Electronic Notes in Theoretical Computer Science 15 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume15.html 22 pages

An Extensible Module Algebra For Maude

Francisco Durán and José Meseguer

Computer Science Laboratory, SRI International
Menlo Park, CA 94025, USA
{duran,meseguer}@csl.sri.com

Abstract

The fact that rewriting logic and Maude are reflective, so that rewriting logic specifications can be manipulated as terms at the metalevel, opens up the possibility of defining an algebra of module composition and transformation operations within the logic. This makes such a module algebra easily modifiable and extensible, enables the implementation of language extensions within Maude, and allows formal reasoning about the module operations themselves. In this paper we discuss in detail the Maude implementation of a specific choice of operations for a module algebra of this type, supporting module operations in the Clear/OBJ tradition as well as the transformation of object-oriented modules into system modules.

1 Introduction

Rewriting logic is reflective [16,9,3] in the precise sense that there is a finitely presented rewrite theory $\mathcal{U}$ that can simulate all other finitely presented rewrite theories, including itself. In particular, such a universal theory can metarepresent all finitely-presentable rewrite theories as terms of a sort `Module`. As explained in [5,8], Maude provides efficient support for reflective computation through its `META-LEVEL` module, that builds in key functionality for the universal theory $\mathcal{U}$. In particular, both functional and system modules are represented as terms of sort `Module`.

As pointed out in [8] and demonstrated in a number of substantial applications such as theorem proving tools and translations between logics [10,3,7], the reflective features of Maude support a novel style of *metaprogramming* with very powerful module-combining and module-transforming operations that surpass those of traditional parameterized programming and can greatly advance software reusability and adaptability.

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

This paper reports on another extended case study on exploiting the metaprogramming features of Maude. The goal in this case is to use reflection in order to define within rewriting logic itself a *module algebra* that can be made available to the user as part of the syntax of the language. Such a module algebra should include the traditional "parameterized programming" module operations in the Clear/OBJ tradition such as module hierarchies, parameterized modules, views, and module expressions, which can now be evaluated to their resulting modules by executing the Maude specifications defining the semantics of such expressions at the metalevel. In addition, new module operations should likewise be provided. One obvious first such operation is the transformation of object-oriented modules—which have a special syntax supporting objects and classes—into their corresponding system modules.

The module algebra thus obtained becomes easily *extensible*, in the sense that a user may easily define new module transformations and module operations at the metalevel; 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-meta-programming* by defining new ones. Furthermore, given the reflective design of Maude [6], it is even possible to change both the module operations provided in the basic syntax of the language and that very syntax itself. In fact, as explained in some detail in [6], both the module operations supported by Maude and the part of the system integrating the different components in the arquitecture, namely the rewrite engine, the parser, and the interface with the user, have been implemented in Maude itself. Therefore, the ideas in the present paper are part of this overall reflective and modifiable design.

We now explain in more detail how the Clear/OBJ operations and the transformation of object-oriented modules are defined.

1.1 Module Hierarchies and Parameterized Programming

As in Clear [2], OBJ [13,17], and other specification languages in that tradition, the abstract syntax for writing specifications in Maude can be seen as given by *module expressions*, where the notion of module expression is understood as an expression that defines a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations. In fact, structuring is essential in all specification languages, not only to facilitate the construction of specifications from already existing ones—with more or less flexible reusability mechanisms—but also for managing the complexity of understanding and analyzing large specifications.

A module importing some combination of modules, given by module expressions, can be seen as a structured module with more or less complex relationships among its component submodules. For execution purposes, however, we typically want to convert this structured module into an equivalent unstructured module, that is, a "flattened" module without submodules. In the case of Maude, this flattened module will then be compiled into the rewrite engine. By systematically using the metaprogramming capabilities of Maude we can both evaluate module expressions into structured module hierarchies, and flatten such hierarchies into unstructured modules for execution by defining all such module operations by rewrite rules that operate on the metalevel term representations of modules.

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. 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.

The key fact systematically exploited by our reflective approach to module composition is that the module `META-LEVEL` [5,4] in Maude's prelude provides an abstract data type `Module` that represents modules and other auxiliary sorts for module items and for auxiliary data. All our module operations are defined as extensions of the data type `Module`; therefore, we refer the reader to [4] for more details on the reflective features of `META-LEVEL` that we use as our basis[2]. For the moment let us mention that it includes constructors `mod_is________endm` and `fmod_is_______endfm` to represent, respectively, system and functional modules with syntax:

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

In the current version of `META-LEVEL` a module has to be flat, that is, the importation list mentioned above always has to be `nil`. Therefore, in the module algebra we have extended the sort `Module` by a supersort `EModule` corresponding to structured modules that can contain other submodules and module expressions, and that can be parameterized. We have also added other operator declarations to support other entities such as theories and

[2] We do not include here the module `META-LEVEL`, although we will use some of the declarations in it. In particular, we will declare many of the sorts we introduce along this paper as supersorts of sorts in `META-LEVEL`. To simplify the reading we will overload those constructors giving the same functionality as those declared in `META-LEVEL` for their subsorts. We follow the convention of naming `nil` the constants for empty lists, and `none` the constants for empty sets.

views needed for module instantiation, so that the reduction of a structured module to normal form includes as a particular case the evaluation of module expressions that may instantiate several parameterized modules, and that may rename certain sorts, operators, and so on.

1.2 Transforming Object-Oriented Modules

The reflective approach followed in Maude allows us not only to modify or add new module-combining operators, but also opens up the possibility of defining a wide range of other specification transformations. The reflective access to a data type of module representations makes it possible to consider any computable module transformation that, thanks to the general metaresult of Bergstra and Tucker [1], can be specified in Maude by a finite number of Church-Rosser and terminating equations.

An important and very general use of the module transformation idea—intimately connected with the logical and semantic framework applications of rewriting logic [14]—is defining and executing *within* rewriting logic itself any effective *representation map* of the form

$$\Phi : \mathcal{L} \longrightarrow \mathcal{L}'$$

for languages or logics $\mathcal{L}$ and $\mathcal{L}'$. The idea is that we can extend the universal theory $\mathcal{U}$ with the definitions of data types $\texttt{Module}_{\mathcal{L}}$ and $\texttt{Module}_{\mathcal{L}'}$ reifying, respectively, the modules in $\mathcal{L}$ and $\mathcal{L}'$, and we can then reify the function Φ as an equationally-defined function $\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}_{\mathcal{L}'}$ mapping terms in $\texttt{Module}_{\mathcal{L}}$ to terms in $\texttt{Module}_{\mathcal{L}'}$. As a particular case we consider the use of reflection to reify the passage from object-oriented specifications to rewriting logic specifications within rewriting logic itself. Object-oriented specifications can be viewed as a language $\mathcal{O}$, and the passage to their corresponding rewriting logic specifications can be understood as a mapping

$$\Phi : \mathcal{O} \longrightarrow \mathit{RWLogic}.$$

We can reify such object-oriented modules as an algebraic data type $\texttt{Module}_{\mathcal{O}}$—we call it `OModule` in its Maude specification—and the mapping Φ as an equationally-defined function

$$\overline{\Phi} : \texttt{OModule} \longrightarrow \texttt{Module}.$$

This map has been defined in Maude to transform object-oriented modules into system modules [3].

The rest of the paper follows the outline of this introduction. We first discuss module hierarchies, then parameterized modules, and finally the trans-

[3] We only discuss here the flat case. However, in the actual implementation of the system the map is from structured object-oriented modules to structured system modules.

formation of object-oriented modules. We conclude with some remarks about future developments.

2 Module Hierarchies

Our aim is to present the main ideas about the way in which reflection can be used to implement new module operations. Thus, in this section we will restrict ourselves to quite simple and well-known module operations: importations—in *protecting* and *including* modes—and renamings.

We describe in Section 2.1 how we have extended the module `META-LEVEL` with declarations to support more complicated forms of modules—as parameterized modules—, module combining and transforming operations—as importations and renamings—, and also other entities as theories and views.

As already mentioned, the general principle in our design consists in first evaluating any module expression reducing it to a canonical form in which only module inclusions appear, that is, to a module hierarchy, which can be seen as a partial order. The development of the system has been accomplished upon the principle of evaluating all module expressions on structured modules returning irreducible structured modules, and only using the flat version of the modules for module execution purposes. We have then two different processes clearly differentiated: a first step in which the structured module is evaluated and reduced to its normal form—explained in Section 2.2—, and a second step in which this normal form is flattened—discussed in Section 2.3. We illustrate both stages in this process with a very simple example, namely the importation of a renamed module.

2.1 Extended Modules

We extend the module `META-LEVEL` with a new sort `EModule`, for structured modules, as a supersort of `Module`, with constructors `mod_is_______endm` and `fmod_is______endfm`, allowing the importation of module expressions, that describe structured modules, and keep the sort `Module` for flattened modules. We also extend the syntax of these modules to allow module expressions as module names, instead of just quoted identifiers, and allow parameterized modules by adding an argument for their parameters. For unparameterized modules this list will be `nil`. We define a sort `ModExp`, ranging over module expressions, as a supersort of `Qid`, and then define the intended module operators. We further extend the signature of modules in `META-LEVEL` to allow more complex forms for the sorts. We will explain the need for this extension in Section 3. We define a sort `ESort` as supersort of `Sort`, and the corresponding declarations to allow this kind of sorts in declarations of operators, variables, etc. The following syntax extends to the supersort `EModule` a number of operations for the sort `Module` in `META-LEVEL` (Cf. Section 3.2 in [4]) and adds the new ones mentioned above.

```
sorts EModule ModExp EImport EImportList Parameter ParameterList
      ESort ESortSet ESortList ESortDecl ESubsortDecl
      ESubsortDeclSet EOpDecl EOpDeclSet EVarDecl EVarDeclSet
      EMembAx EMembAxSet .
subsorts Qid < ModExp .
subsorts ImportList EImport < EImportList .
subsort Module < EModule .
subsort Parameter < ParameterList .
subsort Sort < ESort .
subsorts QidSet ESort < ESortSet .
subsort SortDecl < ESortDecl .
subsort SubsortDecl < ESubsortDecl .
subsorts SubsortDeclSet ESubsortDecl < ESubsortDeclSet .
subsort OpDecl < EOpDecl .
subsorts OpDeclSet EOpDecl < EOpDeclSet .
subsort VarDecl < EVarDecl .
subsorts VarDeclSet EVarDecl < EVarDeclSet .
subsort MembAx < EMembAx .
subsorts EMembAx MembAxSet < EMembAxSet .

op _;_ : ESortSet ESortSet -> ESortSet [assoc comm id: none] .
op __ : ESortList ESortList -> ESortList [assoc id: nil] .
op sorts_. : ESortSet -> ESortDecl .
op subsort_<_. : ESort ESort -> ESubsortDecl .
op __ : ESubsortDeclSet ESubsortDeclSet -> ESubsortDeclSet
      [comm assoc id: none] .
op op_:_->_[_]. : Qid ESortList ESort AttrSet -> EOpDecl .
op __ : EOpDeclSet EOpDeclSet -> EOpDeclSet
      [assoc id: none] .
op var_:_. : Qid ESort -> EVarDecl .
op __ : EVarDeclSet EVarDeclSet -> EVarDeclSet
      [assoc comm id: none] .
op mb_:_. : Term ESort -> EMembAx .
op cmb_:_if_=_. : Term ESort Term Term -> EMembAx .
op __ : EMembAxSet EMembAxSet -> EMembAxSet
      [assoc comm id: none] .

op including : ModExp -> EImport .
op protecting : ModExp -> EImport .
op __ : EImportList EImportList -> EImportList
      [assoc id: nil] .

op _::_ : Qid ModExp -> Parameter .
op nil : -> ParameterList .
op _,_ : ParameterList ParameterList -> ParameterList
      [assoc id: nil] .
```

```
op mod_is_________endm : ModExp ParameterList EImportList
      ESortDecl ESubsortDeclSet EOpDeclSet EVarDeclSet EMembAxSet
      EquationSet RuleSet -> EModule .
op fmod_is________endfm : ModExp ParameterList EImportList
      ESortDecl ESubsortDeclSet EOpDeclSet EVarDeclSet EMembAxSet
      EquationSet -> EModule .
```

We present here only two kinds of module expressions, one for renamed modules—of the form M*(MS) for a module name M and a set of maps MS—and another one for instantiations of parameterized modules—of the form M[VE] for M the name of a module and VE a view expression, where a view expression is just a list of names of views[4]. We will introduce views in Section 3.3.

```
sorts ViewExp .
subsorts Qid < ViewExp .
op _*(_) : ModExp MapSet -> ModExp .
op _[_] : ModExp ViewExp -> ModExp .
op _,_ : ViewExp ViewExp -> ViewExp [assoc] .
```

A renaming can be considered as a function that, given a module M and a set of mappings of the form[5] (`sort` S `to` S') or (`op` O `to` O'), returns a copy of M in which the names of the sorts and operations are changed as indicated. In our reflective design, this function is applied to the metarepresentation of the module, and the semantics of the renaming function is defined by a set of equations.

```
sorts OpMap SortMap Map MapSet .
subsorts OpMap SortMap < Map .
subsort Map < MapSet .
op none : -> MapSet .
op _,_ : MapSet MapSet -> MapSet [assoc comm id: none] .
op op_to_ : Qid Qid AttrSet -> OpMap .
op sort_to_ : ESort ESort -> SortMap .
```

2.2 Normalization of Extended Modules

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

[4] Although treated by the implementation, we prefer not to consider here composition of views or any other more complicated form of view.

[5] The system also supports maps of the form (`op` O : CCL `->` CC `to` O'), for renaming of operators of a particular arity, and (`label` L `to` L'), for renaming of the labels of rules. A renaming of operators can also include changes in its attributes.

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. 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 "confussion" associated to the congruence modulo 3 equation has only been introduced in the sort `Nat3`, which has nothing to do with the sort `Nat`.

$$\texttt{NAT} \xrightarrow[\phi]{} \texttt{NAT *(sort Nat to Nat3)} \hookrightarrow \texttt{NAT3}$$

The metarepresentations of the modules `NAT` and `NAT3` before being evaluated are given by the following terms of sorts `Module` and `EModule`, respectively:

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

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

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 the module expressions should take place in the context of an *environment*, or *database* in which we keep information about the modules already introduced in the system, and also about those modules generated internally. For each module we save the original module (in structured form) and its flat form as a pair of sort `ModuleInfo`. Each time a module is introduced in the system, a new entry in the database is created, and each of its slots is filled accordingly.

```
sorts Database ModuleInfo Info .
subsorts ModuleInfo < Info < Database .
op <_,_> : EModule Module -> ModuleInfo .
op none : -> Database .
op __ : Database Database -> Database [assoc comm id: none] .
```

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 saved 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 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. Note that in this case, as in general for most of 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 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
  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.

$$\texttt{NAT *(sort Nat to Nat3)} \hookrightarrow \texttt{NAT3}$$

2.3 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* [2,11,12]. However, instead of considering the category of specifications and specification morphisms [2], 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

$$\texttt{flatten} : \texttt{EModule Database} \to \texttt{Database}$$

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

The resulting flat specification for `NAT3` is:

```
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
```

Note that in the flattening of a structured module we are assuming that sort names are unique—in each structure, not necessarily in the database. This proposal is simpler than the solution taken in OBJ, 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 further discussed in Section 3.1—is a common occurrence in the presence of parameterized modules.

3 Parameterized Modules

We treat parameterized modules [2,13] using the same approach. As in OBJ, a theory defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter. The instantiation of the formal parameters of a parameterized module with actual parameter modules requires a *view* from the formal interface theory to the corresponding actual module. That is, views provide the interpretation of the actual parameters. We focus here on simple parameterizations, namely the case of modules parameterized with one or more unparameterized theories.

3.1 Naming of Sorts in Parameterized Modules

The convention of not qualifying sorts is particularly weak when dealing with parameterized modules. Let us illustrate the problem with an example. In the case of OBJ, importing, for example, sets or lists of different elements introduces repeated sorts `Set` or `List` and operators that must be qualified by the names of the modules they come from, that is, by module expressions of considerable length. Of course, in OBJ it is possible to rename all these items. But this means that we have to include explicitly many more renamings than we would like.

Given that Maude supports ad-hoc overloading and constants can be qualified in order to be disambiguated, the problem is reduced to the collisions of sorts. Our proposal consists in renaming parameterized sorts, not with the module expression they belong to but with the name of the view used in the instantiation of the module. We assume that all views are named, and these names are the ones used in the qualification [6]. To simplify the notation and to make explicit to the user what is going on, in the body of a parameterized module M`[`X`::`T`]`, any sort S is written in the form S`[`X`]`. When the module is instantiated with some view V this sort becomes S`[`V`]`. To complete the declarations needed to handle this kind of names for sorts of sort `ESort` as already introduced in Section 2.1 we only need to add the following constructor.

```
op _[_] : Qid ViewExp -> ESort .
```

3.2 Theories

Let us consider the following parameterized specification for sets, with parameter theory `TRIV`. In this case this just means that the actual parameter must have a sort whose elements can be used as set elements. As in OBJ3, theories have the same structure as modules.

[6] In some cases these named views may need to be composed, giving rise to somewhat longer qualifications. A full treatment of this more general case will be given elsewhere.

```
fth TRIV is
  sort Elem .
endfth

fmod SET[X :: TRIV] is
  sort Set[X] .
  subsort Elem < Set[X] .
  op empty : -> Set[X] .
  op __ : Set[X] Set[X] -> Set[X] [assoc comm id: empty] .
  var S : Set[X] .
  eq S S = S .
endfm
```

We need some declarations to represent theories. We treat here the case of functional theories.

```
sort FTheory .
subsort FTheory < EModule .
op fth_is________endfth : ModExp EImportList ESortDecl
      ESubsortDeclSet EOpDeclSet EVarDeclSet EMembAxSet
      EEquationSet -> FTheory .
```

Given these declarations, the metarepresentations of the functional theory TRIV and of the functional module SET are as follows:

```
fth 'TRIV is
  nil
  sorts 'Elem .
  none
  none
  none
  none
  none
endfth

fmod 'SET is
  nil
  'X :: 'TRIV
  sorts 'Set['X] .
  subsort 'Elem < 'Set['X] .
  op 'empty : nil -> 'Set['X] .
  op '__ : 'Set['X] 'Set['X] -> 'Set['X] [assoc comm id: 'empty] .
  var 'S : 'Set['X] .
  none
  eq '__['S, 'S] = 'S .
endfm
```

The module SET has only one parameter. In general, parameterized modules can have several parameters. It can furthermore happen that several parameters are declared with the same parameter theory, that is, we can have an interface of the form [X :: TH, Y :: TH]. Therefore, the parameters cannot be treated as normal submodules, since we do not want them to be shared. We regard the relationship between the body of a parameterized module and the interface of its parameters not as an inclusion, but as a module constructor which is evaluated generating a renamed copy of the parameters, which are then included. The intuition behind the construction in the previous example can be again inferred from the following picture. We see how a copy of the theory TRIV is generated, with name X :: TRIV and with each sort S and operation O (in this case just the sort Elem) renamed to S.X and O.X, respectively. Note that all occurrences of these sorts and operators in the parameterized module have to be correspondingly renamed.

$$\mathtt{TRIV} \xrightarrow[\phi]{} \mathtt{X :: TRIV} \hookrightarrow \mathtt{SET}$$

3.3 Views and Instantiation

Each view has also its representation at the metalevel as a term, and therefore can be used, together with the corresponding modules, to evaluate the given module expression. The result of such an instantiation replaces each interface theory by its corresponding actual module, using the views to bind actual names to formal names. Then, for example, to generate a specification for sets of natural numbers we can define the following view:

```
view Nat from TRIV to NAT is
  sort Elem to Nat .
endv
```

We need the following declarations:

```
sort View .
op view_from_to_is_endv : Qid ModExp ModExp MapSet -> View .
```

Thus, the metarepresentation of the previous view is

```
view 'Nat from 'TRIV to 'NAT is
  sort 'Elem to 'Nat .
endv
```

Then, the evaluation of the module expression SET[Nat] consists in applying to the module SET the renaming obtained from the underlying renaming in the view Nat. We denote this transformation and its associated map of specifications by $id_{\mathtt{SET}}$[Nat], since it is somehow the combination of the identity morphism for the parameterized module and the corresponding view defining the instantiation. As it is well-known, the instantiation must satisfy the pushout diagram,

$$\begin{tikzcd}
\mathtt{SET} \arrow[r, dotted, "id_{\mathtt{SET}}\mathtt{[Nat]}"] \arrow[dr, phantom, "p.o."] & \mathtt{SET[Nat]} \\
\mathtt{X :: TRIV} \arrow[u, hook] \arrow[r, "\mathtt{Nat}"'] & \mathtt{NAT} \arrow[u, hook, dotted]
\end{tikzcd}$$

In our metalevel representation of modules, this instantiation is performed by a function

$$\mathtt{instantiate} : \mathtt{EModule\ \ View} \to \mathtt{EModule}$$

that takes the metarepresentation of a parameterized module `M[X::T]` and the metarepresentation of a view `V` from the theory `T` to some module and returns the metarepresentation of the corresponding instantiation. The correctness criterion (relative to the pushout semantics) for the `instantiate` function is given by the equation

$$\mathtt{instantiate}(\overline{M[X :: T]},\ \overline{V}) = \overline{M[V]}.$$

4 From Object-Oriented Modules to System Modules

Our exposition assumes familiarity with the logical theory of concurrent objects and the syntax for object-oriented modules discussed in [15]. We briefly recall the most basic ideas, and then present the syntax for the object-oriented modules, its representation at the metalevel, and the translation of object-oriented modules into system modules.

In a concurrent object-oriented system the concurrent state, which is usually called a *configuration*, has typically the structure of a multiset made up of objects and messages. Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent, and then causing "communication events" by application of rewrite rules. An *object* in a given state is represented as a term

$$\texttt{<}\ O : C\ \texttt{|}\ a_1 : v_1\texttt{,}\ \ldots\texttt{,}\ a_n : v_n\ \texttt{>}$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*. Let us consider then the following module `CONFIGURATION` with declarations for these concepts.

```
fmod CONFIGURATION is
  sorts Oid Cid Aid Attribute AttributeSet Value
        Configuration Object Message .
```

```
  subsort Qid < Oid .
  subsort Object Message < Configuration .
  subsort Attribute < AttributeSet .

  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
          [assoc comm id: none] .
  op _:_ : Aid Value -> Attribute .
  op <_:_|_> : Oid Cid AttributeSet -> Object .
  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration
          [assoc comm id: none] .
endfm
```

In Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword `omod`—using a syntax more convenient than that of system modules because it assumes acquaintance with the basic entities, such as objects, messages and configurations, and supports linguistic distinctions appropriate for the object-oriented case.

For example, the `ACCNT` object-oriented module below specifies the concurrent behavior of objects in a very simple class `Accnt` of bank accounts, each having a `bal`(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts.

```
omod ACCNT is
  protecting MACHINE-INT .
  class Accnt | bal: MachineInt .
  msgs credit debit : Oid MachineInt -> Message .
  msg transfer_from_to_ : MachineInt Oid Oid -> Message .
  vars A B : Oid .
  vars M N N' : MachineInt .
  rl [credit] : credit(A, M) < A : Accnt | bal: N >
     => < A : Accnt | bal: N + M > .
  crl [debit] : debit(A, M) < A : Accnt | bal: N >
     => < A : Accnt | bal: N - M >
        if N >= M .
  crl [transfer] : transfer M from A to B
     < A : Accnt | bal: N > < B : Accnt | bal: N' >
     => < A : Accnt | bal: N - M > < B : Accnt | bal: N' + M >
        if N >= M .
endom
```

Classes are defined with the keyword `class`, followed by the name of the class, and by a list of attribute declarations, pairs of the form $a : S$ separated by commas, where a is an attribute identifier and S is the sort inside which the values of such an attribute identifier must range in the given class. In this example, the only attribute of an account is its bal(ance), which is declared to be a value in MachineInt, sort declared in the module MACHINE-INT in Maude's prelude. The three kinds of messages involving accounts are credit, debit, and transfer messages, whose user-definable syntax is introduced by the keyword msg. The rewrite rules in the module specify in a declarative way the behavior associated with the messages.

4.1 Representing Object-Oriented Modules at the Metalevel

We define the abstract data type OModule to represent object-oriented modules and other auxiliary sorts for module items and for auxiliary data. It includes the constructor omod_is___________endom to represent object-oriented modules and the different items that can appear in them with syntax

```
sorts ClassDecl ClassDeclSet SubclassDecl SubclassDeclSet
      MsgDecl MsgDeclSet AttrDecl AttrDeclSet OModule .

subsort ClassDecl < ClassDeclSet .
subsort SubclassDecl < SubclassDeclSet .
subsort MsgDecl < MsgDeclSet .
subsort AttrDecl < AttrDeclSet .
subsort OModule < EModule .

op omod_is____________endom :
      ModExp ParameterList EImportList ESortDecl ESubsortDeclSet
      ClassDeclSet SubclassDeclSet EOpDeclSet MsgDeclSet
      EVarDeclSet MembAxSet EquationSet RuleSet -> OModule .

op _:_ : Qid ESort -> AttrDecl .
op none : -> AttrDeclSet .
op __ : AttrDeclSet AttrDeclSet -> AttrDeclSet
     [assoc comm id: none] .

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

op subclass : ESort ESort -> SubclassDecl .
op none : -> SubclassDeclSet .
op __ : SubclassDeclSet SubclassDeclSet -> SubclassDeclSet
     [assoc comm id: none] .

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

Note that object-oriented modules can also be parameterized and, as in the case of sorts, class names can be qualified by view names. The representation $\overline{\texttt{ACCNT}}$ of ACCNT at the metalevel is the term

```
omod 'ACCNT is
  protecting 'MACHINE-INT .
  sorts none .
  none
  class 'Accnt | 'bal : 'MachineInt .
  none
  none
  msg 'credit : 'Oid 'MachineInt -> 'Message .
  msg 'debit : 'Oid 'MachineInt -> 'Message .
  msg 'transfer_from_to_ : 'MachineInt 'Oid 'Oid -> 'Message .
  var 'A : 'Oid .
  var 'B : 'Oid .
  var 'M : 'MachineInt .
  var 'N : 'MachineInt .
  var 'N' : 'MachineInt.
  none
  none
  rl ['credit] :
     '__['credit['A, 'M],
         '<_:_|_>['A, {'Accnt}'Accnt@Class,
                  '_:_[{'bal}'Aid, 'N]]]
     => '<_:_|_>['A, {'Accnt}'Accnt@Class,
                 '_:_[{'bal}'Aid, '_+_['N, 'M]]] .
  crl ['debit] :
     '__['debit['A, 'M],
         '<_:_|_>['A, {'Accnt}'Accnt@Class,
                  '_:_[{'bal}'Aid, 'N]]]
     => '<_:_|_>['A, {'Accnt}'Accnt@Class,
                 '_:_[{'bal}'Aid, '_-_['N, 'M]]]
        if '_>_['N, 'M] = {'true}'Bool .
  crl ['transfer] :
     '__['transfer_from_to_['M, 'A, 'B],
         '<_:_|_>['A, {'Accnt}'Accnt@Class,
                  '_:_[{'bal}'Aid, 'N]],
         '<_:_|_>['B, {'Accnt}'Accnt@Class,
                  '_:_[{'bal}'Aid, 'N']]]
     => '__['<_:_|_>['A, {'Accnt}'Accnt@Class,
                     '_:_[{'bal}'Aid, '_-_['N, 'M]]],
            '<_:_|_>['B, {'Accnt}'Accnt@Class,
                     '_:_[{'bal}'Aid, '_+_['N', 'M]]]]
        if '_>_['N, 'M] = {'true}'Bool .
endom
```

4.2 Transforming Object-Oriented Modules into System Modules

We can describe the desired transformation from an object-oriented module to a system module as follows:

- The module `CONFIGURATION` is imported.
- For each class declaration of the form `class` C `|` $a_1 : S_1$`,` ... `,`$a_n : S_n$, the following has to be introduced: a subsort C`@Class` of sort `Cid`, constants $a_1 \dots a_n$ of sort `Aid`, and subsort relations $S_1 \dots S_n$ `< Value`.
- For each subclass relation C `<` C' in the module a subsort declaration C`@Class <` C'`@Class` is introduced, and the set of attributes for objects of class C are completed with those of C'.
- The rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses, that is, not only to objects whose class identifiers are those explicitly given. The rules are then "inherited" by all objects with class identifiers for their subclasses by replacing the class identifiers in the objects in the rules by variables declared of the corresponding class sort. Variables of sort `AttributeSet` are also introduced to range over the additional attributes that may appear. That is, each object `<` O `:` C `|` ... `>` appearing in a rule, is translated into `<` O `:` X `|` ...`,`$Atts$ `>` where the new variable X is declared of sort C`@Class` and the new variable $Atts$ of sort `AttributeSet`.
- As described in [15], we simplify the notation used in object-oriented modules by giving the user the possibility of not mentioning in a given rule those attributes of an object that are not relevant for that rule. To explain this convention, let $\overline{a : v}$ denote the attribute-value pairs $a_1 : v_1, \dots, a_n : v_n$, where the $\overline{a}$ are the attribute identifiers of a given class C (after completing it with all the attributes in its superclasses) having $\overline{S}$ as the corresponding sorts of values prescribed for those attributes. Then, in object-oriented modules we allow rules where the attributes appearing in the lefthand side and righthand side patterns for an object O mentioned in the rule need not exhaust all the object's attributes, but can instead be in any two arbitrary subsets of the object's attributes. We can picture this as follows

$$\dots \langle O : C \mid \overline{al : vl}, \overline{ab : vb} \rangle \dots \longrightarrow \dots \langle O : C \mid \overline{ab : vb'}, \overline{ar : vr} \rangle \dots$$

  where $\overline{al}$ are the attributes appearing only on the *left*, $\overline{ab}$ are the attributes appearing on *both* sides, and $\overline{ar}$ are the attributes appearing only on the *right*. In the transformation into a system module, this rule is translated into

$$\begin{aligned} &\dots \langle O : X \mid \overline{al : vl}, \overline{ab : vb}, \overline{ar : x}, \overline{ac : x'}, Atts \rangle \dots \\ &\quad \longrightarrow \dots \langle O : X \mid \overline{al : vl}, \overline{ab : vb'}, \overline{ar : vr}, \overline{ac : x'}, Atts \rangle \dots \end{aligned}$$

  where X is a variable of sort C`@Class`, $\overline{ac}$ are the attributes defined in the class C that do not appear in $\overline{al}$, $\overline{ab}$ or $\overline{ar}$, the $\overline{x}$ and $\overline{x'}$ are new variables and *Atts* matches the remaining attribute-value pairs. Although the form of the rule obtained is different to the one given in [15], the convention is similar to the convention presented there, the attributes mentioned only on the left are preserved unchanged, the original values of attributes mentioned only on the right do not matter, and all attributes not explicitly mentioned are left unchanged.

The function `transform` : `OModule` $\longrightarrow$ `Module` takes the metarepresentation of an object-oriented module and returns the metarepresentation of the system module into which it is transformed. For example, `transform(`$\overline{\texttt{ACCNT}}$`)` results in the system module

```
mod 'ACCNT is
  protecting 'MACHINE-INT .
  protecting 'CONFIGURATION .
  sorts 'Accnt@Class .
  subsort 'MachineInt < 'Value .
  subsort 'Accnt@Class < 'Cid .
  op 'credit : 'Oid 'MachineInt -> 'Message [none] .
  op 'debit : 'Oid 'MachineInt -> 'Message [none] .
  op 'transfer_from_to_ : 'MachineInt 'Oid 'Oid -> 'Message [none] .
  op 'Accnt: nil -> 'Accnt@Class [none] .
  op 'bal: nil -> 'Aid [none] .
  var 'A : 'Oid .
  var 'B : 'Oid .
  var 'M : 'MachineInt .
  var 'N : 'MachineInt .
  var 'N' : 'MachineInt .
  var 'Atts1 : 'AttributeSet .
  var 'Atts2 : 'AttributeSet .
  var 'X1 : 'Accnt@Class .
  var 'X2 : 'Accnt@Class .
  none
  none
  rl ['credit] :
     '__['credit['A, 'M],
         '<_:_|_>['A, 'X1,
                  '_`,_['_:_[{'bal}'Aid, 'N], 'Atts1]]]
     => '<_:_|_>['A, 'X1,
                 '_`,_['_:_[{'bal}'Aid, '_+_['N, 'M]], 'Atts1]] .
  crl ['debit] :
     '__['debit['A, 'M],
         '<_:_|_>['A, 'X1,
                  '_`,_['_:_[{'bal}'Aid, 'N], 'Atts1]]]
     => '<_:_|_>['A, 'X1,
                 '_`,_['_:_[{'bal}'Aid, '_-_['N, 'M]], 'Atts1]]
        if '_>_['N, 'M] = {'true}'Bool .
```

```
  crl ['transfer] :
     '__['transfer_from_to_['M, 'A, 'B],
         '<_:_|_>['A, 'X1,
                  '_`,_['_:_[{'bal}'Aid, 'N], 'Atts1]],
         '<_:_|_>['B, 'X2,
                  '_`,_['_:_[{'bal}'Aid, 'N'], 'Atts2]]]
     => '__['<_:_|_>['A, 'X1,
                     '_`,_['_:_[{'bal}'Aid, '_-_['N, 'M]], 'Atts1]],
            '<_:_|_>['B, 'X2,
                     '_`,_['_:_[{'bal}'Aid, '_+_['N', 'M]], 'Atts2]]]
        if '_>_['N, 'M] = {'true}'Bool .
endm
```

The corresponding system module at the object level is as follows.

```
mod ACCNT is
  protecting MACHINE-INT .
  protecting CONFIGURATION .
  sort Accnt@Class .
  subsort MachineInt < Value .
  subsort Accnt@Class < Cid .
  ops credit debit : Oid MachineInt -> Message .
  op transfer_from_to_ : MachineInt Oid Oid -> Message .
  op Accnt : -> Accnt@Class .
  op bal : Aid .
  vars A B : Oid .
  vars M N N' : MachineInt .
  vars Atts1 Atts2 : AttributeSet .
  vars X1 X2 : Accnt@Class .
  rl [credit] : credit(A, M) < A : X1 | bal: N, Atts1 >
    => < A : X1 | bal: N + M, Atts1 > .
  crl [debit] : debit(A, M) < A : X1 | bal: N, Atts1 >
    => < A : X1 | bal: N - M, Atts1 >
       if N >= M .
  crl [transfer] : transfer M from A to B
    < A : X1 | bal: N, Atts1 > < B : X2 | bal: N', Atts2 >
    => < A : X1 | bal: N - M, Atts1 > < B : X2 | bal: N' + M, Atts2 >
       if N >= M .
endm
```

5 Conclusions

The module algebra described here is part of an overall reflective and extensible design for the Maude system [6], 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 the development of other notions of parameterized modules. Another research direction opened up by this approach that we plan to pursue is the formal reasoning about the properties of module algebra operations based upon their executable specifications.

Acknowledgments

We have benefited very much from many conversations with our fellow members in the Maude group on the ideas presented in this paper, in particular with Manuel Clavel, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet and José Quesada. We thank all of them, especially Manuel Clavel and Narciso Martí-Oliet, who have read carefully several versions of the paper and have given many helpful suggestions for improvement.

References

[1] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Seventh Colloquium on Automata, Languages and Programming*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[2] 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.

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

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. This volume.

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

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude as a Metalanguage. This volume.

[7] 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.

[8] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89. Elsevier, September 1996.

[9] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 125–147. Elsevier, September 1996.

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

[11] 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, 1993.

[12] 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.

[13] 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, 1992. To appear in J. Goguen and G. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1998.

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

[15] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[16] José Meseguer and Manuel Clavel. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288, 1996.

[17] Timothy Winkler. A short note on the OBJ3 module expression routines. March 1991.