Maude's Module Algebra

Francisco Durán

ETSI Informática, Universidad de Málaga, Spain

José Meseguer

SRI International, Menlo Park, CA, USA

Abstract

The reflective capabilities of rewriting logic and their efficient implementation in the Maude language can be exploited to endow a reflective language like Maude with a module algebra in which structured theories can be combined and transformed by means of a rich collection of module operations. We have followed this approach and we have used the specification of such a module algebra as its implementation, including a user interface and an execution environment for it. The high level at which the specification of the module algebra has been given makes this approach particularly attractive when compared to conventional implementations, because of its shorter development time and the greater flexibility, maintainability, and extensibility that it affords. We explain the general principles of the reflective design of the module algebra and explain the categorical semantics of parameterized theories, modules and views and their instantiation, and the reflective algebraic specification of the different module and view operations.

Key words: Structured theories, module algebra, reflection, Maude, rewriting logic, membership equational logic, parameterization

1 Introduction

Structuring mechanisms for building large specifications in a modular fashion are one of the basic components of most specification languages. Module operations belong to the *metalevel* of the logic in question, that is, they can be seen as functions taking theories and returning other theories as results. This is in fact the idea behind a number of approaches, beginning with the Clear language [5], and further developed in the module expressions in OBJ [34], ACT-ONE [26], ACT-TWO [27], the module algebra by Bergstra, Heering, and Klint in [2], Larch [35], CASL [15], and a vast body of related research [30,4].

In general, any specification language determines a set of constructs which can be used to build specifications. With these constructs, the user has a fixed repertoire of operations available. But he cannot express them, extend them, or reason about them *within the logic*. This means that, although module operations, and in particular parameterization, provide certain "higher-order" capabilities [31], such capabilities remain limited in their applicability.

The introduction of a *reflective logic* [13] can drastically change this situation. Informally, a reflective logic is a logic in which important aspects of its metatheory, including the theories themselves, can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic that can be faithfully interpreted in itself, giving access to its metatheory.

A reflective logic opens up many new possibilities for module composition. Module operations that transform, combine, or manipulate modules can be defined within the logic itself, providing higher-order capabilities and also allowing us to reason about such operations within the logic. A *module algebra* specified in such a way can then be easily modified, thanks to the high level at which the module operations are defined, or extended, by defining new operations in the same way. Furthermore, if the logic is efficiently executable, then the formal specification of module composition operations can itself be used as a reasonable implementation. In addition, it is also possible to use such an implementation in order to create a formal environment for the logic, with tools for formal analysis, transformation, and theorem proving.

This is the approach we have followed to define the module operations for the Maude language [7]. Maude is a high-level reflective language and high-performance system supporting both membership equational logic [39] and rewriting logic [37] specification and programming for a wide range of applications. Rewriting logic is reflective [14], and has very good properties as a logical framework, in which many other logics and many semantic formalisms can be naturally represented [36]. Maude has been influenced in important ways by the OBJ3 language [34], which can be regarded as an equational logic sublanguage.

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective, making the metatheory of rewriting logic accessible to the user in a clear and principled way. We have defined an extensible module algebra for rewriting logic, and have implemented such a design using Maude's reflective capabilities. The language resulting from extending Maude with this module algebra is called Full Maude [1]. The semantics of this module

[1] Full Maude and its documentation are part of the Maude language release, which is available free of charge from its web site at `http://maude.csl.sri.com`.

algebra has been given at two different levels: a logic-independent categorical semantics, which was presented in [23,18], and a rewriting logic semantics given by its algebraic specification, first presented in [25,18,20]. In this paper, we briefly summarize the categorical semantics of structured theories and freeness constraints (about which much more detail can be found in [23]) in Section 3; we then discuss in detail the categorical semantics of parameterized theories, modules, and views and of their instantiation in Section 4; and we give a quite comprehensive overview of Full Maude's algebraic semantics in Sections 5-7.

Parameterized specification and programming is a key modularity and reusability technique crucial for managing the complexity of large specifications and programs. In languages in the Clear/OBJ tradition, including OBJ3 [34], CafeOBJ [16], and Maude [7], there are three key notions: (i) *modules*, which are theories with an *initial* or—in the parameterized case—*free extension* semantics; (ii) *theories*, with a *loose* semantics, that can be used to specify the parameters of modules and to state formal assertions; and (iii) *views*, which are theory interpretations used to instantiate parameter theories, refine specifications, and assert formal properties.

We are thus interested in a module algebra framework in which *theories* with *loose* semantics, and *modules* with *initial* or *free* semantics coexist, giving rise to a uniform setting. In Maude there are two kinds of *modules*, both with initial semantics: functional modules, which are equational theories, and system modules, which are rewrite theories. Full Maude extends Maude with object-oriented notation and with a number of module operations, namely, module hierarchies, module renaming, parameterization of modules, and transformation of object-oriented modules into system modules [2]. Full Maude also provides *theories*, with loose semantics, which can be functional, system, or object-oriented. Theories can also be combined by using the module operations, and they can be parameterized by other theories. The module operations available in Full Maude are inspired and mirror quite closely those in OBJ3 [34], although with some significant differences and with greater generality. A detailed discussion of the language design of Full Maude can be found in [7,18]. Here, we give a brief presentation of the design of the module operations available in Full Maude, but mainly focusing in the features not present in other languages, namely, parameterization of theories, and parameterization, composition, and *lifting* of views.

Typically, theory composition operations begin with theories structured in some way and result in an unstructured, or less structured, specification. That is, structured theories are often "flattened" when being composed. There are

[2] Object-oriented modules are used to specify object-oriented systems but can be desugared into system modules [38,22,18].

however good reasons for preserving their structure. Besides the obvious understandability and design documentation reasons, it is often very useful to consider theory-building operations whose results are structured theories. For example, refining a software design can be best understood as refining structured theories [44]. There are also more intrinsic reasons, namely, when the semantics associated to a structured module *essentially depends on its structure*. For example, we often want to associate to the inclusion of a parameter theory into the body of a parameterized specification a *freeness constraint*, requiring that the models of the body are free extensions of the models of the parameter.

Although a number of concepts and techniques have been suggested both at the theoretical and specification language levels to keep and use the necessary amount of structure for specific purposes, the most satisfactory way of addressing the need for preserving structure is to make structured theories *first-class citizens*. Categorical constructions giving semantics to structured theories and to operations on them have been given in [23,18]. The most basic form of structured theory is that of a *hierarchy* of theory inclusions, in the sense that more complex forms of structured theories can often be *normalized* to theory hierarchies [22,18], perhaps keeping some additional information such as freeness constraints. This is one of the central ideas in our module algebra. We do not have explicit constructors for diagrams in the language, as it is the case in systems like SPECWARE [44]. Instead, we represent module inclusions in these normalized structures as modules that refer to other modules in a database.

The module algebra that we are proposing can be easily extended in different ways. We can, for example, define new module combining operations, or new module transforming operations in the style of the transformation from object-oriented modules to system modules, which would allow us to define new types of modules or other types of structuring units for our language. We give two examples of such extensions in Section 7, but there are many more possibilities. For example, an important area for defining new module operations is viewing *formal tools* as appropriate module operations that analyze and transform specifications.

The specification of Full Maude in Maude involves not only a set of module operations. It specifies in fact a complete language, including its syntax and an execution environment for it. The fact that Full Maude has Maude as a "sublanguage" does simplify the representation map between them, but the same methodology can be used for many other languages and logics [22,18]. Once we have an appropriate representation map expressing a given logic in the rewriting logic framework, we can then define an execution environment for such a logic in the same way.

The rest of the paper is structured as follows. In Section 2 we introduce very briefly the notion of reflection in rewriting logic and in Maude, and discuss the general methodology for building formal tools using rewriting logic's reflection and the high-level architecture of the Full Maude execution environment. Section 3 summarizes the categorical semantics of the module algebra, based on the notion of *structured theories with freeness constraints* of [23], that generalize the theories with freeness constraints in [33] and provide a natural *conceptual unification* of the notions of parameterized module and parameterized theory. In Section 4 we explain the relevant concepts in our language design, emphasizing parameterized programming, and specifically the parameterization of modules, theories, and views, and the operations on views. Section 5 describes the main abstract data types involved in the algebraic specification of the module algebra. In Section 6 we present the general schema for the evaluation of module expressions. Section 7 describes the way of extending the specification of the module algebra with new module operations. We finish with some conclusions and future directions in Section 8.

2 Reflection and the Construction of Formal Tools in Maude

A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational theory, and with R a collection of (possibly conditional) rewrite rules between Σ-terms, which are applied *modulo* the equations E. Given a rewrite theory $\mathcal{R}$, *rewriting logic* [37] then gives simple rules of inference to derive all rewrites $t \longrightarrow t'$ provable in the theory $\mathcal{R}$. Since when the set of rules R is empty a rewrite theory becomes an equational theory, rewriting logic contains equational logic as a sublogic. In the Maude language, the version of equational logic chosen as a sublogic is *membership* equational logic [39], in which the atomic sentences are equations $t = t'$, and membership assertions $t : s$, stating that a term t has a sort s.

Rewriting logic is reflective [14] in the precise sense that there is a finitely presented rewrite theory $\mathcal{U}$ which is *universal*, that is, for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence:

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle,$$

where $\overline{\mathcal{R}}$, $\overline{t}$, and $\overline{t'}$ are terms representing, respectively, $\mathcal{R}$, t, and t' as data elements of $\mathcal{U}$.

Maude's design and implementation [7] systematically exploits the reflective capabilities of rewriting logic, providing key features of the universal theory $\mathcal{U}$ in its built-in module `META-LEVEL` [8]. In particular, `META-LEVEL` has sorts

`Term` and `Module`, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\overline{t}$ of sort `Term` and a term $\overline{\mathcal{R}}$ of sort `Module`. For example, the basic cases in the representation of terms are obtained by subsorts `Constant` and `Variable` of the sort `Qid` of quoted identifiers. Constants are quoted identifiers that contain the constant's name and its type separated by a dot, e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a colon, e.g., `'N:Nat`. Then, a term is constructed in the usual way, by applying an operator symbol to a list of terms.

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

For example, the term `0 + s 0` of sort `Nat` in the module `NAT` in Section 5.1 is metarepresented as `'_+_['0.Nat, 's_['0.Nat]]`.

The module `META-LEVEL` also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, `meta-apply`, `meta-reduce`, and `meta-rewrite`, and also generic parsing and pretty printing functions `meta-parse` and `meta-pretty-print` [7,8].

Reflection, together with the good properties of rewriting logic as a logical framework [36,40], make it quite easy to develop formal tools in Maude for any logic $\mathcal{L}$ of interest, including—as in the case of Full Maude's module algebra—rewriting logic itself. Suppose, for example, that we want to build a theorem prover for a logic, or to implement 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

$$\Phi : \mathcal{L} \longrightarrow \textit{Rewriting Logic}.$$

Such representations are typically very simple and natural [36,40]. Using reflection we can *internalize* such a map as an executable equationally defined function $\bar{\Phi}$. Specifically, we can reify the above representation map Φ by defining an abstract data type $\texttt{Module}_{\mathcal{L}}$ representing theories in the logic $\mathcal{L}$. We can then internalize the representation map Φ as an equationally defined function

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

with `Module` the Maude predefined sort in `META-LEVEL` for representing theories in rewriting logic. Then, we can use the Maude predefined functions, or more complex strategies that use such functions [7,6], to execute in Maude the metarepresentation $\bar{\Phi}(\bar{T})$ 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 [11]. In Maude, these additional metalanguage features are supported as follows: The *syntax definition* for $\mathcal{L}$ is accomplished by defining a data type $\texttt{Grammar}_{\mathcal{L}}$, which 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* [7], that tailor the adequate notions of token and identifier to the language in question; parsing and pretty printing for $\mathcal{L}$ is accomplished by the predefined functions `meta-parse` and `meta-pretty-print` in `META-LEVEL`, in conjunction with the bubble sorts defined for $\mathcal{L}$; and input/output of theory definitions, and of commands for execution in $\mathcal{L}$ is accomplished by a predefined module that provides a generic read-eval-print loop [7].

In the case of Full Maude, for example, we 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, 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 $\bar{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}$ 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

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

Of course, in the case of Full Maude, $\texttt{Module}_{\mathcal{L}} = \texttt{Module}$, and $\bar{\Phi}$ is the identity, but the above design is more general and can be applied to other specification languages based on other logics.

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, theories and views can be parameterized by theories, and both modules and theories 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 in order to evaluate module expressions, evaluate commands, and so on.

3 The Categorical Semantics of Structured Theories

Category theory provides an excellent foundation for studying structured specifications and their composition. We summarize in this section our work on *structured theories* and freeness constraints on such theories [23,18], and how it provides a categorical semantics for structured theories, and in particular for the parameterized theories, views, and modules, and the module operations of Full Maude.

The general notion of *structured theory* is introduced in [23,18] as a way of considering structured theories as *first-class citizens*, so that the structure of each theory is preserved when theories are combined. As in Clear [5] and SPECWARE [44], the categorical notion of diagram is used for giving semantics to structured theories. One can specialize the concept to a hierarchy of theories by requiring that the "diagram scheme" is a finite poset and that all the arrows in the diagram are inclusions in an appropriate subcategory of inclusion morphisms. However, our treatment is done for the more general notion of structured theory. Such a formalization provides a logic-independent categorical semantics based on the notion of institution [33]. In particular, the institution $\boldsymbol{\mathcal{S}(\mathcal{I})}$ of structured theories over a given institution $\boldsymbol{\mathcal{I}}$ is defined as follows.

Definition 1 [43] *Let $\mathcal{C}$ be a category. The* diagram category $Dg(\mathcal{C})$ *has as objects functors $T : \mathcal{P} \to \mathcal{C}$, where $\mathcal{P}$ is a small category. If $T : \mathcal{P} \to \mathcal{C}$ and $T' : \mathcal{P}' \to \mathcal{C}$ are objects, then a morphism $(R, \varrho) : T \to T'$ consists of a functor $R : \mathcal{P} \to \mathcal{P}'$ and a natural transformation $\varrho : T \to T' \cdot R$.*

The composition of morphisms (R, ϱ) and (R', ϱ'), as depicted in the diagram *below, is given by the morphism* $(R' \cdot R,\ \varrho' R \cdot \varrho)$.

$$\begin{tikzcd}
\mathcal{P} \arrow[d, "R"'] \arrow[dr, "T"] \arrow[dr, phantom, "\Downarrow\varrho"', near start] & \\
\mathcal{P}' \arrow[r, "T'"] \arrow[d, "R'"'] \arrow[r, phantom, "\Downarrow\varrho'"', shift right=3ex] & \mathcal{C} \\
\mathcal{P}'' \arrow[ur, "T''"'] &
\end{tikzcd}$$

A *structured signature* can be formalized as a functor $D\colon I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ from a small category I to the category $\mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ of signatures and signature morphisms in a given institution $\boldsymbol{\mathcal{I}}$. We build an institution $\boldsymbol{\mathcal{S}(\mathcal{I})}$, whose theories are called *structured* $\boldsymbol{\mathcal{I}}$*-theories*, by defining functors $\mathrm{sen}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ and $\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ associating to each structured signature D in $\mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ a set of D-sentences and a category of D-models, respectively. Then, we give a satisfaction relation for it.

Definition 2 [23] *Let us denote by* $\mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ *the category* $Dg(\mathbf{Sign}_{\boldsymbol{\mathcal{I}}})$ *of diagrams over the category of signatures in the institution* $\boldsymbol{\mathcal{I}}$. *We shall call the objects of* $\mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ structured ($\boldsymbol{\mathcal{I}}$-)signatures*, and will denote each structured signature by its corresponding diagram* $D : I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$. *The morphisms in* $\mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ *are called* structured signature morphisms.

Definition 3 [23] *The functor* $\mathrm{sen}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}\colon \mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} \to \mathbf{Set}$*, associating to each structured signature* $D\colon I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ *a set of sentences and to each structured signature morphism* $(K, H)\colon D \to D'$ *a corresponding translation at the level of sentences, is defined as follows:*

$$\mathrm{sen}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}(D) = \coprod_{i \in I} \mathrm{sen}_{\boldsymbol{\mathcal{I}}}(D(i))$$

$$\mathrm{sen}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}((K, H)) = \coprod_{i \in I} \mathrm{sen}_{\boldsymbol{\mathcal{I}}}(H_i)$$

We can see each of the sentences of D as a pair (i, φ), where φ is a sentence in $\mathrm{sen}_{\boldsymbol{\mathcal{I}}}(D(i))$.

Definition 4 [23] *Given a structured signature* $D : I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$*, its category of models* $\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}(D)$ *has as objects families* $M = \{M_i\}_{i\in I}$ *with* M_i *in* $\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(D(i))$*, such that for each* $\phi\colon i \to j$ *in* I, $\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(D(\phi))(M_j) = M_i$. *A morphism between two such models* $f : M \to M'$ *is given by a family* $\{f_i\colon M_i \to M'_i\}_{i\in I}$ *with* f_i *in* $\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(D(i))$ *such that for each* $\phi\colon i \to j$ *in* I, $\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(D(\phi))(f_j) = f_i$.

Definition 5 [23] *The functor* $\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}\colon \mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} \to \mathbf{Cat}^{\mathrm{op}}$ *assigns to each structured signature* $D\colon I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ *its category of models* $\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}(D)$*, and to each structured signature morphism* $(K, H)\colon D \to D'$ *the forgetful functor*

$\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}((K,H))\colon \mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}(D') \to \mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}(D)$, *defined as follows:*

$$\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}((K,H))(\{M'_j\}_{j\in I'}) = \{\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(H_i)(M'_{K(i)})\}_{i\in I}$$

$$\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}((K,H))(\{f'_j\}_{j\in I'}) = \{\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(H_i)(f'_{K(i)})\}_{i\in I}$$

Definition 6 [23] *Given a structured signature* $D\colon I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$*, a* D*-model* $M = \{M_i\}_{i\in I}$ *satisfies a* D*-sentence* (i,φ) *if and only if* $M_i \models_{D(i)} \varphi$*. In this case, we write* $M \models_D (i,\varphi)$*.*

Definition 7 [23] *Let* $\boldsymbol{\mathcal{S}(\mathcal{I})}$ *be the institution with:*

- $\mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ *as category of signatures,*
- *the sentence functor* $\mathrm{sen}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}\colon \mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} \to \mathbf{Set}$*, of Definition 3,*
- *the model functor* $\mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}\colon \mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} \to \mathbf{Cat}^{\mathrm{op}}$*, of Definition 5, and*
- *the satisfaction relation given in Definition 6, for which the satisfaction condition holds as shown in [23, Proposition 13].*

Note that the notion of structured $\boldsymbol{\mathcal{I}}$-theory, that is, of a theory presentation in $\boldsymbol{\mathcal{S}(\mathcal{I})}$, captures well the intuitive notion of structured theory found in actual specifications. Indeed, when a subtheory is imported, its axioms typically *are not repeated again*; they are implicitly inherited from the subtheory. This means that axioms are presented *locally*, for a specific local signature $D(i)$, corresponding to our formal notion of a pair (i,φ). It also means that at each stage in the specification only the *incremental information* of additional axioms has to be made explicit. This correspondence of our notion with the actual specification practice can be made even more intuitive by remarking that, in practice, theories are typically *named* entities like `BOOL`, `NAT`, `LIST`, etc. Therefore, we should think of the index set $|I|$ in a structured signature $D\colon I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ as the set of *names* for the different theories present in the structure. Then a sentence such as `nil . L = L` in, say, the theory `LIST` is expressed in our framework as the pair (`LIST`, `nil . L = L`), indicating how the axiom is localized to the `LIST` component of the corresponding structured theory.

Since $\mathbf{Sign}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} = Dg(\mathbf{Sign}_{\boldsymbol{\mathcal{I}}})$, there should be a close and systematic relationship between the category $\mathbf{Th}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ of structured $\boldsymbol{\mathcal{I}}$-theories in the institution $\boldsymbol{\mathcal{S}(\mathcal{I})}$ and the diagram category $Dg(\mathbf{Th}_{\boldsymbol{\mathcal{I}}})$. We can express this relationship as an adjunction with particularly good properties.

Let $J\colon \mathbf{Th}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} \to Dg(\mathbf{Th}_{\boldsymbol{\mathcal{I}}})$ be the functor defined on objects by the equality

$$J(D,\Gamma) = D_{\Gamma^*},$$

where if $D\colon I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ is a structured signature, then $D_{\Gamma^*}\colon I \to \mathbf{Th}_{\boldsymbol{\mathcal{I}}}$ has

$D_{\Gamma^*}(i) = (D(i), \Gamma_i^*)$ and $D_{\Gamma^*}(\phi\colon i \to j) = D(\phi)$, where

$$\Gamma_i^* = \{\varphi \in \mathrm{sen}_{\boldsymbol{\mathcal{I}}}(D(i)) \mid \forall M \in \mathrm{Mod}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}(D,\Gamma), M_i \models_{D(i)} \varphi\}.$$

The definition of J on morphisms assigns to each theory morphism $(K, H)\colon (D, \Gamma) \to (D', \Gamma')$ in $\mathbf{Th}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ the diagram morphism $(K, \tilde{H})\colon D_{\Gamma^*} \to D'_{\Gamma'^*}$ with $\tilde{H}_i = H_i$ for each $i \in I$.

Let $R\colon Dg(\mathbf{Th}_{\boldsymbol{\mathcal{I}}}) \to \mathbf{Th}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ be the functor defined on objects by the equality

$$R(D) = (\mathrm{sign}_{\boldsymbol{\mathcal{I}}} \cdot D,\ \coprod_{i\in I} \mathrm{ax}(D(i))),$$

where, for (Σ, Γ) a theory, we use the notation $\mathrm{ax}(\Sigma, \Gamma) = \Gamma$.

Note that for any theory (D, Γ) in $\mathbf{Th}_{\boldsymbol{\mathcal{S}(\mathcal{I})}}$ we have a natural isomorphism $RJ(D, \Gamma) \overset{\varepsilon_{(D,\Gamma)}}{\simeq} (D, \Gamma)$. Indeed, by construction we have $RJ(D, \Gamma) = (D, \Gamma^\bullet)$. Therefore, both theories are isomorphic, with the identity signature morphism as the isomorphism.

Proposition 8 [23] *The functor $J\colon \mathbf{Th}_{\boldsymbol{\mathcal{S}(\mathcal{I})}} \to Dg(\mathbf{Th}_{\boldsymbol{\mathcal{I}}})$ is full and faithful, with R left adjoint to J and ε as the counit.*

Several key results about the cocompleteness of the categories of structured signatures and theories are given in [23]. One of these results states that, if the category of signatures of $\boldsymbol{\mathcal{I}}$ has colimits, then the categories of signatures and theories of $\boldsymbol{\mathcal{S}(\mathcal{I})}$ both have colimits, making then possible extending the proposal of Goguen and Burstall [5] of taking colimits of theories as a systematic way of "putting theories together" to structured theories, that is, allowing us to use colimits of structured theories as a systematic way of putting structured theories together. In particular, the semantics of the instantiation of structured theories is given by the pushouts in the category of structured theories, which can be obtained, using the functor J, from pushouts in $Dg(\mathbf{Th}_{\boldsymbol{\mathcal{I}}})$ [23].

One of the key motivations for making structured theories a direct object of study is dealing with freeness constraints. They are crucial for the notion of *parameterized module*, in which the model of the parameterized module's body should be a *free extension* of the model of the parameter theory. In many specification languages, including Maude, this leads to a distinction between *theories*, with loose semantics, and *modules*, with initial or, more generally, free extension semantics. Both theories and modules can be parameterized, but in the case of parameterized modules, a freeness constraint between models of the parameter and models of the body is enforced.

Intuitively, freeness constraints are associated to particular theory maps appearing in the diagram of a structured theory, as illustrated, for example, in

Figures 1-2 in Section 4.2, where the maps with freeness constraints are represented by arrows of the form $\Rightarrow$. Suppose a structured theory (D, Γ), with $D\colon I \to \mathbf{Sign}_{\mathcal{I}}$, and consider a morphism $\phi\colon i \to j$ in I. Then, we can associate a freeness constraint to the theory map $D(\phi)\colon D_{\Gamma^*}(i) \to D_{\Gamma^*}(j)$ by requiring that the models M of (D, Γ), in addition to satisfying the axioms Γ, satisfy the constraint

$$M_j \simeq F_{D(\phi)}(M_i)$$

for $F_{D(\phi)}: \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(i)) \to \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(j))$ the left adjoint to the forgetful functor $\mathrm{Mod}_{\mathcal{I}}(D(\phi))\colon \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(j)) \to \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(i))$. For example, $D_{\Gamma^*}(i)$ may be the theory `TRIV`, specifying just one sort `Elt`, and $D_{\Gamma^*}(j)$ may be the theory `LIST` with a sort `List`, specifying lists formed with data elements from the `Elt` parameter sort. Then, the freeness constraint requires that the models of `LIST` are really lists, freely generated from the data elements.

The above notion of freeness constraint should in fact be generalized somewhat, to allow an extra signature map bringing the model to the context in which the constraint is applied. This leads us to the following definition, due to Goguen and Burstall.

Definition 9 [33] *Let $\mathcal{I}$ be an institution. Then a* freeness constraint *on a signature Σ is a pair*

$$c = (H\colon T'' \to T',\ G\colon \mathrm{sign}(T') \to \Sigma)$$

with H a theory morphism and G a signature morphism. A Σ-model M satisfies *c if and only if $\mathrm{Mod}_{\mathcal{I}}(G)(M)$ satisfies T' and $\mathrm{Mod}_{\mathcal{I}}(H)(\mathrm{Mod}_{\mathcal{I}}(G)(M))$ has a free extension along H such that the corresponding component of the counit of the adjunction $\varepsilon_{\mathrm{Mod}_{\mathcal{I}}(G)(M)}: F_H(\mathrm{Mod}_{\mathcal{I}}(H)(\mathrm{Mod}_{\mathcal{I}}(G)(M))) \to \mathrm{Mod}_{\mathcal{I}}(G)(M)$ is an isomorphism; in this case we write $M \models_\Sigma c$.*

Using this idea, the general construction by Goguen and Burstall [33] associating to an institution $\mathcal{I}$ another institution $\mathcal{C}(\mathcal{I})$ of theories with freeness constraints can be used to add such constraints to structured theories. The institution of structured theories with freeness constraints is then given by $\mathcal{S}(\mathcal{C}(\mathcal{I}))$. Therefore, structured $\mathcal{I}$-theories with freeness constraints unify the notions of *theory* (with loose semantics) and *module* (with initial or free extension semantics). That is, both theories and modules are unified in the more general notion of *structured specification with freeness constraints*, whose semantics explicitly depends on their structure. This more general notion serves as a kind of conceptual unification of the notion of module (with initial or freeness constraints) and of theory (with loose semantics) and provides the underlying semantics for the notions of parameterized theory and parameterized module proposed in Section 4. We call a structured theory with freeness constraints a *module* if it has a freeness constraint at the top level, and we call it a *theory* if freeness constraints only appear at lower levels in the structure.

Similarly, parameterized views, as well as lifted views and composed views, are particular morphisms between such structured specifications with freeness constraints.

4 Parameterized Programming in Full Maude

Full Maude extends Maude with object-oriented notation and with a number of module operations. Specifically, a module can import other modules, in *including* and *protecting* modes [34,7], as given by the user or as part of a module expression in which module operations are used to combine or transform previously defined modules. The typical module operations available in the languages in the Clear/OBJ tradition and in other languages are the renaming of modules and the instantiation of parameterized modules with views. Full Maude supports an extended form of parameterized programming in which theories and views can also be parameterized, and in which views can be combined in view expressions by using the composition and lifting view operations, or by instantiating parameterized views with other views. We explain in this section this novel form of parameterized programming.

In Full Maude, like in OBJ3 [34], CafeOBJ [16], and other algebraic specification languages, theories are used to declare the interface requirements of parameterized modules, that is, the semantic properties that must be satisfied by the actual parameter modules used in an instantiation. Nevertheless, it has for long been understood that the full generality and power of a module algebra based on the primitives of the Clear/OBJ tradition requires *parameterized* theories and views, not just parameterized modules [34,32]. In this way, a considerably greater degree of genericity, modularity, and reusability can be achieved for specifications and proofs. However, at present none of the implementations of the algebraic languages we are aware of, except perhaps for some partial prototype effort, supports parameterized theories and views. In our view, the reasons for this omission are both practical and theoretical:

- *practically*, a conventional module algebra implementation requires several man-years, and obviously even longer with parameterized theories and views, while with our reflective approach this implementation effort can be reduced to man-months;
- *theoretically*, parameterized theories and views can be best understood in a full-fledged institution of *structured theories* with freeness constrains $\boldsymbol{\mathcal{S}}(\boldsymbol{\mathcal{C}}(\boldsymbol{\mathcal{I}}))$ over a given institution $\boldsymbol{\mathcal{I}}$, that has only recently been developed [23] (see Section 3).

As functional modules, *functional theories* are membership equational logic theories, but they do not need to be Church-Rosser; they have a *loose* interpretation, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model. System theories and object-oriented theories have a similar loose interpretation. While the semantics of an unparameterized module is the *initial algebra* specified by its theory, the semantics of a *parameterized* module is the free functor associated to the inclusion of the parameter theory into the body of the parameterized module [39]. If the parameterized module has several parameter theories, we should form their colimit, and consider instead the inclusion of such a colimit into the body. For example, a parameterized list module `LIST(X :: TRIV)` forms lists of models of the trivial parameter theory `TRIV` with one sort `Elt`, whose models are sets of elements, and its semantics is the free functor sending each set to the algebra of lists of elements in the set. The case of a *parameterized module*, illustrated by the theory inclusion `TRIV` $\hookrightarrow$ `LIST`, corresponds to freeness constraints of the form

$$(D(\phi)\colon D_{\Gamma^*}(i) \to D_{\Gamma^*}(j),\ 1_{D(j)}\colon D(j) \to D(j)),$$

whereas the case of an *unparameterized module*, like `NAT` or `BOOL`, for which we want an *initial model semantics*, corresponds to a freeness constraint of the form

$$(\emptyset_{D_{\Gamma^*}(j)}\colon \emptyset \to D_{\Gamma^*}(j),\ 1_{D(j)}\colon D(j) \to D(j)),$$

where $\emptyset$ is the initial object in the category of signatures of an exact liberal institution $\boldsymbol{\mathcal{I}}$ (so that $\mathrm{Mod}_{\boldsymbol{\mathcal{I}}}(\emptyset)$ has only one model, let us call it also $\emptyset$) because then the initial model of $D_{\Gamma^*}(j)$ coincides with $F_{\emptyset_{D_{\Gamma^*}(j)}}(\emptyset)$.

The `including` importation of a theory into another theory keeps its loose semantics. However, if the imported theory contains a module, which therefore must be interpreted with an initial semantics,[3] then that initial semantics is maintained by the importation, that is, the constraints in the structure being imported are maintained. For example, in the definition of the `POSET` theory of partially ordered sets below, the declaration `protecting BOOL` ensures that the initial semantics of the functional module for the Booleans is preserved, which is in fact a crucial requirement. This requirement is then preserved by the theory `TOSET` of totally ordered sets when `POSET` is included in it. In fact, we are dealing with a structure in which part of it, not only the top theory, has a loose semantics, while other parts may contain modules with an initial (or, more generally, free) semantics. The kind of semantics of a module or theory

[3] In Full Maude, the importation of a module into a theory is supported only in `protecting` mode [7].

is determined by the keyword used in its definition and by the importation mode.

```
fth POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false .
  ceq X < Z = true
    if X < Y and Y < Z .
endfth
```

```
fth TOSET is
  including POSET .
  vars X Y : Elt .
  eq X < Y or Y < X or X == Y
    = true .
endfth
```

Modules can be parameterized by one or more theories, each of which is labeled, so that one can refer to the different parameters even when the same theory appears several times. The general form for the interface of a parameterized module is $(X_1 :: T_1, \ldots, X_n :: T_n)$, where $X_1, \ldots, X_n$ are the labels, and $T_1, \ldots, T_n$ are the names of the parameter theories[4]. In this, we follow the parameter labeling convention of OBJ3. However, in Maude—unlike OBJ3—sorts are not systematically qualified by their module name. Instead, we assume that all views are named, and that these names are the ones used in the qualification of the sorts in a module. Specifically, any sort S declared in the body of a module M with interface $(\mathtt{X}_1 :: \mathtt{T}_1, \ldots, \mathtt{X}_n :: \mathtt{T}_n)$ can be written in the form $S(X_1, \ldots, X_n)$. When the module is instantiated with views $V_1, \ldots, V_n$, then the corresponding instantiated sort becomes $S(V_1, \ldots, V_n)$. A notion similar to our parameterized sorts has been used in languages like Larch [35], LPG [3], and CASL [15]. However, none of these approaches qualifies sorts in a systematic way as we do.

Thus, a module to define sets can be given as follows.

```
fmod SET(X :: TRIV) is
  sorts Set(X) NeSet(X) .
  subsorts X.Elt < NeSet(X) < Set(X) .
  op mt : -> Set(X) .
  op __ : Set(X) Set(X) -> Set(X) [assoc comm id: mt] .
  op __ : NeSet(X) NeSet(X) -> NeSet(X) [assoc comm id: mt] .
  op _in_ : X.Elt Set(X) -> Bool .
  vars E E' : X.Elt .
  var S : Set(X) .
  eq E E = E .
  eq E in E' S = E == E' or E in S .
  eq E in mt = false .
endfm
```

[4] Note, however, that the parameter theories $T_1, \ldots, T_n$ can themselves be *parameterized* by other theories. We explain parameterized theories in Section 4.2

This convention for naming the sorts of a parameterized module avoids lengthy qualifications by module expressions and many unintended collisions of sort names, thus making sort names simple and renaming practically unnecessary when importing different instances of the same parameterized module.

4.2 Parameterized Theories and Views

Suppose modules `LIST(X :: TRIV)` and `SET(X :: TRIV)`, specifying, respectively, lists and sets, and suppose that we need the data type of lists of sets of natural numbers. Typically, we first instantiate the module `SET` with a view, say `Nat`, from `TRIV` to the module `NAT` mapping the sort `Elt` to the sort `Nat`, thus getting the module `SET(Nat)` of sets of natural numbers. Then, we instantiate the module specifying lists with a view, say `NatSet`, from `TRIV` to `SET(Nat)`, obtaining the module `LIST(NatSet)`. But, what if we need now the data type of lists of sets of Booleans? Should we repeat the whole process again? One possibility is defining a combined module `SET-LIST(X :: TRIV)`. But what if we later want stacks of sets instead of lists of sets?

We can greatly improve the reusability of specifications by using parameterized views. Let us consider the following parameterized view `Set` from `TRIV` to `SET`, which maps the sort `Elt` to the sort `Set(X)`.

```
view Set(X :: TRIV) from TRIV to SET(X) is
   sort Elt to Set(X) .
endv
```

With this kind of views, which can be seen as structured theory morphisms in which the components of the morphism corresponding to the parameter theories of the target of the morphism are identities, we can keep the parameter part of the target module still as a parameter. With such a parameterized view `Set`, we can now have lists of sets, stacks of sets, and so on, for any instance of `TRIV`, by instantiating the appropriate parameterized module with the appropriate view. For example, given the view `Nat` above, we can have the module `LIST(Set(Nat))` of lists of sets of natural numbers, or lists of sets of Booleans with `LIST(Set(Bool))`, given a view `Bool` from `TRIV` to the built-in module `BOOL`. Similarly, we can have `STACK(Set(Nat))` or `STACK(Set(Bool))`. As we shall explain below, the semantics of such a view instantiation is given by the composition of the corresponding structured theory morphisms.

The use of parameterized views in the instantiation of parameterized modules allows very reusable specifications. However, reusability can still be further improved by the combined use of parameterized views and *parameterized theories*. Although the parameterization of a theory does not involve any freeness constraint on the structured theory, and in fact it has the same effect as in-

$$
\begin{tikzcd}
 & \mathtt{M} & \\
 & \mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2) + \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_4 :: \mathtt{T}_2) \arrow[u, Rightarrow] & \\
\mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2) \arrow[ur, hook] & & \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_4 :: \mathtt{T}_2) \arrow[ul, hook'] \\
\mathtt{X}_2 :: \mathtt{T}_2 \arrow[u, hook] & & \mathtt{X}_4 :: \mathtt{T}_2 \arrow[u, hook]
\end{tikzcd}
$$

Fig. 1. Structure of module $\mathtt{M}(\mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2), \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_4 :: \mathtt{T}_2))$.

cluding one theory into another, in practice it provides a great improvement in expressiveness and reusability.

Indeed, modules, theories, and views can now all be *parameterized by parameterized theories*. That is, the general form for the interface of a parameterized module is still $M(X_1 :: T_1, \ldots, X_n :: T_n)$, where $X_1, \ldots, X_n$ are the labels, and $T_1, \ldots, T_n$ are the names of the parameter theories, but now each of the T_i, for $i = 1 \ldots n$, can follow this very same pattern. For example, we can have a module `MAP(F :: FUN(X :: TRIV, Y :: TRIV))` extending a given function between two parameter sets to a function between the corresponding sets of lists of elements.

Note that, by naming with explicit labels each of the theories in the structured parameters, we can specify which part of the structured parameter is *shared*. This mechanism is illustrated in Figures 1 and 2, where we depict the parameterized modules[5] $\mathtt{M}(\mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2), \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_4 :: \mathtt{T}_2))$ and $\mathtt{M}'(\mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2), \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_2 :: \mathtt{T}_2))$, respectively. Although both parameters in the module $\mathtt{M}(\mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2), \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_4 :: \mathtt{T}_2))$ of Figure 1 include the theory $\mathtt{T}_2$, it is not shared, since we use different labels for each of the two occurrences of $\mathtt{T}_2$. On the other hand, the structure of module $\mathtt{M}'(\mathtt{X}_1 :: \mathtt{T}_1(\mathtt{X}_2 :: \mathtt{T}_2), \mathtt{X}_3 :: \mathtt{T}_3(\mathtt{X}_2 :: \mathtt{T}_2))$ includes only one copy of the theory $\mathtt{T}_2$, since it appears with the same label in its two occurrences.

This idea is quite close to that of renaming part of the structure of the parameters [22] to be able to share part of their structure, which also appears in CafeOBJ [16]. In CafeOBJ one can distinguish between so-called "share" and "non-share" cases, stating the kind of sharing that one wants for the parameters. In the "non-share" case it is still possible to explicitly share some submodule in the structure of the parameter.

Note also that the parameters of the sorts in modules parameterized by pa-

[5] Note the use of arrows of the form $\Rightarrow$ to represent inclusion maps with freeness constraints as pointed out in Section 3.

$$\begin{tikzcd}[column sep=small]
& \mathtt{M'} & \\
& \mathtt{X_1 :: T_1(X_2 :: T_2)} + \mathtt{X_3 :: T_3(X_2 :: T_2)} \arrow[u, Rightarrow] & \\
\mathtt{X_1 :: T_1(X_2 :: T_2)} \arrow[ur, hook] & & \mathtt{X_3 :: T_3(X_2 :: T_2)} \arrow[ul, hook'] \\
& \mathtt{X_2 :: T_2} \arrow[ul, hook'] \arrow[ur, hook] &
\end{tikzcd}$$

Fig. 2. Structure of module $\mathtt{M'(X_1 :: T_1(X_2 :: T_2), X_3 :: T_3(X_2 :: T_2))}$.

rameterized theories must be qualified accordingly. For example, the sorts declared in the parameterized module $\mathtt{M(X_1 :: T_1(X_2 :: T_2), X_3 :: T_3(X_4 :: T_2))}$ will be of the form $\mathtt{S(X_1(X_2), X_3(X_4))}$, while the sorts declared in the module $\mathtt{M'(X_1 :: T_1(X_2 :: T_2), X_3 :: T_3(X_2 :: T_2))}$ will be of the form $\mathtt{S(X_1(X_2), X_3(X_2))}$.

The requirements explained above for modules and theories apply also to parameterized views, whose interfaces have the form $V(X_1 :: T_1, \ldots, X_n :: T_n)$, where $X_1, \ldots, X_n$ are the labels, and $T_1, \ldots, T_n$ are the names of the parameter theories, each of which can be a parameterized theory with the pattern explained above. As unparameterized views, parameterized views define a morphism between a theory, which can be parameterized or not, and a module or theory, which can also be parameterized. However, the interface of the parameterized view must coincide with the interface of the target theory or module, up to a change in the order in which the parameters appear, and the parameters of the source theory must be a subset of the parameters of the target theory or module. More precisely, given a view $V(\vec{X} :: \vec{P'})$ from a theory $T(\vec{Y})$ to a module or theory $T'(\vec{X})$, it must be the case that each parameter $Y_i :: P_i$ of the source theory is a *labeled subtheory* of some $X_j :: P'_j$, where by a labeled subtheory of a given labeled theory we mean a labeled theory appearing in its nested structure, including itself. For example, the labeled subtheories of `X :: T(Y :: T'(Z :: T''))` are itself, `Y :: T'(Z :: T'')`, and `Z :: T''`.

Given the parameterized view $\mathtt{V}(\vec{X} :: \vec{P'})$ from $T(\vec{Y})$ to $T'(\vec{X})$, and given views $\mathtt{W}_1 : \mathtt{P}'_1 \to \mathtt{Q}_1, \ldots, \mathtt{W}_n : \mathtt{P}'_n \to \mathtt{Q}_n$, we can define the instantiation $\mathtt{V}(\mathtt{W}_1, \ldots, \mathtt{W}_n)$ of the parameterized view $\mathtt{V}$ by the views $\mathtt{W}_1, \ldots, \mathtt{W}_n$ as the composition of the view $\mathtt{V}$ and of the pushout of $\mathit{Colim}(\vec{\mathtt{W}})$ along the inclusion of $\mathit{Colim}(\vec{\mathtt{P'}})$ into $\mathtt{T}'$ depicted in the diagram of Figure 3, where $\mathit{Colim}(\vec{\mathtt{P}})$, $\mathit{Colim}(\vec{\mathtt{P'}})$, and $\mathit{Colim}(\vec{\mathtt{Q}})$ denote the colimits of $\vec{\mathtt{P}}$, $\vec{\mathtt{P'}}$, and $\vec{\mathtt{Q}}$, respectively, and where $\mathit{Colim}(\vec{\mathtt{W}})$ denotes the induced morphism between $\mathit{Colim}(\vec{\mathtt{P'}})$ and $\mathit{Colim}(\vec{\mathtt{Q}})$. Thus, for example, the view instantiation `Set(Nat)` in the module expression `LIST(Set(Nat))` above can be depicted as shown in Figure 4.

$$
\begin{tikzcd}
\mathtt{T} \arrow[r, "\mathtt{V}(\vec{X} :: \vec{P'})"'] \arrow[rr, bend left, "\mathtt{V}(\vec{W})"] & \mathtt{T'} \arrow[r] \arrow[dr, phantom, "p.o."] & \mathtt{T'}(\vec{\mathtt{W}}) \\
Colim(\vec{\mathtt{P}}) \arrow[u, hook] \arrow[r, hook] & Colim(\vec{\mathtt{P'}}) \arrow[u, hook] \arrow[r, "Colim(\vec{\mathtt{W}})"'] & Colim(\vec{\mathtt{Q}}) \arrow[u, hook]
\end{tikzcd}
$$

Fig. 3. View instantiation.

$$
\begin{tikzcd}
\mathtt{TRIV} \arrow[r] \arrow[rr, bend left, "\mathtt{Set(Nat)}"] & \mathtt{SET} \arrow[r] \arrow[dr, phantom, "p.o."] & \mathtt{SET(Nat)} \\
\emptyset \arrow[u, hook] \arrow[r, hook] & \mathtt{TRIV} \arrow[u, hook] \arrow[r, "\mathtt{Nat}"'] & \mathtt{NAT} \arrow[u, hook]
\end{tikzcd}
$$

Fig. 4. The `Set(Nat)` view instantiation.

4.3 Composed and Lifted Views

Instantiation of parameterized views is not enough. In order to exploit all the potential of structured theories we must be able to express explicitly the *composition of views*, which we denote with a semicolon. Furthermore, we should also be able to denote views that occur at different levels in the structure of the theories involved, which we call *lifted views*. Lifted views give us a way of denoting structured theory morphisms for which only its components in lower levels are non identities.

Let us consider, for example, the theory `MODULE(X :: RING)` of modules[6] over a ring. We can define a view from `MODULE(X :: RING)` to the theory `VECTOR-SP(X :: FIELD)` of vector spaces over a field by first defining a view `F` from `RING` to `FIELD`,

```
view F from RING to FIELD is
  sort Elt to Elt .
endv
```

and then defining a parameterized view `V-AS-M(X :: FIELD)` from the theory `MODULE(F)(X)` to `VECTOR-SP(X)`.

```
view V-AS-M(X :: FIELD) from MODULE(F)(X) to VECTOR-SP(X) is
  sort Elt to Elt .
  sort X.Elt to X.Elt .
endv
```

[6] Note that the word "module" here refers to the usual algebraic structure generalizing vector spaces, and *not* to the structuring unit with initial or free semantics of the Maude language.

$$\begin{tikzcd}
\mathtt{MODULE} \arrow[r, "\mathtt{MODULE\{F\}}"] & \mathtt{MODULE(F)} \arrow[r, "\mathtt{V\text{-}AS\text{-}M}"] & \mathtt{VECTOR\text{-}SP} \\
\mathtt{RING} \arrow[u, hook] \arrow[r, "\mathtt{F}"'] \arrow[ur, phantom, "p.o."] & \mathtt{FIELD} \arrow[u, hook] \arrow[r, "="'] & \mathtt{FIELD} \arrow[u, hook]
\end{tikzcd}$$

Fig. 5. Views `F` and `V-AS-M`.

Note that, by instantiating the theory `MODULE(X :: RING)` with `F`, we get the parameterized theory `MODULE(F)(X :: FIELD)`. The different views involved are depicted in Figure 5. A specification of the theories can be found in [25].

In Figure 5, we have used the notation `MODULE{F}` to refer to the pushout of `F` along the inclusion of the theory `RING` into `MODULE`. We call `MODULE{F}` *the lifting of* `F` *(to* `MODULE`*)*, since it is just the result of combining `F` with the identity morphism for `MODULE`, that is, we extend `F` in order to obtain the morphism at the right level. More precisely, given a parameterized module or theory $\mathtt{M(X_1 :: T_1, \ldots, X_n :: T_n)}$, and given views $\mathtt{V}_i : \mathtt{T}_i \to \mathtt{K}_i$, with $\mathtt{K}_i$ modules or theories (or more generally, views for some labeled subtheories, see below), for $i = 1 \ldots n$, then $\mathtt{M\{V_1, \ldots, V_n\}}$, called the *lifting of* $\mathtt{V_1, \ldots, V_n}$ *to* $\mathtt{M}$, denotes the theory map given by the following pushout.

$$\begin{tikzcd}
\mathtt{M} \arrow[r, "\mathtt{M\{V_1,\ldots,V_n\}}"] & \mathtt{M(V_1, \ldots, V_n)} \\
\mathit{Colim}(\vec{\mathtt{T}}) \arrow[u, hook] \arrow[r, "\mathit{Colim}(\vec{\mathtt{V}})"'] \arrow[ur, phantom, "p.o."] & \mathit{Colim}(\vec{\mathtt{K}}) \arrow[u, hook]
\end{tikzcd}$$

As said above, there can be lifted views for any of the labeled subtheories of a given theory, or, in other words, we can lift views to any level above it. For example, let us consider a theory with name $\mathtt{T(X_1 :: T_1(X_2 :: T_2, X_3 :: T_3))}$ and views $\mathtt{V_2 : T_2 \to K_2}$ and $\mathtt{V_3 : T_3 \to K_3}$, i.e., views on the labeled subtheories $\mathtt{X_2 :: T_2}$ and $\mathtt{X_3 :: T_3}$. We can lift the views $\mathtt{V_2}$ and $\mathtt{V_3}$ to the level of $\mathtt{T_1}$ as $\mathtt{T_1\{V_2,V_3\}}$, or to the level of `T` as $\mathtt{T\{T_1\{V_2,V_3\}\}}$. These views are depicted in Figure 6. Note that, as above, we denote by $\mathit{Colim}(\mathtt{T_2}, \mathtt{T_3})$ the colimit of theories $\mathtt{T_2}$ and $\mathtt{T_3}$ and by $\mathit{Colim}(\mathtt{K_2}, \mathtt{K_3})$ the colimit of $\mathtt{K_2}$ and $\mathtt{K_3}$, and that we denote as $\mathit{Colim}(\mathtt{V_2}, \mathtt{V_3})$ the induced morphism between $\mathit{Colim}(\mathtt{T_2}, \mathtt{T_3})$ and $\mathit{Colim}(\mathtt{K_2}, \mathtt{K_3})$.

We can now instantiate a module or theory parameterized by `MODULE(X :: RING)`, for example, the module `TENSOR(Y :: MODULE(X :: RING))` specifying the tensor algebra construction, with a view from `MODULE(X :: RING)` to the theory `VECTOR-SP(X :: FIELD)` to specialize the same construction to vector spaces. As depicted in Figure 7, for such an instantiation we can use the composition of the views `MODULE{F}` and `V-AS-M`, which we denote by `MODULE{F} ; V-AS-M`. By instantiating the parameterized module `TENSOR(Y :: MODULE(X :: RING))`

$$
\begin{tikzcd}
\mathtt{T} \arrow[r, "\mathtt{T}\{\mathtt{T}_1\{\mathtt{V}_2,\mathtt{V}_3\}\}"] \arrow[dr, phantom, "p.o."] & \mathtt{T}(\mathtt{T}_1(\mathtt{V}_2,\mathtt{V}_3)) \\
\mathtt{T}_1 \arrow[u, hook] \arrow[r, "\mathtt{T}_1\{\mathtt{V}_2,\mathtt{V}_3\}"] \arrow[dr, phantom, "p.o."] & \mathtt{T}_1(\mathtt{V}_2,\mathtt{V}_3) \arrow[u, hook] \\
Colim(\mathtt{T}_2,\mathtt{T}_3) \arrow[u, hook] \arrow[r, "Colim(\mathtt{V}_2,\mathtt{V}_3)"'] & Colim(\mathtt{K}_2,\mathtt{K}_3) \arrow[u, hook]
\end{tikzcd}
$$

Fig. 6. Views $\mathtt{T}_1\{\mathtt{V}_2,\mathtt{V}_3\}$ and $\mathtt{T}\{\mathtt{T}_1\{\mathtt{V}_2,\mathtt{V}_3\}\}$.

$$
\begin{tikzcd}
\mathtt{TENSOR} \arrow[r] \arrow[dr, phantom, "p.o."] & \mathtt{TENSOR(MODULE\{F\})} \arrow[r] \arrow[dr, phantom, "p.o."] & \mathtt{TENSOR(MODULE\{F\}\ ;\ V\text{-}AS\text{-}M)} \\
\mathtt{MODULE} \arrow[u, hook] \arrow[r, "\mathtt{MODULE\{F\}}"] \arrow[dr, phantom, "p.o."] & \mathtt{MODULE(F)} \arrow[u, hook] \arrow[r, "\mathtt{V\text{-}AS\text{-}M}"'] & \mathtt{VECTOR\text{-}SP} \arrow[u, hook] \\
\mathtt{RING} \arrow[u, hook] \arrow[r, "\mathtt{F}"'] & \mathtt{FIELD} \arrow[u, hook] \arrow[r, "="'] & \mathtt{FIELD} \arrow[u, hook]
\end{tikzcd}
$$

Fig. 7. Instantiation of `TENSOR`.

with the composed view `MODULE{F} ; V-AS-M` we obtain the parameterized module `TENSOR(MODULE{F} ; V-AS-M)(Y :: VECTOR-SP(X :: FIELD))`, which specializes to vector spaces the tensor algebra construction. We can then encapsulate this specialized construction in a module definition such as the following.

```
fmod TENSOR-VSPACE(Y :: VECTOR-SPACE(X :: FIELD)) is
  protecting TENSOR(MODULE{F} ; V-AS-M)(Y(X)) .
  ...
endfm
```

Note that the module `TENSOR(MODULE{F} ; V-AS-M)` is equivalent to the module `TENSOR(MODULE{F})(V-AS-M)`, as can be easily inferred from the diagram in Figure 7.

5 The `Unit`, `View`, and `Database` Abstract Data Types

We now explain how, using reflection, the data types of modules, theories, and views are algebraically specified in the Maude specification of Full Maude. In order to be able to handle the different types of modules, theories, and views, we have extended the module `META-LEVEL` with several data types, and with several transformation functions that act on a database in which the declarations being entered are stored. Thus, 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.

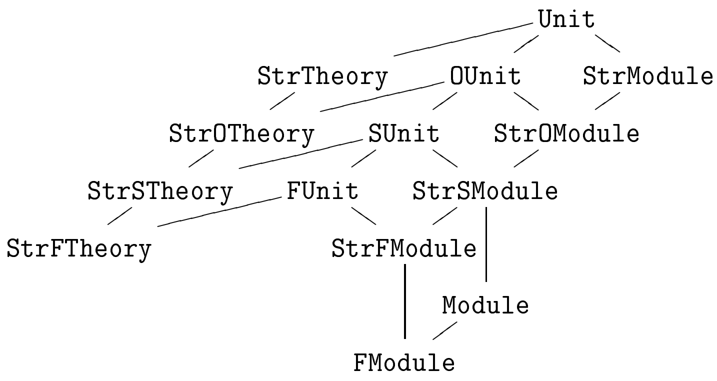

Fig. 8. Hierarchy of Unit Sorts.

5.1 Representing Modules and Theories

The different types of modules and theories are metarepresented in a syntax very similar to the original user syntax as terms of sort `Unit`. The abstract data type `Unit` can be seen as an extension of the predefined sort `Module` in `META-LEVEL`. 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. There are six different types of units, namely, functional, system, and object-oriented modules and theories, which can be parameterized and can import module expressions. There are constructors for each of them in their respective sorts. Modules and theories of any kind are considered to be elements in specific subsorts of the sort `Unit`. The structure of the hierarchy of unit sorts is depicted in Figure 8.

Besides sorts for the different types of modules and theories, there are sorts whose terms represent the different items that can appear in a module. We illustrate the syntax for the top-level operators representing the different types of modules and theories by showing the top-level operator for system theories.

```
op th_is_________endth : ModName ParameterList EImportList
      ESortDecl ESubsortDeclSet EOpDeclSet EMembAxSet EquationSet
      RuleSet -> StrSTheory .
```

To motivate the general syntax for representing modules, we illustrate it with a simple example, namely, a functional module `NAT` for natural numbers with zero and successor, and with a commutative and associative addition operator with `0` as its identity element.

```
fmod NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
```

```
    op _+_ : Nat Nat -> Nat [comm assoc id: 0] .
    vars N M : Nat .
    eq s N + s M = s s (N + M) .
  endfm
```

The representation $\overline{\texttt{NAT}}$ of NAT is the following term of sort StrFModule.

```
  fmod 'NAT is
    nilParameterList
    nil
    sorts 'Nat .
    none
    op '0 : nil -> 'Nat [ctor] .
    op 's_ : 'Nat -> 'Nat [ctor] .
    op '_+_ : 'Nat 'Nat -> 'Nat [comm assoc id('0.Nat)] .
    none
    eq '_+_['s_['N:Nat], 's_['M:Nat]]
      = 's_['s_['_+_['N:Nat, 'M:Nat]]] .
  endfm
```

Since NAT has no parameters, no imported submodules, no subsort declarations, and no membership axioms, those fields are filled with the constants representing empty sets or lists of the appropriate sorts.

5.2 The Abstract Data Type View

Basically, the data elements of sort View, representing theory maps, are expressions containing the name of the view, its list of parameters, the names of the source and target units, and a set of terms representing the maps asserting how the given target unit is claimed to satisfy the source theory. The syntax for the top-level operator representing views is as follows:

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

There are also constructors for the different types of maps that can be used in a view, all of which generate terms of sort ViewMap. Full Maude supports all the maps in OBJ3, plus some others [18,7]. For example, operator maps allow the specification of the operator's arity and coarity, and maps related to object-oriented notions have been incorporated.

Let us motivate the general Full Maude syntax for representing views by illustrating it with a simple example. Given the theory TRIV of sets, with just a sort Elt, and given the previous module NAT of natural numbers, we can define a view Nat from TRIV to NAT as follows.

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

The internal metarepresentation $\overline{\texttt{Nat}}$ of the above view Nat is the following term of sort View:

```
view('Nat, nilParameterList, 'TRIV, 'NAT, sort 'Elt to 'Nat .)
```

This internal representation is stored in the database when the user enters the above view Nat. In case it is needed for the evaluation of some module expression, it will be retrieved from the database and then used.

5.3 The Abstract Data Type Database

In order to be able to refer to modules, theories, and views by name, which is extremely useful for module definition purposes at the user level, the evaluation of module and view expressions takes place in the context of a database, in which we keep information about the modules, theories, and views already introduced in the system, and also about those generated internally. The database is also used to keep structured modules and theories that refer to submodules or subtheories that they include as modules and theories in the database. In turn, such submodules and subtheories may themselves refer to other modules or theories.

6 Operations on Structured Theories

Our module operations take structured theories as arguments and return structured theories as results. For example, the union of two structured theories results in a new structured theory in which the structure of such theories is maintained, and the instantiation of a parameterized structured theory with a parameterized view results in a structured theory, which is still parameterized. Note that terms of sort Unit are structured theories, where the structure is given by module expressions that refer by name to other modules or theories, or combinations of modules or theories, which are stored in the database. However, we define module operations on *normalized* structures, which allows us to simplify the specification of the operations and also to add new operations on structured theories by following a general basic schema. These normal forms, terms of sort Unit as well, are given by hierarchies of *module inclusions*, to which more complex forms of structured theories are transformed.

The normalization of module expressions to structured theories corresponds to the evaluation of the corresponding constructions into the resulting structured theories in the category of structured theories given in Section 3. Since parameter theories can be given by module expressions, the normalization of parameterized modules, theories, or views requires, in addition, the normalization of the possibly structured parameters.

To be able to apply the predefined functions in the module `META-LEVEL` to the modules entered to Full Maude, for example, for execution purposes, we need to map these normalized structures into terms of sort `Module`, which are then compiled into the rewrite engine.

In our approach, transforming a module, theory, or view from its possibly complex structured version to its unstructured form is therefore 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, and which will be used for any purpose other than execution. This first step can be seen as the reduction of a structured specification to its *structured normal form*, which is carried out by the `normalization` function. It takes place in the database, and yields as result a possibly modified version of the database state. Basically, the evaluation of each of these module expressions results in a new module or theory with such a module expression as its name. Then, in a second step, this structured normal form is flattened into an unstructured specification, resulting in a term of sort `Module`.

The evaluation of structured modules and theories is accomplished by the function `evalUnit`, which takes a unit and a database state and returns the database after including the resulting module or theory in it. Similarly, the function `evalView` takes a view and a database state and returns the database resulting from evaluating such a view in it.

```
op evalUnit : Unit Database -> Database .
op evalView : View Database -> Database .
```

6.1 The General Schema for the Evaluation of Units

The schema followed for evaluating module expressions is very simple and can be presented in a generic way. In our design, the evaluation of a unit consists in first evaluating any module expression in it, which may result in the generation of new modules with such module expressions as *names*, and then reducing its structure to a canonical form in which only unit inclusions appear, that is, to a unit hierarchy.

We illustrate this process with a very simple example, namely, the importation of a renamed module. Let us consider the following specification `NAT` of natural numbers given in Section 5.1, and a specification `NAT3` of the natural numbers modulo three, given by importing a renamed copy of the module `NAT` and adding the corresponding congruence equation.

```
fmod NAT3 is
  including NAT * (sort Nat to Nat3) .
  eq s s s 0 = 0 .
endfm
```

The renaming process is summarized in the diagram below. 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 "confusion" created by the congruence modulo three equation has only been introduced in the sort `Nat3`, which has nothing to do with the sort `Nat`.

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

The metarepresentation of the module `NAT3` before being evaluated is given by the following term of sort `StrFModule`:

```
fmod 'NAT3 is
  nilParameterList
  including 'NAT * (sort 'Nat to 'Nat3) .
  sorts none .
  none none none
  eq 's_['s_['s_['0.Nat3]]] = '0.Nat3 .
endfm
```

6.2 Normalization of Units and Views

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.

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 is as follows:

```
fmod 'NAT * (sort Nat to Nat3) is
  nilParameterList
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [ctor] .
  op 's_ : 'Nat3 -> 'Nat3 [ctor] .
  op '_+_ : 'Nat3 'Nat3 -> 'Nat3 [comm assoc id('0.Nat3)] .
  none
  eq '_+_['s_['N:Nat3], 's_['M:Nat3]]
    = 's_['s_['_+_['N:Nat3, 'M:Nat3]]] .
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 its name is entered into the database. In this way, the normalization yields a structured module in which only inclusions appear.

The module expressions appearing as formal parameters of modules, theories, and views must be evaluated as well. Similarly, and although they will not appear in normalized structured theories, view expressions are also evaluated following the same schema and for the same reasons. However, the normal forms of views is quite simpler, since there are no view inclusions. Each composition and lifting of views generates an equivalent new view with such a view expression as its name.

Module and view expressions are evaluated using the functions `evalModExp` and `evalViewExp`, respectively, which, given the sorts `ModExp` and `ModView` for representing module and view expressions, respectively, are declared as follows.

```
op evalModExp : ModExp Database -> Database .
op evalViewExp : ViewExp Database -> Database .
```

There are equations giving semantics to `evalModExp` and `evalViewExp` on each of the constructors of module and view expressions. In addition to the appropriate declarations, the introduction of a new module (resp. view) expression operator requires the addition of new equations defining the behavior of `evalModExp` (resp. `evalViewExp`) on such a new operator (see Section 7 for two examples).

6.3 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* [5,17]. However, instead of considering the category of specifications and specification morphisms [5], 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.

Note that in the flattening of a structured theory 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 [34], 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 is a common occurrence in the presence of parameterized modules and theories. The parameterized sorts discussed in Section 4 provide the mechanism available in Maude to solve these name collisions. A comparison of the approaches taken in Maude and in OBJ3 and other languages can be found in [18], and a detailed discussion of parameterized sorts can be found in [7,18].

7 Extending Full Maude

We illustrate in this section some of the ways 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 another example illustrating the possibility of defining completely new module operations we present a module expression such that, given any nonzero natural number, it generates a parameterized module specifying tuples of the corresponding size. The evaluation of these new module expressions follows the general schema for the evaluation of module expressions discussed in Section 6.2. That is, a new module with such module expression as its name is created, giving to the expression the intended semantics.

In order to add any new module expressions the first thing we must do is to extend the signature of Full Maude with the declarations for the syntax of the new module expressions. Moreover, the declaration of any new kind of module expression must come together with the definition of several functions on the new module operator. We shall focus here on the specification of the `evalModExp` function for these new module expression operators. See [18] for a detailed discussion on the rest of definitions required.

7.1 The Union Module Expression

The syntax used for the union of module expressions is

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

The evaluation of a union module expression results in the creation of a new unit, with such a module expression as its name, which imports the two module expressions being combined. 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 highest of the sorts of its two arguments. For example, the union of an object-oriented module and a functional module will be an object-oriented module. If one of the two module expressions corresponds to a theory, then a theory is generated.

Therefore, assuming variables `ME` and `ME'` of sort `ModExp` and a variable `DB` of sort `Database`, and with `addImportList`, `setName`, and `getUnit` functions to, respectively, add a list of importations to a module, set the name of a module, and retrieve a unit with a particular name from a database state, the equation defining the semantics of `evalModExp` for the union module expression is as follows.

```
ceq evalModExp(ME + ME', DB)
  = evalUnit(
      addImportList((including ME .) (including ME' .),
        setName(
          rightEmptyUnit(
            getUnit(ME, evalModExp(ME, evalModExp(ME', DB)))),
          ME + ME')),
      evalModExp(ME, evalModExp(ME', DB)))
    if not unitInDatabase(ME + ME', DB) .
```

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

```
op TUPLE[_] : NzNat -> 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. For example, the module expression `TUPLE[2]` produces as its result the following module.

```
fmod TUPLE[2](C1 :: TRIV, C2 :: TRIV) is
  sorts Tuple(C1, C2) .
  op (_,_) : C1.Elt C2.Elt -> Tuple(C1, C2) [ctor].
  op p1_ : Tuple(C1, C2) -> C1.Elt .
  op p2_ : Tuple(C1, C2) -> C2.Elt .
  var E1 : C1.Elt . var E2 : C2.Elt .
  eq p1(E1, E2) = E1 .
  eq p2(E1, E2) = E2 .
endfm
```

Even though the n-tuple module expression is in principle of a completely different nature than the usual Clear/OBJ module operations, 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 its name.

The equation for the `evalModExp` is reduced to the creation of a module as indicated above. Given variables `N` and `DB` of sorts `NzNat` and `Database`, respectively, and given some auxiliary functions to generate the different declarations in the module, the equation defining the semantics of `evalModExp` for the n-tuple module expression is as follows.

```
ceq evalModExp(TUPLE[N], DB)
  = evalUnit(fmod TUPLE[N] is
               tupleParameterList(N)
               tupleImportList(N)
               sorts tupleSortSet(N) .
               none
               tupleOpDeclSet(N)
               tupleVarDeclSet(N)
               none
               tupleEquationSet(N)
             endfm, DB)
    if not unitInDatabase(TUPLE[N], DB) .
```

7.3 Other Extensions

We can extend Full Maude in different ways. Indeed, the module algebra has been extended following the approach discussed since it was first specified [18], and we plan a number of new extensions for it, for example, module operations to hide sorts and operators, and operations to redefine and remove messages in object-oriented modules.

An important area for defining new module operations is viewing *formal tools* as appropriate module operations that analyze and transform specifications. Substantial examples of this kind are:

- The inductive theorem prover developed by Clavel *et al.* [9,10] and the Church-Rosser checker presented in [18,9,10,24]. Both tools can be used to prove properties of equational specifications in Maude [7].
- The termination checker and Knuth-Bendix completion tools for Maude equational specifications [21].
- The coherence checker and completion tools for Maude specifications [19].
- The Real-Time Maude tool, which extends Full Maude to a specification language and formal analysis tool for rewriting logic specification of real-time systems [42,41].

These tools are written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement. They have a *reflective* design. Indeed, the fact that rewriting logic is a reflective logic and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in all of them [11]. The reflective capabilities of Maude have also been used for building user interfaces and complete execution environments for these tools. In fact, the execution environment of Full Maude has been extended so that these tools have been completely integrated within it [18]. The integration of these tools in the same framework and the way in which they have been built allows the interoperation of the tools in a very convenient way [11].

The same advantages making a reflective design appropriate for the definition of the proposed module algebra make it appropriate for the definition of formal tools like these. The very high level of abstraction at which the tools have been developed has made it relatively easy to build them, makes understanding their implementation much easier, and will make their maintenance and future extensions much simpler than if a conventional implementation had instead been chosen.

[7] Thanks to the efforts of our colleagues in Japan, both tools have been integrated within the overall Cafe environment [29,28] and can be used to prove formal properties of order-sorted equational specifications in CafeOBJ [16].

8 Conclusions and Future Directions

We have described how the reflective capabilities of rewriting logic and their efficient implementation in the Maude language can be exploited to endow a reflective language like Maude with an extensible module algebra in which structured modules can be combined and transformed by a rich collection of module operations. This has been the approach followed in order to use the specification of such a module algebra as its implementation.

The Full Maude system includes a user interface and an execution environment for it, which have also been implemented using the reflective capabilities of Maude. The reasonable efficiency of the resulting system has allowed us to include it as part of the distribution package of Maude.

The design of Full Maude is just the result of using the general methodology presented in [22,18] to the case of Maude, but the same approach could be followed for defining module algebras and execution environments for other languages and logics, once we have the appropriate representation map. The high level at which the specification of the module algebra has been given makes this approach particularly attractive when compared to conventional implementations, because of its shorter development time and the greater flexibility, maintainability, and extensibility that it affords. In fact, the reflective design of Full Maude follows a general pattern that could be applied to many other specification languages, based on different logics. This suggest the idea of generalizing Full Maude to a *generic module algebra* that, given a logic satisfying some minimum requirements, could be instantiated to provide a module algebra for such a logic. Since the notion of structured theory on which such an algebra will be based is very general and depends only on properties of the underlying institution [23], it will be possible to instantiate such a generic module algebra for a wide variety of specification languages. This will dramatically decrease the effort currently required to implement an algebra of module composition operations for a given a specification language.

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, Full Maude also allows the use of modules with object-oriented notation [38,22,18]. The module algebra thus obtained is easily *extensible*: 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.

Another interesting research direction is formal reasoning about the properties of module algebra operations. Some promising first steps in this direction are the proofs of metalogical properties carried out in the Maude inductive theorem prover by Basin, Clavel, and Meseguer [1] and the use of reflection to specify and prove properties of polytipic functions [12]. It would be very interesting to exploit such techniques for proving module algebra properties, and in particular to relate in a systematic way the categorical semantics given in [23] with the Maude specification of the corresponding module operations.

Acknowledgments

We are very thankful to Narciso Martí-Oliet for his collaboration in the development of this work. We are also very thankful for the many fruitful discussions on Full Maude with the other members of the Maude team, Manuel Clavel, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Quesada, with whom we have designed and built the Maude language and system.

Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312 and NASA Contract NAS2-98073, by Office of Naval Research Contract N00014-99-C-0198, by National Science Foundation Grants CCR-9900334, and by CYCIT project TIC98-0445-C03-03.

References

[1] D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. In *Proceedings of Workshop on Logical Frameworks and Meta-languages, LFM'99*, 1999. Available at `http://www.cs.bell-labs.com/~felty/LFM99/`.

[2] J. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.

[3] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In B. Robinet and R. Wilhelm, editors, *Proceedings of European Symposium on Programming (ESOP'86)*, volume 213 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag, 1986.

[4] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development. A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[5] R. Burstall and J. Goguen. The semantics of Clear, a specification language. In D. 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.

[6] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Lecture Notes. CSLI Publications, 2000.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, 1999. Available at `http://maude.csl.sri.com`.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[9] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium*. CafeOBJ Project, April 1998.

[10] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In K. Futatsugi, A. Nakagawa, and T. Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*, pages 1–31. Elsevier, 2000.

[11] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods (Vol. II)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1704. Springer-Verlag, 1999.

[12] M. Clavel, F. Durán, and N. Martí-Oliet. Polytypic programming in Maude. In K. Futatsugi, editor, *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

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

[14] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proceedings of 1st International Workshop on Rewriting Logic and its Applications (WRLA'96)*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[15] 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.

[16] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.

[17] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In G. Huet, G. Plotkin, and C. Jones, editors, *Proceedings of Workshop on Logical Frameworks*, pages 83–130. Cambridge University Press, 1991.

[18] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, June 1999.

[19] F. Durán. Coherence checker and completion tools for Maude specifications. Technical Report ITI-2000-7, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.csl.sri.com`.

[20] F. Durán. The extensibility of Maude's module algebra. In T. Rus, editor, *Proceedings of 8th International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer-Verlag, 2000.

[21] F. Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Technical Report ITI-2000-6, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.csl.sri.com`.

[22] F. Durán and J. Meseguer. An extensible module algebra for Maude. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 185–206. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[23] F. Durán and J. Meseguer. Structured theories and institutions. In M. Hofmann, G. Rosolini, and D. Pavlović, editors, *Proceedings of 8th Conference on Category Theory and Computer Science (CTCS'99)*, volume 29 of *Electronic Notes in Theoretical Computer Science*, pages 71–90. Elsevier, 1999. Available at `http://www.elsevier.nl/locate/entcs/volume29.html`.

[24] F. Durán and J. Meseguer. A Church-Rosser checker tool for Maude equational specifications. Technical Report ITI-2000-5, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.csl.sri.com`.

[25] F. Durán and J. Meseguer. Parameterized theories and views in Full Maude 2.0. In K. Futatsugi, editor, *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[26] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. Springer-Verlag, 1985.

[27] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints*. Springer-Verlag, 1990.

[28] K. Futatsugi, A. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial-Strength Algebraic Formal Method.* Elsevier, 2000.

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

[30] M. Gogolla and M. Cerioli. What is an Abstract Data Type after all? In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science*, pages 499–523. Springer Verlag, 1995.

[31] J. Goguen. Higher-order functions considered unnecessary for higher order programming. In D. Turner, editor, *Proceedings, University of Texas Year of Programming, Institute on Declarative Programming.* Addison-Wesley, 1990. Preliminary version as SRI Tech. Rep. SRI-CSL-88-1, January 1988.

[32] J. Goguen. Types as theories. In G. Reed, A. Roscoe, and R. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford University Press, 1991.

[33] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[34] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer, 2000.

[35] J. Guttag and J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[36] N. Martí-Oliet and J. 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.

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

[38] J. 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.

[39] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[40] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic.* Springer-Verlag, 1999.

[41] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.

[42] P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In K. Futatsugi, editor, *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 361–383. Elsevier, 2000.

[43] H. Schubert. *Categories*. Springer-Verlag, 1972.

[44] Y. Srinivas and R. Jüllig. SPECWARE: Formal support for composing software. In B. Moeller, editor, *Proceedings of Conference on Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 399–422. Springer-Verlag, 1995.