(T, PV, EODS, MDS, M), EODS, MDS, M, DB) .
ceq processViewAux(F[TL], PV, EODS, MDS, M, DB)
  = insertView(
      view(name(parseDecl(F[TL], PV, EODS, MDS, M)),
```

```
          nilParameterList,
          source(parseDecl(F[TL], PV, EODS, MDS, M)),
          target(parseDecl(F[TL], PV, EODS, MDS, M)),
          solveBubbles(
            mapSet(parseDecl(F[TL], PV, EODS, MDS, M)),
            varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M)),
            addVarDeclSet(varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M)), M),
            (if target(parseDecl(F[TL], PV, EODS, MDS, M))
                  inModNameSet builtIns
             then (fmod 'DUMMY is
                     including target(parseDecl(F[TL], PV, EODS, MDS, M)) .
                     sorts none .
                     none
                     none
                     applyMapSetToVarDeclSet(
                       sortMapSet(
                         mapSet(parseDecl(F[TL], PV, EODS, MDS, M))),
                           varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M)))
                     none
                     none
                   endfm)
             else addVarDeclSet(
                    applyMapSetToVarDeclSet(
                      sortMapSet(mapSet(parseDecl(F[TL], PV, EODS, MDS, M))),
                      varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M))),
                    getSignature(target(parseDecl(F[TL], PV, EODS, MDS, M)),
                      DB))
             fi))),
        DB)
      if F =/= '__ .

   ...

endfm
```

9.3.3 Command Processing

The function `processCommand` only handles the `reduce`, `rewrite`, and `down` commands. The other commands are directly evaluated by the rules for the top-level handling of the database (see Section 10.1). The `processCommand` function takes a term, which represents one of these commands, the name of the default module, and a database. The result is a list of quoted identifiers representing the result of the evaluation of the command that will be placed in the read-eval-print loop to be printed in the terminal.

The `reduce` and `rewrite` commands are basically evaluated calling the built-in functions `meta-reduce` and `meta-rewrite`, respectively. These functions are called with the appropriate modules. In the case of commands in which an explicit module is not specified the default module is used.

The preparation of the output for these functions becomes more complex when the `down` command is used. To deal with the `down` command, an auxiliary function `processCommand2` is introduced, returning the term resulting from the evaluation of the command.

```
fmod COMMAND-PROCESSING is
  pr UNIT-PROCESSING .
  pr UNIT-META-PRETTY-PRINT .
```

```
  vars T T' T'' : Term .
  var DB : Database .
  var M : Module .
  var MN : ModName .

  op processCommand : Term ModExp Database -> QidList .
  op processReduce : ModExp Module Term Database -> QidList .
  op processReduce : ModExp Unit Term Database -> QidList .
  op processRewrite : ModExp Module Term MachineInt Database -> QidList .
  op processRewrite : ModExp Unit Term MachineInt Database -> QidList .
  op processCommand2 : ModExp Term Module Database -> Term .
  op processReduce2 : ModExp Module Term Database -> Term .
  op processRewrite2 : ModExp Module Term MachineInt Database -> Term .
```

To illustrate the processing of commands, we give here the equations needed for the handling of the `red_.` and `down_:_.` commands.

```
  eq processCommand('down_:_[T, T'], MN, DB)
    = if processCommand2(MN, T', getFlatUnit(MN, DB), DB) =/= error*
      then ('\n 'Result
            meta-pretty-print(
              leastSort(getFlatUnit(parseModExp(T), DB),
                downTerm(processCommand2(MN, T', getFlatUnit(MN, DB), DB))))
            ':
            meta-pretty-print(getFlatUnit(parseModExp(T), DB),
              downTerm(processCommand2(MN, T', getFlatUnit(MN, DB), DB))))
      else ('\n 'ERROR: 'Incorrect 'input '.)
      fi .
  eq processCommand('red_.['bubble[T]], MN, DB)
    = processReduce(MN, getFlatUnit(MN, DB), 'bubble[T], DB) .

  eq processCommand2(MN, 'down_:_[T, T'], M, DB)
    = downTerm(processCommand2(MN, T', M, DB)) .

  eq processCommand2(MN, 'red_.['bubble[T]], M, DB)
    = processReduce2(MN, M, 'bubble[T], DB) .

  eq processReduce(MN, M, T, DB)
    = if meta-reduce(M, solveBubbles(T, M, true, DB)) =/= error*
      then ('\n 'Reduce 'in meta-pretty-print(MN) ':
            meta-pretty-print(M, solveBubbles(T, M, true, DB)) '.
            '\n 'Result
            meta-pretty-print(
              leastSort(M, meta-reduce(M, solveBubbles(T, M, true, DB)))) ':
            meta-pretty-print(M,
               meta-reduce(M, solveBubbles(T, M, true, DB))))
      else ('ERROR: 'Incorrect 'Command '.)
      fi .

  eq processReduce2(MN, M, T, DB)
    = meta-reduce(M, solveBubbles(T, M, true, DB)) .

   ...

endfm
```

10 Interaction with the Persistent Database

In the case of Full Maude, the persistent state of the system is given by a single object which maintains the database of the system. This object has an attribute `db`, to keep the actual database in which all the modules being entered are stored, an attribute `default`, to keep the identifier of the current module by default, and attributes `input` and `output` to simplify the communication of the read-eval-print loop given by the `LOOP-MODE` module with the database. Using the notation for classes in object-oriented modules (see [14, Section 3.2] and [26, Section 3.3]) we can declare the class `database` as follows:

```
  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 .
```

10.1 Top Level Handling of the Persistent Database

Note that, since the Full Maude specification is given as a system module in Core Maude, object-oriented declarations cannot be given directly. Instead, the equivalent declarations desugaring the desired object-oriented declarations have to be specified. We use also the same conventions discussed in [14, Section 3.2.2] and in [26, Section 3.3.2] regarding the use of variables instead of class names in the objects and in the addition of variables of sort `AttributeSet` to range over the additional attributes. As demostrated in [26, Chapter 5], this convention will allow us to extend the Full Maude system in a very simple and clean way.

To allow the use of the object-oriented notation the predefined module `CONFIGURATION`, presented in [14, Section 3.2.2] and in [26, Section 3.3.2], is included in the following module `DATABASE-HANDLING`.

```
mod DATABASE-HANDLING is
  inc META-LEVEL .
  inc CONFIGURATION .
  pr VIEW-META-PRETTY-PRINT .
  pr VIEW-PROCESSING .
  pr COMMAND-PROCESSING .

  var F : Qid .
  var QIL : QidList .
  vars T T' T'' T''' : Term .
  var TL : TermList .
  var DB : Database .
  var MN : ModName .
```

We start introducing a subsort `DatabaseClass` of sort `Cid`, the operator declarations necessary for representing objects in class `DatabaseClass` as defined above, and variables to range over subclasses of class `DatabaseClass` and over attributes.

```
  sort DatabaseClass .
  subsort DatabaseClass < Cid .
```

```
op Database : -> DatabaseClass .
op db :_ : Database -> Attribute .
op input :_ : TermList -> Attribute .
op output :_ : QidList -> Attribute .
op default :_ : ModName -> Attribute .

var Atts : AttributeSet .
var X@Database : DatabaseClass .
var O : Oid .
```

Next, we introduce an auxiliary function `parseModName` to parse names of user-defined modules, and a constant `nilTermList` of sort `TermList`. Note that the name of a user-defined module must be a single identifier (a token) or, for parameterized modules, its name—a single identifier— and its interface.

```
op parseModName : Term -> Qid .
eq parseModName('token[T]) = downQid(T) .
eq parseModName('_`[_`]['token[T], T']) = downQid(T) .

op nilTermList : -> TermList .

eq (nilTermList, TL) = TL .
eq (TL, nilTermList) = TL .
```

Finally, we present the rules processing the inputs of the database. These rules define the behavior of the system for the different commands, modules, theories, and views entered into the system. For example, the first rule processes the different types of modules entered to the system. Note that the operators declared as constructors of sort `PreUnit` in the signature of Full Maude, given in Appendix A, are declared with two arguments, namely the name of the unit, or its name plus its interface, and the list of declarations of such a unit.

```
crl [module] :
   < O : X@Database | db : DB, input : (F[T, T']),
      output : nil, default : MN, Atts >
   => < O : X@Database |
         db : processUnit(F[T, T'], DB), input : nilTermList,
         output : ('\n 'Introduced 'module: modNameToQid(parseModName(T))),
         default : parseModName(T), Atts >
      if (F == 'fmod_is_endfm) or (F == 'mod_is_endm)
           or (F == 'omod_is_endom) .
```

Notice 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 of the module `T` can be complex—a parameterized module—some extra parsing has to be performed by the auxiliary function `parseModName`. Similar rules are given for the processing of theories and views.

```
crl [theory] :
   < O : X@Database | db : DB, input : (F[T, T']),
      output : nil, default : MN, Atts >
   => < O : X@Database |
         db : processUnit(F[T, T'], DB), input : nilTermList,
         output : ('\n 'Introduced 'theory: modNameToQid(parseModName(T))),
         default : parseModName(T), Atts >
      if (F == 'fth_is_endfth) or (F == 'th_is_endth)
           or (F == 'oth_is_endoth) .
```

```
  rl [view] :
     < O : X@Database | db : DB,
        input : ('view_from_to_is_endv['viewToken[T], T', T'', T''']),
        output : nil, default : MN, Atts >
     => < O : X@Database |
           db : processView(
                  'view_from_to_is_endv['viewToken[T], T', T'', T'''], DB),
           input : nilTermList,
           output : ('\n 'Introduced 'view: downQid(T)),
           default : MN, Atts > .
```

Commands are handled by rules as well. For example, the `down`, `reduce`, and `rewrite` commands are handled by the following rules.

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

  crl [red/rew] :
     < O : X@Database | db : DB, input : (F[TL]),
        output : QIL, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : processCommand(F[TL], MN, DB),
           default : MN, Atts >
        if (F == 'red_.) or (F == 'reduce_.)
            or (F == 'red-in_:_.) or (F == 'reduce-in_:_.)
            or (F == 'rew_.) or (F == 'rewrite_.)
            or (F == 'rew`[_`]_.) or (F == 'rewrite`[_`]_.)
            or (F == 'rew-in_:_.) or (F == 'rewrite-in_:_.)
            or (F == 'rew-in`[_`]_:_.)
            or (F == 'rewrite-in`[_`]_:_.) .
```

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 : X@Database | db : DB,
        input : ({'show`module`.}'PreCommand),
        output : nil, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : meta-pretty-print(getFlatUnit(MN, DB),
                      getTopUnit(MN, DB)),
           default : MN, Atts >  .
  rl [show-module-2] :
     < O : X@Database | db : DB, input : ('show`module_.[T]),
        output : nil, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : meta-pretty-print(getFlatUnit(parseModExp(T), DB),
                      getTopUnit(parseModExp(T), DB)),
           default : MN, Atts > .

   ...

endm
```

10.2 The Full Maude Module

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.

```
mod FULL-MAUDE is
  pr META-FULL-MAUDE-SIGN .
  pr DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .
```

The state of the persistent system, which is supported by the built-in module LOOP-MODE, described in Section 2.3.6, is represented as a single object.

```
  subsort Object < State .

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

  var Atts : AttributeSet .
  var X@Database : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .
```

The rule specifying the initial value of the loop is given below. We initialize the database with the predefined modules introduced in Section 6.1.

```
  rl [init] :
     init
     => [nil,
         < o : Database |
            db : evalUnit(CONFIGURATION,
                   evalUnit(TRIV,
                     evalUnit(UP, emptyDatabase))),
            input : nilTermList, output : nil, default : nullModName >,
         nil] .
```

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 Section 3 and Appendix A). This signature is used by the `meta-parse` function to parse the input. If the input is syntactically valid[11], 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 : X@Database | db : DB, input : nilTermList,
```

[11] Of course, the input may be syntactically valid, but not semantically valid, since further processing—for example, of bubbles—may reveal a semantic inconsistency.

```
      output : nil, default : MN, Atts >,
    QIL']
   => if meta-parse(GRAMMAR, QIL) == error*
      then [nil,
            < O : X@Database | db : DB, input : nilTermList,
               output : ('ERROR: 'incorrect 'input '.),
               default : MN, Atts >,
            QIL']
      else [nil,
            < O : X@Database | db : DB,
               input : meta-parse(GRAMMAR, QIL),
               output : nil, default : MN, Atts >,
            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 : X@Database | db : DB, input : TL,
         output : QIL', default : MN, Atts >,
      QIL'']
    => [QIL,
        < O : X@Database | db : DB, input : TL,
           output : nil, default : MN, Atts >,
        (QIL' QIL'')]
       if QIL' =/= nil .

endm
```

11 Extending Full Maude

We illustrate in this section the way in which the module algebra of Full Maude can be extended by defining new module expressions and integrating them with the rest of the system. As an example of module-combining operations we present the classical sum, or union, of module expressions; and as an example showing the new possibilities for defining module operations we present a module expression such that, given any nonzero natural number, it generates a parameterized module specifying a tuple of the corresponding size. The evaluation of these new module expressions follows the general scheme for the evaluation of module expressions discussed in Section 7.5, that is, a new module with such module expression as name must be created, giving to the expression the intended semantics.

The syntax used for the union of module expressions is

```
  op _+_ : ModExp ModExp -> ModExp [assoc prec 42] .
```

Its evaluation consists in generating a unit importing the two module expressions given as arguments [60, 32].

The syntax used for the n-tuple module expression is as follows:

```
  op TUPLE(_) : Token -> ModExp .
```

Its evaluation consists in the generation of a parameterized functional module with the number of TRIV parameters specified by the argument. A sort for tuples of such size, and the corresponding constructor and selector operators, are also defined. Note that the TRIV theory is predefined in Full Maude (see Sections 10.2 and 6.1). For example, the module expression TUPLE(3) produces the following module.

```
fmod TUPLE(3)[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV] is
   sorts 3Tuple .
   op (_,_,_) : Elt.C1 Elt.C2 Elt.C3 -> 3Tuple .
   op p1_ : 3Tuple -> Elt.C1 .
   op p2_ : 3Tuple -> Elt.C2 .
   op p3_ : 3Tuple -> Elt.C3 .
   var E1 : Elt.C1 .
   var E2 : Elt.C2 .
   var E3 : Elt.C3 .
   eq p1(E1, E2, E3) = E1 .
   eq p2(E1, E2, E3) = E2 .
   eq p3(E1, E2, E3) = E3 .
endfm
```

11.1 Extending the Signature

As mentioned in Section 3, in order to add any new module expressions the first thing we must do is to extend the signature of Full Maude. In the following module EXT-FULL-MAUDE-SIGN we extend the module FULL-MAUDE-SIGN, given in Appendix A, which contains the declarations of the signature of Full Maude, with the declarations for the syntax of the new module expressions.

```
fmod EXT-FULL-MAUDE-SIGN is
  including FULL-MAUDE-SIGN .

  op TUPLE(_) : Token -> ModExp .
  op _+_ : ModExp ModExp -> ModExp [assoc prec 42] .

endfm
```

In Section 3 we saw how a constant GRAMMAR of sort FModule was declared in the META-FULL-MAUDE-SIGN module. Such a constant was then identified with the metarepresentation of a module GRAMMAR, in which there was a declaration importing the FULL-MAUDE-SIGN module and declarations for the constructors of the bubble sorts. To extend the metarepresentation of the module to be used in the parsing process, we define a new constant EXT-GRAMMAR, of sort FModule as well, which extends the module GRAMMAR. We can do this using the addImportList function presented in Section 4.5 as in the following module.

```
fmod META-EXT-FULL-MAUDE-SIGN is
  inc META-LEVEL .
  pr META-FULL-MAUDE-SIGN .
  pr UNIT .

  op EXT-GRAMMAR : -> FModule .

  eq EXT-GRAMMAR
    = addImportList((including 'EXT-FULL-MAUDE-SIGN .), GRAMMAR) .

endfm
```

Now the constant EXT-GRAMMAR is identified with the metarepresentation of a module which contains the constructors of the bubble sorts and importation declarations for both FULL-MAUDE-SIGN and EXT-FULL-MAUDE-SIGN. Since EXT-FULL-MAUDE-SIGN imports FULL-MAUDE-SIGN, we get the desired result.

11.2 The Union Module Expression

As we explained in Sections 7.9 and 7.10 for the cases of the instantiation and the renaming module expressions, respectively, the declaration of any new kind of module expression must come together with the definition of the functions `evalModExp`, `modNameToQidList`, and `setUpModExpDependencies` on the new module operator. As discussed in Sections 7.9 and 9.1.1, equations for the `prepareModName` and `parseModExp` functions have to be given as well.

```
fmod UNION-EXPR is
  inc MOD-EXPR .
  pr EVALUATION .
  pr UNIT-DECL-PARSING .
```

First of all, we declare the new module operator of sort ModExp as follows.

```
  op _+_ : ModExp ModExp -> ModExp [assoc comm] .

  var PL : ParameterList .
  var DB : Database .
  vars T T' : Term .
  var EIL : EImportList .
  var QIVEPS : QidViewExpPairSet .
  vars ME ME' : ModExp .
  vars U U' U'' U''' : Unit .
  vars M M' M'' M''' : Module .
  vars MNS MNS' MNS'' : ModNameSet .
  vars VES VES' VES'' : ViewExpSet .
  var IS : InfoSet .
  var QIL : QidList .
```

As mentioned above, the evaluation of a union module expression consists in the creation of a new unit, with such a module expression as name, which imports the two module expressions being united. Note, however, that the unit being created has to be of the right type. The new unit will be generated having one type or another, depending on the types of the arguments of the union module expression.

The function `rightEmptyUnit` generates an empty unit of the lowest of the sorts of its two arguments. In case of having a nonstructured module as argument, the corresponding structured one is considered. If one of the two module expressions corresponds to a theory, then a theory is generated, and the lowest sort is taken between the sort of such a theory and the Unit sort immediately above the sort of the other unit; that is, sorts FUnit, SUnit, or OUnit are considered to do the comparison.

```
  ceq evalModExp(ME + ME', PL, DB)
    = evalUnit(
        addImportList((including ME .) (including ME' .),
          setName(
            rightEmptyUnit(
               getTopUnit(ME, evalModExp(ME, PL, evalModExp(ME', PL, DB))),
               getTopUnit(ME, evalModExp(ME, PL, evalModExp(ME', PL, DB)))),
```

```
        ME + ME')),
      evalModExp(ME, PL, evalModExp(ME', PL, DB)))
    if not unitInDatabase(ME + ME', DB) .

op rightEmptyUnit : Unit Unit -> Unit [comm] .

ceq rightEmptyUnit(U, U')
  = emptyStrFModule
    if U : StrFModule and U' : StrFModule .
ceq rightEmptyUnit(U, U')
  = emptyStrFTheory
    if U : StrFTheory and U' : FUnit .
ceq rightEmptyUnit(U, U')
  = emptyStrSModule
    if U : StrSModule and U' : StrSModule
       and not (U : StrFModule or U' : StrFModule) .
ceq rightEmptyUnit(U, U')
  = emptyStrSTheory
    if U : StrSTheory and U' : SUnit
       and not (U : StrFTheory or U' : StrFTheory) .
ceq rightEmptyUnit(U, U')
  = emptyStrOModule
    if U : StrOModule and U' : StrOModule
       and not (U : StrSModule or U' : StrSModule) .
ceq rightEmptyUnit(U, U')
  = emptyStrOTheory
    if U : StrOTheory and U' : OUnit
       and not (U : StrSTheory or U' : StrSTheory) .
eq rightEmptyUnit(unitError(QIL), U) = unitError(QIL) .
eq rightEmptyUnit(U, unitError(QIL)) = unitError(QIL) .
```

As pointed out in Section 4.4.2, for each new module expression operator being introduced, we need to add equations for the `modNameToQid` function. For the union module expression we only need the following equation:

```
eq modNameToQidList(ME + ME')
  = (modNameToQidList(ME) '+ modNameToQidList(ME')) .
```

Similarly, we need to give equations for the `setUpModExpDependencies` function, presented in Section 6.2, on the union module expression. Given a module expression of the form `ME + ME'` such that `ME` and `ME'` are in the database, we just need to add `ME + ME'` to the set of names of the modules depending on `ME` and `ME'`. In this way, if `ME` or `ME'` are redefined or removed from the database, `ME + ME'` will be removed as well.

```
eq setUpModExpDependencies((ME + ME'),
     database(
       (< ME ; U ; U' ; M ; M' ; MNS ; VES >
        < ME' ; U'' ; U''' ; M'' ; M''' ; MNS' ; VES' > IS),
       MNS'', VES'', QIL))
  = database(
      (< ME ; U ; U' ; M ; M' ; MNS & (ME + ME') ; VES >
       < ME' ; U'' ; U''' ; M'' ; M''' ; MNS' & (ME + ME') ; VES' > IS),
      MNS'', VES'', QIL) .
ceq setUpModExpDependencies((ME + ME'), DB)
  = warning(DB,
      ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
```

```
    if not unitInDatabase(ME, DB) .
  ceq setUpModExpDependencies((ME + ME'), DB)
    = warning(DB,
        ('\n 'ERROR: 'module modNameToQidList(ME') 'not 'in 'database '.))
      if not unitInDatabase(ME', DB) .
```

The prepareModName function on a union module expression makes recursive calls with each of the module expressions given as arguments.

```
  eq prepareModName(ME + ME', QIVEPS)
    = prepareModName(ME, QIVEPS) + prepareModName(ME', QIVEPS) .
```

Finally, the equation for the parseModExp function is as follows:

```
  eq parseModExp('_+_[T, T'])
    = parseModExp(T) + parseModExp(T') .

endfm
```

11.3 The n-tuple Module Expression

Even though the n-tuple module expression is in principle of a completely different nature, the way of handling it is the same as the way of handling any other module expression. Its evaluation produces a new unit, a parameterized functional module in this case, with the module expression as name. New equations defining the semantics of functions evalModExp, modNameToQidList, setUpModExpDependencies, prepareModName, and parseModExp are given for this module expression.

```
fmod N-TUPLE-EXPR is
  inc MOD-EXPR .
  pr EVALUATION .
  pr UNIT-DECL-PARSING .
```

We assume that the parameter given to the expression is a nonzero natural number.

```
  sort NzNat .
  subsort NzNat < NzMachineInt .
  var I : NzMachineInt .
  cmb I : NzNat if I > 0 .
```

Then, the declaration of the operator for the n-tuple module expression is as follows:

```
  op TUPLE(_) : NzNat -> ModExp .

  vars N N' : NzNat .
  var PL : ParameterList .
  var DB : Database .
  var T : Term .
  var EIL : EImportList .
  var QIVEPS : QidViewExpPairSet .
```

The equation for the evalModExp is reduced to the creation of a module as indicated above. Some auxiliary functions are defined in order to generate the different declarations in the module.

```
op tupleParameterList : NzNat -> ParameterList .
op tupleImportList : NzNat -> ImportList .
op createCopyParameters : NzNat Database -> Database .
op tupleOpDeclSet : NzNat -> OpDeclSet .
op tupleOpDeclSetConstructor : NzNat -> OpDecl .
op tupleOpDeclSetConstructorName : NzNat -> Qid .
op tupleOpDeclSetConstructorArity : NzNat -> QidList .
op tupleOpDeclSetSelectors : NzNat NzNat -> OpDeclSet .
op tupleVarDeclSet : NzNat -> VarDeclSet .
op tupleEquationSet : NzNat -> EquationSet .
op tupleEquationSetAux : NzNat Term -> EquationSet .
op tupleTermArgs : NzNat -> TermList .

ceq evalModExp(TUPLE(N), PL, DB)
  = evalUnit(
       fmod TUPLE(N) is
          tupleParameterList(N)
          tupleImportList(N)
          sorts conc(index(', N), 'Tuple) .
          none
          tupleOpDeclSet(N)
          tupleVarDeclSet(N)
          none
          tupleEquationSet(N)
       endfm, createCopyParameters(N, DB))
    if not unitInDatabase(TUPLE(N), DB) .

eq createCopyParameters(N, DB)
  = if N == 1
    then createCopy('TRIV, (par index('C, N) :: 'TRIV), DB)
    else createCopyParameters(_-_(N, 1),
           createCopy('TRIV, (par index('C, N) :: 'TRIV), DB))
    fi .

eq tupleParameterList(N)
  = if N == 1
    then (par index('C, N) :: 'TRIV)
    else parameterList(tupleParameterList(_-_(N, 1)),
           (par index('C, N) :: 'TRIV))
    fi .

eq tupleImportList(N)
  = if N == 1
    then (including par index('C, N) :: 'TRIV .)
    else (tupleImportList(_-_(N, 1))
          (including par index('C, N) :: 'TRIV .))
    fi .

eq tupleOpDeclSet(N)
  = (tupleOpDeclSetConstructor(N) tupleOpDeclSetSelectors(N, N)) .

eq tupleOpDeclSetConstructor(N)
  = (op conc('`(, conc(tupleOpDeclSetConstructorName(N), '`))) :
          tupleOpDeclSetConstructorArity(N) -> conc(index(', N), 'Tuple)
          [none] .) .

eq tupleOpDeclSetConstructorName(N)
```

```
  = if N == 1
    then '_
    else conc('_`,, tupleOpDeclSetConstructorName(_-_(N, 1)))
    fi .

eq tupleOpDeclSetConstructorArity(N)
  = if N == 1
    then index('Elt.C, N)
    else tupleOpDeclSetConstructorArity(_-_(N, 1)) index('Elt.C, N)
    fi .

eq tupleOpDeclSetSelectors(N, N')
  = if N == 1
    then (op conc(index('p, N), '_) :
               conc(index(', N'), 'Tuple) -> index('Elt.C, N)
               [none] .)
    else (tupleOpDeclSetSelectors(_-_(N, 1), N')
          (op conc(index('p, N), '_) :
                conc(index(', N'), 'Tuple) -> index('Elt.C, N)
                [none] .))
    fi .

eq tupleVarDeclSet(N)
  = if N == 1
    then (var index('E, N) : index('Elt.C, N) .)
    else (tupleVarDeclSet(_-_(N, 1))
          (var index('E, N) : index('Elt.C, N) .))
    fi .

eq tupleEquationSet(N)
  = tupleEquationSetAux(N,
      (conc('`(, conc(tupleOpDeclSetConstructorName(N), '`)))
         [tupleTermArgs(N)])) .

eq tupleTermArgs(N)
  = if N == 1
    then index('E, 1)
    else (tupleTermArgs(_-_(N, 1)), index('E, N))
    fi .

eq tupleEquationSetAux(N, T)
  = if N == 1
    then (eq conc(index('p, N), '_)[T] = index('E, N) .)
    else (tupleEquationSetAux(_-_(N, 1), T)
          (eq conc(index('p, N), '_)[T] = index('E, N) .))
    fi .
```

The equations for the modNameToQidList, parseModExp, prepareModName, and setUpModExpDependencies functions on the n-tuple module expression are as follows:

```
eq modNameToQidList(TUPLE(N))
  = ('TUPLE '`( index(', N) '`)) .

eq parseModExp('TUPLE`(_`)['token[T]])
  = TUPLE(parseMachineInt(T)) .

eq prepareModName(TUPLE(N), QIVEPS) = TUPLE(N) .
```

```
  eq setUpModExpDependencies(TUPLE(N), DB) = DB .

endfm
```

11.4 Extended Full Maude

To integrate the specification of Full Maude with the specification dealing with these two new module expressions, we must redefine the module `FULL-MAUDE` given in Section 10.2. For the present purposes, we only need to redefine the `in` rule of that module. In general we may need to change any of the rules in it. Further possibilities of extending Full Maude are discussed in [26, Chapter 5], adding new predefined modules in the initial database, defining new commands, or adding new attributes to keep some extra state in the persistent object. In the present case, however, we only need to change the module in which the inputs are parsed. This is done by changing the `in` rule such that the constant `EXT-GRAMMAR` is used instead of `GRAMMAR`. Notice that we also need to import the modules `UNION-EXPR` and `N-TUPLE-EXPR`. To facilitate incremental extensions of the system, we define a new module `EXT-DATABASE-HANDLING` importing these modules, and then another module `EXT-FULL-MAUDE` importing it. `EXT-FULL-MAUDE` also imports `PREDEF-UNITS`, `LOOP-MODE`, and `META-EXT-FULL-MAUDE-SIGN`.

```
mod EXT-DATABASE-HANDLING is
  pr DATABASE-HANDLING .
  pr UNION-EXPR .
  pr N-TUPLE-EXPR .
endm

mod EXT-FULL-MAUDE is
  pr META-EXT-FULL-MAUDE-SIGN .
  pr EXT-DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .

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

  var Atts : AttributeSet .
  var X@Database : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .
```

We only give here the new `in` rule.

```
  crl [in] :
     [QIL,
      < O : X@Database | db : DB, input : nilTermList,
         output : nil, default : MN, Atts >,
      QIL']
     => if meta-parse(EXT-GRAMMAR, QIL) == error*
        then [nil,
```

```
          < O : X@Database | db : DB, input : nilTermList,
              output : ('ERROR: 'incorrect 'input '.),
              default : MN, Atts >,
            QIL']
      else [nil,
            < O : X@Database | db : DB,
              input : meta-parse(EXT-GRAMMAR, QIL),
              output : nil, default : MN, Atts >,
            QIL']
      fi
      if QIL =/= nil .

  ...

endm
```

12 Concluding Remarks

We have seen in the previous sections how, by using the metaprogramming capabilities of Maude, we have been able to define a Core Maude executable specification of Full Maude. With this specification we have endowed Full Maude with a module algebra in the Clear/OBJ tradition, including module hierarchies, parameterized modules, views, and module expressions. In addition, new module-combining and module-transforming operations have likewise been provided. We have seen, for example, how we have integrated the transformation of object-oriented modules into their corresponding system modules with the module algebra, and how new kinds of "parameterized" modules, such as parameterized extensions of `META-LEVEL` and the `TUPLE(`n`)` construction can be easily added.

The module algebra thus obtained becomes easily *extensible*, in the sense that a user may easily define new module transformations and module operations; that is, the user can not only do metaprogramming by using the module operations provided in the module algebra of the language, but can also do *meta-metaprogramming* by defining new ones. We have illustrated some of the possibilities for extending the module algebra by defining some new module expressions. Furthermore, given the reflective design of Maude, it is even possible to change both the module operations provided in the basic syntax of the language and that very syntax itself.

The Full Maude specification described here is part of an overall reflective and extensible design for the Maude system, in which key components of the system are written in Maude itself and can be modified or extended with relative ease. In particular, we plan a number of such extensions for the module algebra, including the addition of parameterized theories and views, and the development of other module composition and module transformation operations. Module operations to hide sorts and operators, and operations to redefine and remove messages in object-oriented modules are among the most interesting operations we plan to consider in extensions of our module algebra.

Furthermore, we plan to generalize the module algebra that we have developed for Maude into a *module algebra generator*, that could endow many other specification languages with powerful and extensible algebras for combining and transforming specifications. Module algebra generator uses can be supported by defining a *parameterized* algebraic data type, say $ModAlg[X]$, that, given a logic $\mathcal{L}$ having "good" metalogical properties, extends the data type $\mathtt{Module}_{\mathcal{L}}$ of $\mathcal{L}$-theories to an algebra of theory-composition operations $ModAlg[Module_{\mathcal{L}}]$. The module algebra of Maude provided by the Full Maude specification should then be regarded as a particular instance of such a generic construction, namely, for the case in which the underlying logic $\mathcal{L}$ is rewriting logic.

Another research direction opened up by this approach that we plan to pursue in the future is formal reasoning about the properties of module algebra operations based upon their executable specifications.

References

[1] Jan Bergstra, Jan Heering, and Paul Klint. *Algebraic Specification.* ACM Press, 1989.

[2] 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, *Seventh Colloquium on Automata, Languages and Programming*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[3] Peter Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proc. SOFTSEM'95*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer, 1995.

[4] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[5] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. To appear in *Theoretical Computer Science* Available at `http://maude.csl.sri.com`.

[6] 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`.

[7] 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.

[8] Roberto Bruni, José Meseguer, and Ugo Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, July 1998.

[9] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjørner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, 1980.

[10] Carlos Castro. An approach to solving binary CSP using computational systems. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[11] Manuel Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD thesis, University of Navarre, 1998.

[12] Manuel Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.

[13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [35]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[14] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewriting logic. SRI International, January 1999. Available at `http://maude.csl.sri.com`.

[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude as a metalanguage. In Kirchner and Kirchner [35]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[17] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.

[18] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Manuscript, SRI International, February 1998.

[19] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Maude as a formal meta-tool. Submitted for publication, February 1999.

[20] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[21] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[22] Manuel Clavel and José Meseguer. Internal strategies in a reflective logic. In B. Gramlich and Hèléne Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.

[23] CoFI Task Group on Language Design. CASL—The common algebraic specification language, version 1.0. Available at `http://www.brics.dk/Projects/CoFI`, October 1998.

[24] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.

[25] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet, Gordon Plotkin, and C. Jones, editors, Proceedings *of Workshop on Logical Frameworks (Edinburgh, United Kingdom, May 1991)*, pages 83–130. Cambridge University Press, 1991.

[26] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[27] Francisco Durán and José Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [35]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[28] Steven Eker. Term rewriting with operator evaluation strategy. In Kirchner and Kirchner [35]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[29] Kokichi Futatsugi and Toshimi Sawada. Cafe as an extensible specification environment. In *Proceeding of the Kunming International CASE Symposium, Kunming, China*, November 1994.

[30] Joseph Goguen and Rod Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Proc. Logics of Programming Workshop*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1984.

[31] 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.

[32] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992. To appear in Joseph Goguen and Grant Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press.

[33] 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.

[34] Jan Heering, Paul Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF. reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Revised version of December 1992 available at `http://www.cwi.nl/~gipe/`.

[35] Claude Kirchner and Hélène Kirchner, editors. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, Nancy, France*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[36] Claude Kirchner, Hélène Kirchner, and Marian 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.

[37] Hélène Kirchner and Pierre-Etienne Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.

[38] Paul Klint, editor. *The ASF+SDF Meta-Environment User's Guide*. Centrum voor Wiskunde en Informatica (CWI), 1995. Available at `http://www.cwi.nl/~gipe/`.

[39] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, August 1993. To appear in D.M. Gabbay, editor, *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[40] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[41] 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`.

[42] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[43] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[44] 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 Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[45] José Meseguer, editor. *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications, Asilomar, California, U.S.A*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[46] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Ugo Montanari and Vladimiro Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory, CONCUR'96, Pisa, Italy, August 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer-Verlag, 1996.

[47] 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.

[48] José Meseguer. Membership algebra as a semantic framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[49] 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.

[50] 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.

[51] José Meseguer and Carolyn 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. Available at `http://www-formal.stanford.edu/clt/ArpaNsf/adl-interop.html`.

[52] Peter Mosses. *Action Semantics*. Cambridge University Press, 1992.

[53] 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*, pages 157–176. Chapman & Hall, 1997.

[54] 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.

[55] José Quesada. Bidirectional and event-driven parsing with multi-virtual trees. In C. Martin-Vide, editor, *Mathematical and Computational Models in Linguistics*. John Benjamins, 1996.

[56] José Quesada. The Maude parser: Parsing and meta-parsing β-extended Technical Report, SRI International, Computer Science Laboratory, 1999.

[57] Christophe Ringeissen. Prototyping combination of unification algorithms with the ELAN rule-based programming language. In Hubert Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*. Springer, 1997.

[58] Carolyn Talcott. An actor rewrite theory. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[59] Arie van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[60] Timothy Winkler. A short note on the OBJ3 module expression routines. SRI International, March 1991.

[61] Martin Wirsing and Alexander Knapp. A formal approach to object-oriented software engineering. In Meseguer [45]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

A 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. 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 O-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 .

  *** message declaration
  op msg_:_->_. : Token SortList Sort -> MsgDecl .
  op msgs_:_->_. : NeTokenList SortList Sort -> MsgDecl .
endfm

fmod MOD-EXPRS is
  including O-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 O-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
```