Integrating Maude into Hets

Mihai Codescu, Till Mossakowski, Adrián Riesco, and Christian Maeder

Technical Report 07/10

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

September 2010
(Revised February 2, 2011)

Abstract

Maude modules can be understood as models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. However, Maude lacks the formal tools to perform some of these analyses and thus they can only be done by hand. The Heterogeneous Tool Set Hets is an institution-based combination of different logics and corresponding rewriting, model checking, and proof tools. We present in this paper an integration of Maude into Hets that allows to use the logics and tools already integrated in Hets with Maude specifications. To achieve such integration we have defined an institution for Maude based on preordered algebras and a comorphism between Maude and Casl, the central logic in Hets.

Keywords: rewriting logic, heterogeneous specifications, Maude, Casl

Contents

1 Introduction

Maude [6] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in rewriting logic, a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. The key point is that there are three different uses of Maude modules:

1. As programs, to implement some application. We may have chosen Maude because its features make the programming task easier and simpler than other languages.

2. As formal executable specifications, that provide a rigorous mathematical model of an algorithm, a system, a language, or a formalism. Because of the agreement between operational and mathematical semantics, this mathematical model is at the same time executable. Therefore, we can use it as a precise prototype of our system to simulate its behavior.

3. As models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. For example, we may want to prove that our Maude module terminates; or that a given function, equationally defined in the module, satisfies some properties expressed as first-order formulas.

However, when we follow this last approach we find that, although Maude can automatically perform analyses like model checking of temporal formulas or verification of invariants, other formal analyses have to be done "by hand," thus disconnecting the real Maude code from its logical meaning. Although some efforts, like the Inductive Theorem Prover [8], have been dedicated to palliate this problem, they are restricted to inductive proofs in Church-Rosser equational theories, and they lack the generality to deal with all the features of Maude. Hence, and since it is unlikely that a single system will ever have all the analysis tools required by every programmer, it would be useful to connect Maude to other systems in a correct and general way. Moreover, when designing complex systems it is usually required to use different formalisms to define the different their different parts such as databases, specifications, or real-time mechanisms. For this reason, it would also be interesting to have a framework where different tools can be used and related to each other.

Among all the tools providing such an integration, we have selected the Heterogeneous Tool Set (Hets) [27], an institution-based combination of different logics and corresponding rewriting, model checking, and proof tools. The reasons for this decision are:

- It is formal, that is, it allows the user to reason about the mathematical properties of his specifications. This feature makes it much more adequate for our purposes than the Unified Modeling Language UML [2], probably the best known system of this kind.

- It is multilateral, in the sense that the specifications can be introduced in any of the logics supported by Hets, while other approaches, like the Prosper toolkit [9], which provides several decision procedures and model checkers based in the theorem prover HOL98 [31], only provide one logic, and all specifications must be translated to it before using the tool.

- It focus on codings between logics, unlike other approaches that focus on codings between theories as OMDoc [18], an ontology language for mathematics. The former allows to reason about different elements in different logics, while the latter only permits to reason about different elements in the same logic.

- Several tools have already been integrated into Hets, including the SAT solvers zChaff [22] and MiniSat [14], the automated provers SPASS [37], Vampire [35], and Darwin [15], and the interactive provers Isabelle [30] and VSE [1].

In this paper, we describe an integration of Maude into Hets from which we expect several benefits. On the one hand, Maude will be the first dedicated rewriting engine that is integrated into Hets (so far, only the rewriting engine of Isabelle is integrated, which however is quite specialized towards higher-order proofs). On the other hand, certain features of the Maude module system like views lead to proof obligations that cannot be checked with Maude—Hets will be the suitable framework to prove them, using the above mentioned proof tools; with our approach, we cover arbitrary first-order properties (also written in logics different from Maude), and open the door to automated induction strategies such as those of ISAplanner [11].

The rest of the paper is organized as follows. After briefly introducing rewriting logic in Section 2 and HETS in Section 3, Section 4 describes the institution we have defined for Maude and the comorphism from this institution to CASL. Section 5 shows how development graphs for Maude specifications are built, and then how they are normalized to deal with freeness constraints. Section 6 illustrates the integration of Maude into Hets with the help of an example, while Section 7 outlines the implementation of the integration. Section 8 concludes and outlines some future work.

2 Rewriting logic and Maude

As mentioned in the introduction, Maude modules are executable rewriting logic specifications. Rewriting logic [20] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses *membership equational logic* [4, 21], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort. In the following sections we present both logics and how their specifications are represented as Maude modules.

2.1 Membership equational logic

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n,k}\}_{(k_1 \ldots k_n,k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of membership equational logic are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ.

Models of membership equational logic specifications are Σ-*algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$.

A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [4, 21] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, as well as the construction of initial and free algebras.

2.2 Maude functional modules

Maude functional modules [6, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification. The membership equational logic specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness.

In a functional module we can declare sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly, and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat] = [Nat]`.

An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

2.3 Rewriting logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible.

A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses membership equational logic, as described in the previous sections. A rule in R has the general conditional form[1]

$$(\forall X)\ t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$$

where the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [20] (for a generalization see also [5]).

Models of rewrite theories are called $\mathcal{R}$*-systems*. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. Then, if we keep in this context the notation $\mathcal{A}$ to denote an $\mathcal{R}$-system, a rewrite $t \Rightarrow t'$ is satisfied by $\mathcal{A}$, denoted $\mathcal{A} \models t \Rightarrow t'$, when there is a transition $[\![t]\!]_{\mathcal{A}} \rightarrow_{\mathcal{A}} [\![t']\!]_{\mathcal{A}}$ in the system between the corresponding meanings of both sides of the rewrite, where $\rightarrow_{\mathcal{A}}$ will be our notation for such transitions.

The rewriting logic deduction rules introduced in [20] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial and free models; see [20] for details.

2.4 Maude system modules

Maude system modules [6, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

The executability requirements for equations and memberships in a system module are the same as those of functional modules, namely, confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [6] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [36, 6]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rule in R to a term u' such that $u' =_E t'$ (that is, the equation $t' = u'$ can be deduced from E).

[1] There is no need for the condition listing first equations, then memberships, and then rewrites: this is just a notational abbreviation, they can be listed in any order.

2.5 Advanced features

In addition to the modules presented thus far, we present in this section some other Maude features that will be used throughout this paper. More information on these topics can be found in [6].

2.5.1 Module operations

To ease the specification of large systems, Maude provides several mechanisms to structure its modules. We describe in this section these structuring mechanisms, that will be used later to build the development graphs in HETS.

Maude modules can import other modules in three different modes:

- The `protecting` mode (abbreviated as `pr`) indicates that *no junk and no confusion* can be added to the imported module, where junk refers to new terms in canonical form while confusion implies that different canonical terms in the initial module are made equal by equations in the imported module.
- The `extending` mode (abbreviated as `ex`) indicates that junk is allowed but confusion is forbidden.
- The `including` mode (abbreviated as `inc`) allows both junk and confusion.

More specifically, these importation modes do not import modules but *module expressions* that, in addition to a single module identifier, can be:

- A summation of two module expressions ME_1 `+` ME_2, which creates a new module that includes all the information in its summands.
- A renaming ME `*(`*Renaming*`)`, where *Renaming* is a list of renamings. They can be renaming of sorts:

  $$\texttt{sort}\ sort_1\ \texttt{to}\ sort_2\ \texttt{.}$$

  of operators, distinguishing whether it renames all the operators with the given identifier (when the attributes are modified, only `prec`, `gather`, and `format` are allowed, see [6])

  $$\texttt{op}\ id_1\ \texttt{to}\ id_2\ \texttt{.}$$
  $$\texttt{op}\ id_1\ \texttt{to}\ id_2\ \texttt{[}atts\texttt{]}\ \texttt{.}$$

  or it renames the operators of the given arity:

  $$\texttt{op}\ id_1\ \texttt{:}\ arity\ \texttt{->}\ coarity\ \texttt{to}\ id_2\ \texttt{.}$$
  $$\texttt{op}\ id_1\ \texttt{:}\ arity\ \texttt{->}\ coarity\ \texttt{to}\ id_2\ \texttt{[}atts\texttt{]}\ \texttt{.}$$

  or of labels:

  $$\texttt{label}\ label_1\ \texttt{to}\ label_2\ \texttt{.}$$

2.5.2 Theories

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: functional theories and system theories, with the same structure of their module counterparts, but with a different semantics. Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can also have rules. Although there is no restriction on the operator attributes that can be used in a theory, there are some subtle restrictions and issues regarding the mapping of such operators (see Section 2.5.3). Like functional modules, functional theories are membership equational logic theories, but they do not need to be Church-Rosser and terminating.

For example, we can define a theory for some processes. First, we indicate that a sort for processes is required:

```
fth PROCESS is
 pr BOOL .

 sort Process .
```

Then, we state that two operators, one updating the processes and another one checking whether a process has finished, have to be defined:

```
 op update : Process -> Process .
 op finished? : Process -> Bool .
```

Finally, we require an operator `_<_` over processes that is required to be irreflexive and transitive:

```
 vars X Y Z : Process .

 op _<_ : Process Process -> Bool .
 eq X < X = false [nonexec label irreflexive] .
 ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
endfth
```

2.5.3 Views

We use views to specify how a particular target module or theory satisfies a source theory. In general, there may be several ways in which such requirements might be satisfied by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort and operator in the source theory. The source and target of a view can be any module expression, with the source module expression evaluating to a theory and the target module expression evaluating to a module or a theory. Each view declaration has an associated set of proof obligations, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques. Such proof obligations are not discharged or checked by the system.

The mappings allowed in views are:

- Mappings between sorts:

  `sort` $sort_1$ `to` $sort_2$ `.`

- Mappings between operators, where the user can specify the arity and coarity of the operators to disambiguate them:

  `op` id_1 `to` id_2 `.`

  `op` id_1 `:` $arity$ `->` $coarity$ `to` id_2 `.`

- In addition to these mappings, the user can map a term $term_1$, that can only be a single operator applied to variables, to any term $term_2$ in the target module, where the sorts of the variables in the first term have been translated by using the sort mappings. Note that in that case the arity of the operator in the source theory and the one in the target module can be different:

  `op` $term_1$ `to term` $term_2$ `.`

Notice that we cannot map labels, and thus we cannot identify the statements in the theory with those in the target module.

We can now create a view `NatProcess` from the theory `PROCESS` in the previous section to `NAT`, the predefined module for natural numbers:

```
view NatProcess from PROCESS to NAT is
```

We need a sort in `NAT` to identify processes. We use `Nat`, the sort for natural numbers:

```
sort Process to Nat .
```

Since we identify now processes with natural numbers, we can `update` a process by applying the successor function, which is declared as `s_` in `NAT`:

```
op update to s_ .
```

We map the operator `finished?` in a different way: we create a term with this operator with a variable as argument, and it is mapped to a term in the syntax of the target module. In that case we consider a process has finished if it reaches `100`:

```
op finished?(P:Process) to term P:Nat < 100 .
```

Since the `NAT` module already contains an operator `_<_`, it is not necessary to explicitly indicate the corresponding mapping, i.e., identity mappings for sorts and operators can be omitted when defining views.

2.5.4 Parameterized modules

Maude modules can be parameterized. A parameterized system module has syntax

$$\texttt{mod M\{}X_1 :: T_1, \ldots, X_n :: T_n\texttt{\} is ... endm}$$

with $n \geq 1$. Parameterized functional modules have completely analogous syntax.

The $\{X_1 :: T_1, \ldots, X_n :: T_n\}$ part is called the interface, where each pair $X_i :: T_i$ is a parameter, each X_i is an identifier—the parameter name or parameter label—, and each T_i is an expression that yields a theory—the parameter theory. Each parameter name in an interface must be unique, although there is no uniqueness restriction on the parameter theories of a module. The parameter theories of a functional module must be functional theories.

In a parameterized module M, all the sorts and statement labels coming from theories in its interface must be qualified by their names. Thus, given a parameter $X_i :: T_i$, each sort S in T_i must be qualified as $X_i\$S$, and each label l of a statement occurring in T_i must be qualified as $X_i\$l$. In fact, the parameterized module M is flattened as follows. For each parameter $X_i :: T_i$, a renamed copy of the theory T_i, called $X_i :: T_i$ is included. The renaming maps each sort S to $X_i\$S$, and each label l of a statement occurring in T_i to $X_i\$l$. The renaming has no effect on importations of modules. Thus, if T_i includes a theory T', when the renamed theory $X_i :: T_i$ is created and included into M, the renamed theory $X_i :: T'$ will also be created and included into $X_i :: T_i$. However, the renaming will have no effect on modules imported by either the T_i or T'; for example, if `BOOL` is imported by one of these theories, it is not renamed, but imported in the same way into M. Moreover, sorts declared in parameterized modules can also be parameterized, and these may duplicate, omit, or reorder parameters.

The parameters in parameterized modules are bound to the formal parameters by *instantiation*. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the formal parameters to the corresponding sorts, operators (or expressions), etc. in the actual target. The instantiation of a parameterized module must be made with views explicitly defined previously.

We can define a parameterized module for multisets of the processes shown in Section 2.5.3. This module defines the sort `MSet{X}` for multisets, which is a supersort of `Process`:

```
fmod PROCESS_MSET{X :: PROCESS} is
 sort MSet{X} .
 subsort X$Process < MSet{X} .
```

The constructors of multisets are `empty` for the empty multiset and the juxtaposition operator `__` for bigger multisets:

```
 op empty : -> MSet{X} [ctor] .
 op __ : MSet{X} MSet{X} -> MSet{X} [ctor assoc comm id: empty] .
```

We can also use the operators declared in the view. For example, we can remove a process from the multiset if it is finished:

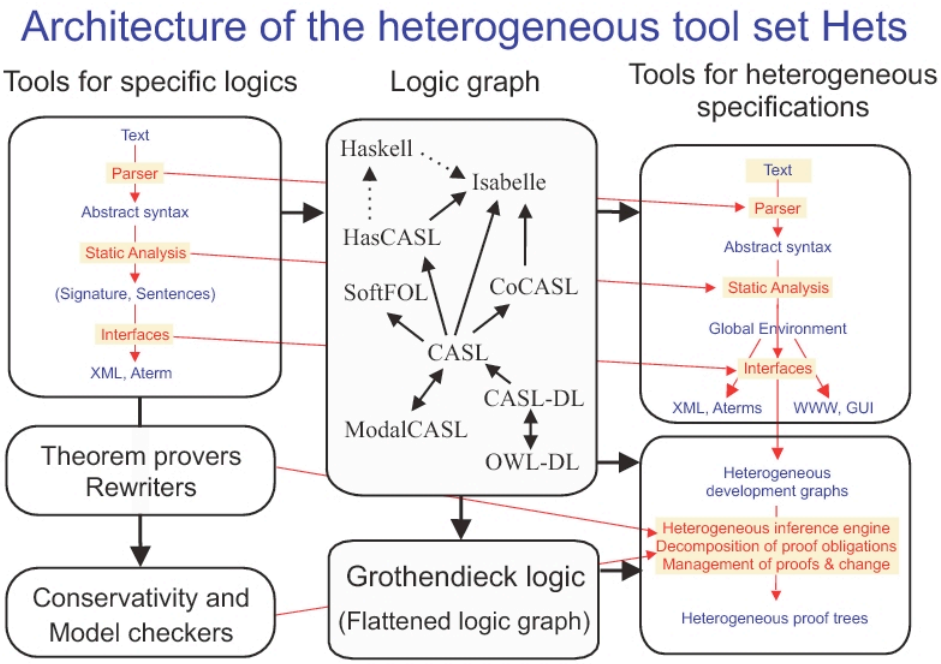

Figure 1: The Heterogeneous Tool Set

```
 var P : X$Process .
 var MS : MSet{X} .

 ceq P MS = MS if finished?(P) .
endfm
```

We can use the view `NatProcess` to instantiate this parameterized module and create multisets of processes identified as natural numbers.

```
fmod NAT_PROCSES_MSET is
 pr PROCESS_MSET{NatProcess} .
endfm
```

3 Hets

The central idea of Hets [24, 27, 28] is to provide a general logic integration and proof management framework. One can think of Hets acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations. The benefit of plugging in a new logic and tool such as Maude into the Hets motherboard is the gained interoperability with the other logics and tools available in Hets; for example, we can now compose the translation from Maude to Casl with the corresponding translation to Isabelle to prove properties in Maude specifications.

Figure 1 shows how, by adding up different tools for single logics, we obtain a bigger tool which supports all of them by (i) providing a small set of mechanisms specific for each tool, (ii) relating the logics in such a way that all of them are connected, and (iii) using these relations to translate the initial specification to another logic where the required tool is available. The current logic graph for Hets is depicted in Figure 2 (where different colors stand for different subgraphs and relations we are not interested here), where each node corresponds to a different logic and the links are relations between them. Note that the central node of this graph is Casl, the Common Algebraic Specification Language [29], which is based in first order logic; new logics are expected to be translated to this logic in order to relate them with all the other logics in an easy and efficient way.

Following the ideas shown in Figure 1, the work that needs to be done for such an integration is to prepare both the Maude logic and tool so that it can act as an expansion card for Hets:

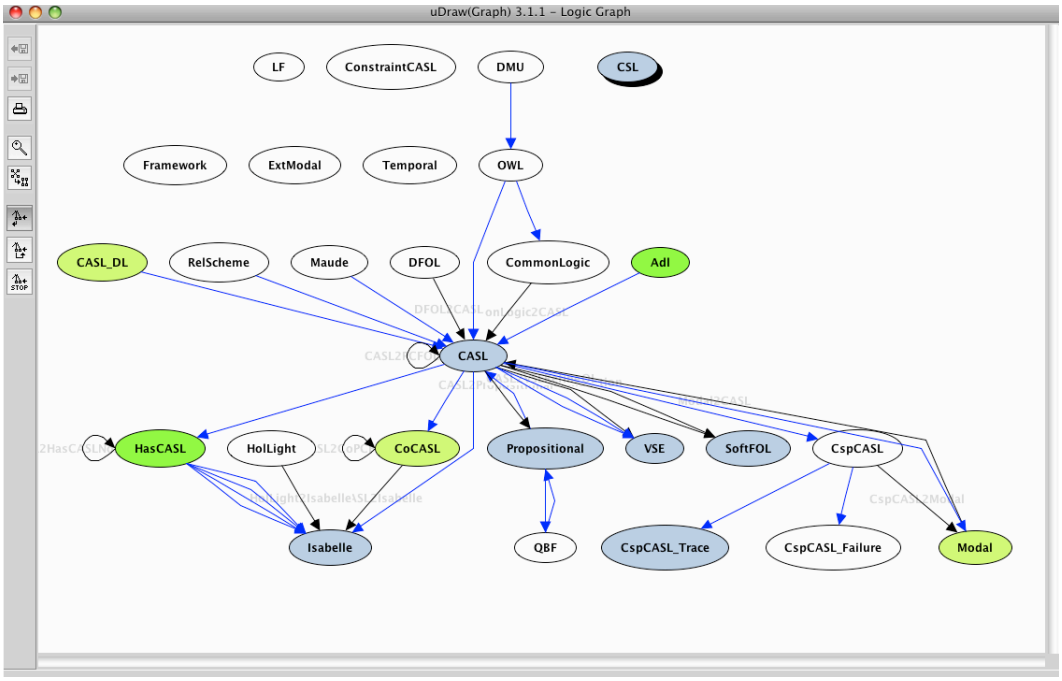

Figure 2: Current logic graph

- On the side of the semantics, this means that the logic needs to be organized as an *institution* [17, 34].
- On the side of the tool, each one must provide a parser and static analysis mechanisms, in such a way that specifications in different logics can be translated to a common framework, which in our case consists of development graphs, a graphical representation of structured specifications.

Before describing institutions we recall the notions of *category*, *small category*, *dual category*, and *functor*.

A *category* $\mathbf{C}$ consists of:

- a class $|\mathbf{C}|$ of *objects*;
- a class $hom(\mathbf{C})$ of *morphisms*, (also known as *arrows*), between the objects;
- operations assigning to each morphism f an object $dom(f)$, its *domain*, and an object $cod(f)$, its *codomain* (we write $f : A \to B$ to indicate that $dom(f) = A$ and $cod(f) = B$);
- a composition operator assigning to each pair of morphisms f and g, with $cod(f) = dom(g)$, a *composite* morphism $g \circ f : dom(f) \to cod(g)$, satisfying the associative law: for any morphisms $f : A \to B$, $g : B \to C$, and $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$; and
- for each object A, an identity morphism $id_A : A \to A$ satisfying the identity law: for any morphism $f : A \to B$, $id_B \circ f = f$ and $f \circ id_A = f$.

A category $\mathbf{C}$ is called *small* if both $|\mathbf{C}|$ and $hom(\mathbf{C})$ are sets and not proper classes.

For each category $\mathbf{C}$, its *dual category* $\mathbf{C}^{op}$ is the category that has the same objects as $\mathbf{C}$ and whose arrows are the opposites of the arrows in $\mathbf{C}$, that is, if $f : A \to B \in \mathbf{C}$, then $f : B \to A \in \mathbf{C}^{op}$. Composite and identity arrows are defined in the obvious way.

Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A *functor* $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ is a map taking each $\mathbf{C}$-object A to a $\mathbf{D}$-object $\mathbf{F}(A)$ and each $\mathbf{C}$-morphism $f : A \to B$ to a $\mathbf{D}$-morphism $\mathbf{F}(f) : \mathbf{F}(A) \to \mathbf{F}(B)$, such that for all $\mathbf{C}$-objects A and composable $\mathbf{C}$-morphisms f and g:

- $\mathbf{F}(id_A) = id_{\mathbf{F}(A)}$,
- $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ \mathbf{F}(f)$.

It is worth mentioning two interesting categories: **Set**, the category whose objects are sets and whose morphisms between sets A and B are all functions from A to B; and **Cat**, the category whose objects are all small categories and whose morphisms are functors between them.

An *institution* consists of:

- a category **Sign** of *signatures*;
- a functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ giving a set $\mathbf{Sen}(\Sigma)$ of Σ-sentences for each signature $\Sigma \in |\mathbf{Sign}|$;
- A functor $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$, giving a category $\mathbf{Mod}(\Sigma)$ of Σ-*models* for each signature $\Sigma \in |\mathbf{Sign}|$; and
- for each signature $\Sigma \in |\mathbf{Sign}|$, a *satisfaction relation* $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ between models and sentences such that for any signature morphism $\sigma : \Sigma \to \Sigma'$, Σ-sentence $\varphi \in \mathbf{Sen}(\Sigma)$ and Σ'-model $M' \in |\mathbf{Mod}(\Sigma)|$:

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \iff \mathbf{Mod}(\sigma)(M') \models_\Sigma \varphi$$

  which is called the *satisfaction condition*.

It is also important to define, for our purposes in this work, the notion of *institution comorphism* [32]. Given two institutions $\mathcal{I} = (\mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models)$ and $\mathcal{I}' = (\mathbf{Sign}', \mathbf{Mod}', \mathbf{Sen}', \models')$, an *institution comorphism* from $\mathcal{I}$ to $\mathcal{I}'$ consists of $\Phi : \mathbf{Sign} \to \mathbf{Sign}'$, a natural transformation $\alpha : \mathbf{Sen} \Rightarrow \Phi; \mathbf{Sen}'$, and a natural transformation $\beta : \Phi; \mathbf{Mod}' \Rightarrow \mathbf{Mod}$ such that the following satisfaction condition holds for each $\Sigma \in |\mathbf{Sign}|$, $f \in |\mathbf{Sen}(\Sigma')|$, and $m' \in |\mathbf{Mod}'(\Phi(\Sigma))|$:

$$\beta_\Sigma(m') \models_\Sigma f \iff m' \models'_{\Phi(\Sigma)} \alpha_\Sigma(f)$$

Institutions capture in a very abstract and flexible way the notion of a logical system, by leaving open the details of signatures, models, sentences (axioms), and satisfaction (of sentences in models). The *satisfaction condition* states that *truth is invariant under change of notation* (also called enlargement of context), which is captured by the notion of signature morphism (which leads to translations of sentences and reductions of models). See [17] for formal details.

Indeed, Hets has interfaces for plugging in the different components of an institution: signatures, signature morphisms, sentences, and their translation along signature morphisms. Recently, even (some) models and model reducts have been covered, although this is not needed here. Note, however, that the model theory of an institution (including model reducts and the satisfaction condition) is essential when relating different logics via institution comorphisms. The logical correctness of their use in multi-logic proofs is ensured by model-theoretic means.

For proof management, Hets uses *development graphs* [25]. They can be defined over an arbitrary institution, and they are used to encode structured specifications in various phases of the development. Roughly speaking, each node of the graph represents a theory. The links of the graph define how theories can make use of other theories. In this way, we represent complex specifications by representing each component (e.g. each module) as a node in the development graph and the relation between them (e.g. importations) as links.

A *development graph* is an acyclic, directed graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$, where:

- $\mathcal{N}$ is a set of nodes. Each node $N \in \mathcal{N}$ is a pair (Σ^N, Φ^N) such that Σ^N is a signature and $\Phi^N \subseteq \mathbf{Sen}(\Sigma^N)$ is the set of **local axioms** of N.
- $\mathcal{L}$ is a set of directed links, so-called **definition links**, between elements of $\mathcal{N}$. Each definition link from a node M to a node N is either
  - **global** (denoted $M \xrightarrow{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or
  - **local** (denoted $M \xrightarrow{\sigma} N$), again annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or
  - **hiding** (denoted $M \xrightarrow[hide]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^N \to \Sigma^M$ *going against the direction of the link*, or
  - **free** (denoted $M \xrightarrow[free]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma \to \Sigma^M$ *where* Σ *is a subsignature of* Σ^M.

In addition to these links we add a new link, denoted $M \xRightarrow[n.p.free]{\sigma} N$, that stands for non-persistent free links and will be used when dealing with `protecting` importations in Maude modules. However, these nodes are only used for Maude specifications and thus are nonstandard, we will show in Section 5.2 how these links and the associated nodes are transformed into a new graph that only uses standard constructions. Intuitively, these links indicate that no new elements can be added to the sorts, although they can be added to the kind.

Given a node M in a development graph $\mathcal{DG}$, its associated class $\mathbf{Mod}_{\mathcal{DG}}(M)$ of models (or M-models for short) is inductively defined to consist of those Σ^M-models m for which

1. m satisfies the local axioms Φ^M,
2. for each $N \xRightarrow{\sigma} M \in \mathcal{DG}$, $m|_\sigma$ is an N-model,
3. for each $N \xrightarrow{\sigma} M \in \mathcal{DG}$, $m|_\sigma$ satisfies the local axioms Φ^N,
4. for each $N \xRightarrow[hide]{\sigma} M \in \mathcal{DG}$, m has a σ-expansion m' (i.e. $m'|_\sigma = m$) that is an N-model, and
5. for each $N \xRightarrow[free]{\sigma} M \in \mathcal{DG}$, m is an N-model that is persistently σ-free in $\mathbf{Mod}(N)$. The latter means that for each N-model m' and each model morphism $h : m|_\sigma \to m'|_\sigma$, there exists a unique model morphism $h^\# : m \to m'$ with $h^\#|_\sigma = h$.

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Again, we distinguish between local and global theorem links (denoted by $N \overset{\sigma}{=\!\Rightarrow} M$ and $N \overset{\sigma}{-\!\!\succ} M$ respectively). We also need theorem links $N \overset{\sigma}{\underset{hide}{=\!=\!\Rightarrow}}_\theta M$ (where for some Σ, $\theta : \Sigma \to \Sigma^N$ and $\sigma : \Sigma \to \Sigma^M$) involving hiding. The semantics of theorem links is given as follows:

Let $\mathcal{DG}$ be a development graph and N, M nodes in $\mathcal{DG}$.

- $\mathcal{DG}$ **implies** a global theorem link $N \overset{\sigma}{=\!\Rightarrow} M$ (denoted $\mathcal{DG} \models N \overset{\sigma}{=\!\Rightarrow} M$) iff for all $m \in Mod(M)$, $m|_\sigma \in Mod(N)$.
- $\mathcal{DG}$ **implies** a local theorem link $N \overset{\sigma}{-\!\!\succ} M$ (denoted $\mathcal{DG} \vdash N \overset{\sigma}{-\!\!\succ} M$) iff for all $m \in Mod(M)$, $m|_\sigma \models \phi$ for all $\phi \in \Phi^N$.
- $\mathcal{DG}$ **implies** a hiding theorem link $N \overset{\sigma}{\underset{hide}{=\!=\!\Rightarrow}}_\theta M$ (denoted $\mathcal{DG} \models N \overset{\sigma}{\underset{hide}{=\!=\!\Rightarrow}}_\theta M$) iff for all $m \in Mod(M)$, the $m|_\sigma$ has a θ-expansion to some N-model.

We refer to [26, 24] for more details on how to use development graphs.

4 Relating the Maude and CASL logics

In this section, we will relate Maude and CASL at the level of logical systems. The structuring level will be considered in the next section by means of development graphs.

4.1 Maude

As we have seen in Section 2, Maude is an efficient tool for equational reasoning and rewriting. Methodologically, Maude specifications are divided into a specification of the data objects and a specification of some concurrent transition system, the states of which are given by the data part. Two logics have been introduced and studied in the literature for this binary relation: rewriting logic [20] and preordered algebra [16]. They essentially differ in the treatment of rewrites: whereas in rewriting logic, rewrites are named, and different rewrites between two given states (terms) can be distinguished (which corresponds to equipping each carrier set with a category of rewrites), in preordered algebra, only the existence of a rewrite does matter (which corresponds to equipping each carrier set with a preorder of rewritability).

Rewriting logic has been announced as the logic underlying Maude [6]. Maude modules lead to rewriting logic theories, which can be equipped with loose semantics (`fth`/`th` specifications) or initial/free

semantics (`fmod`/`mod` specifications). Although rewriting logic is not given as an institution [10], a so-called specification frame (collapsing signatures and sentences into theories) would be sufficient for our purposes.

However, after a closer look at Maude and rewriting logic, we found out that de facto, the logic underlying Maude differs from the rewriting logic as defined in [20]. The reasons are:

1. In Maude, labels of rewrites cannot (and need not) be translated along signature morphisms. This means that *e.g. Maude views do not lead to theory morphisms in rewriting logic!*

2. Although labels of rewrites are used in traces of counterexamples, they play a subsidiary role, because they cannot be used in the linear temporal logic of the Maude model checker.

Specially the first reason completely rules out a rewriting logic-based integration of Maude into HETS: if a view between two modules is specified, HETS definitely needs a theory morphism underlying the view.[2] However, the Maude user does not need to provide the action of the signature morphism on labeled rewrites, and generally, there is more than one possibility to specify this action.

The conclusion is that, for the time being, the most appropriate logic to use for Maude is preordered algebra [16]. In this logic, rewrites are neither labeled nor distinguished, only their existence is important. This implies that Maude views lead to theory morphisms in the institution of preordered algebras. Moreover, this setting also is in accordance with the above observation that in Maude rewrite labels are not first-class citizens, but are mere names of sentences that are convenient for decorating tool output (e.g. traces of the model checker). Labels of sentences play a similar role in HETS, which perfectly fits here.

Actually, the switch from rewriting logic to preordered algebras has effects on the consequence relation, contrary to what is said in [20]. Consider the following Maude theory:

```
th A is
 sorts S T .
 op a : -> S .
 eq X:S = a .
 ops h k : S -> T .
 rl [r] : a => a .
 rl [s] : h(a) => k(a) .
endfth
```

This logically implies $h(x) \Rightarrow k(x)$ in preordered algebra, but not in rewriting logic, since in the latter logic it is easy to construct models in which the naturality condition $r; k(r) = h(r); s$ fails to hold.

Thus, we will work with preordered algebra semantics for Maude. We will define an institution, that we will denote $Maude^{pre}$, which can be, like in the case of Maude's logic, parametric over the underlying equational logic. Following the Maude implementation, we have used membership equational logic [21]. Notice that the resulting institution $Maude^{pre}$ is very similar to the one defined in the context of CafeOBJ [16, 10] for preordered algebra (the differences are mainly given by the discussion about operation profiles below, but this is only a matter of representation). This allows us to make use of some results without giving detailed proofs.

Signatures of $Maude^{pre}$ are tuples $(K, F, kind : (S, \leq) \to K)$, where K is a set of *kinds*, *kind* is a function assigning a kind to each *sort* in the poset $(S, \leq)$, and F is a set of function symbols of the form $F = \{F_{k_1 \ldots k_n \to k} \mid k_i, k \in K\} \cup \{F_{s_1 \ldots s_n \to s} \mid s_i, s \in S\}$ such that if $f \in F_{s_1 \ldots s_n \to s}$, there is a symbol $f \in F_{kind(s_1) \ldots kind(s_n) \to kind(s)}$. Notice that there is actually no essential difference between our putting operation profiles on sorts into the signatures and Meseguer's original formulation putting them into the sentences.

Given two signatures $\Sigma_i = (K_i, F_i, kind_i)$, $i \in \{1, 2\}$, a signature morphism $\phi : \Sigma_1 \to \Sigma_2$ consists of a function $\phi^{kind} : K_1 \to K_2$ which preserves $\leq_1$, a function between the sorts $\phi^{sort} : S_1 \to S_2$ such that $\phi^{sort}; kind_2 = kind_1; \phi^{kind}$ and the subsorts are preserved, and a function $\phi^{op} : F_1 \to F_2$ which maps operation symbols compatibly with the types. Moreover, the overloading of symbol names must be preserved, i.e. the name of $\phi^{op}(\sigma)$ must be the same both when mapping the operation symbol σ on sorts and on kinds. With composition defined component-wise, we get the category of signatures.

[2] If the Maude designers would let (and force) users to specify the action of signature morphisms on rewrite labels, it would not be difficult to switch the HETS integration of Maude to being based on rewriting logic. In that case the labels would be taken into account, while the sentences would remain unchanged.

For a signature Σ, a model M interprets each kind k as a preorder $(M_k, \leq)$, each sort s as a subset M_s of $M_{kind(s)}$ that is equipped with the induced preorder, with M_s a subset of $M_{s'}$ if $s < s'$, and each operation symbol $f \in F_{k_1 \ldots k_n, k}$ as a function $M_f : M_{k_1} \times \ldots \times M_{k_n} \to M_k$ which has to be monotonic and such that for each function symbol f on sorts, its interpretation must be a restriction of the interpretation of the corresponding function on kinds. For two Σ-models A and B, a homomorphism of models is a family $\{h_k : A_k \to B_k\}_{k \in K}$ of preorder-preserving functions which is also an algebra homomorphism and such that $h_{kind(s)}(A_s) \subseteq B_s$ for each sort s.

The sentences of a signature Σ are Horn clauses built with three types of atoms: equational atoms $t = t'$, membership atoms $t : s$, and rewrite atoms $t \Rightarrow t'$, where t, t' are Σ-terms and s is a sort in S.[3] Given a Σ-model M, an equational atom $t = t'$ holds in M if $M_t = M_{t'}$, a membership atom $t : s$ holds when M_t is an element of M_s, and a rewrite atom $t \Rightarrow t'$ holds when $M_t \leq M_{t'}$. The satisfaction of sentences extends the satisfaction of atoms in the obvious way.

4.2 CASL

CASL, the Common Algebraic Specification Language [3, 29], has been designed by CoFI, the international *Common Framework Initiative for algebraic specification and development*. Its underlying logic combines first-order logic and induction (the latter is expressed using so-called sort generation constraints, which express term-generatedness of a part of a model; this is needed for the specification of the usual inductive datatypes) with subsorts and partial functions. The institution underlying CASL is introduced in two steps: first, many-sorted partial first-order logic with sort generation constraints and equality ($PCFOL^=$) is introduced, and then, subsorted partial first-order logic with sort generation constraints and equality ($SubPCFOL^=$) is described in terms of $PCFOL^=$ [23]. Basically this institution is composed of:

- A *subsorted signature* $\Sigma = (S, TF, PF, P, \leq_S)$, where S is a set of sorts, TF and PF are two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of total function symbols and partial function symbols, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each $(w, s) \in S^* \times S$, $P = (P_w)_{w \in S^*}$ a family of predicates, and $\leq_S$ is a reflexive and transitive subsort relation on the set S. Given two signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, a signature morphism $\sigma : \Sigma \to \Sigma'$ consists of:
  - a map $\sigma^S : S \to S'$ preserving the subsort relation,
  - a map $\sigma^F_{w,s} : TF_{w,s} \cup PF_{w,s} \to TF'_{\sigma^{S^*}(w), \sigma^S(s)} \cup PF'_{\sigma^{S^*}(w), \sigma^S(s)}$ preserving totality, for each $w \in S^*, s \in S$, and
  - a map $\sigma^P_w : P_w \to P'_{\sigma^{S^*}(w)}$ for each $w \in S^*$.

  Identities and composition are defined in the obvious way.

  With each subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ we associate a many-sorted signature $\hat{\Sigma}$, which is the extension of the underlying many-sorted signature (S, TF, PF, P) with:
  - a total *injection* function symbol $inj : s \to s'$, for each pair of sorts $s \leq_S s'$,
  - a partial *projection* function symbol $pr : s' \to ? s$, for each pair of sorts $s \leq_S s'$, and
  - a unary *membership* predicate symbol $\in^s : s'$, for each pair of sorts $s \leq_S s'$.

  Signature morphisms $\sigma : \Sigma \to \Sigma'$ are extended to signature morphisms $\hat{\sigma} : \hat{\Sigma} \to \hat{\Sigma}'$ by just mapping the injections, projections, and memberships in $\hat{\Sigma}$ to the corresponding injections, projections, and memberships in $\hat{\Sigma}'$.

  For a subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$, we define *overloading relations* (also called *monotonicity orderings*), $\sim_F$ and $\sim_P$, for function and predicate symbols, respectively:

  Let $f : w_1 \to s_1, f : w_2 \to s_2 \in TF \cup PF$, then $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ iff there exist $w \in S^*$ with $w \leq_{S^*} w_1$ and $w \leq_{S^*} w_2$ and $s \in S$ with $s_1 \leq_S s$ and $s_2 \leq_S s$.

  Let $p : w_1, p : w_2 \in P$, then $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ with $w \leq_{S^*} w_1$ and $w \leq_{S^*} w_2$.

[3] Note that this is slightly more general than Maude's version, because rewrite conditions are allowed in equations and membership axioms.

- A set of *subsorted* Σ-*sentences*, that correspond to ordinary $\hat{\Sigma}$-many-sorted sentences, that is, closed many-sorted first-order $\hat{\Sigma}$-formulas or sort generation constraints over Σ. Sentence translation along a subsorted signature morphism σ is just sentence translation along the many-sorted signature morphism $\hat{\sigma}$.
- *Subsorted* Σ-*models* M are ordinary many-sorted $\hat{\Sigma}$-models, which are composed of:
  - a non-empty carrier set M_s for each sort $s \in S$,
  - a partial function f_M from M_w to M_s for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, $w \in S^*$, $s \in S$, the function being total if $f \in TF_{w,s}$, and
  - a predicate $p_M \subseteq M_w$ for each predicate symbol $p \in P_w$, $w \in S^*$.

  satisfying the following set of axioms $\hat{J}(\Sigma)$:
  - $inj_{(s,s)}(x) \stackrel{e}{=} x$ (identity),
    where $\stackrel{e}{=}$ stands for existential equation.
  - $inj_{(s,s')}(x) \stackrel{e}{=} inj_{(s,s')}(x) \implies x \stackrel{e}{=} y$ for $s \leq_S s'$ (embedding-injectivity),
  - $inj_{(s',s'')}(inj_{s,s'}(x)) \stackrel{e}{=} inj_{(s,s'')}(x)$ for $s \leq_S s' \leq_S s''$ (transitivity),
  - $pr_{(s',s)}(inj_{(s,s')}(x)) \stackrel{e}{=} x$ for $s \leq_S s'$ (projection),
  - $pr_{(s',s)}(x) \stackrel{e}{=} pr_{(s',s)}(y) \implies z \stackrel{e}{=} y$ for $s \leq_S s'$ (projection-injectivity),
  - $\in^s_{s'}(x) \iff pr_{(s',s)}(x)$ for $s \leq_S s'$ (membership),
  - $inj_{(s',s)}(f_{w',s'}(inj_{s_1,s'_1}(x_1), \ldots, inj_{s_n,s'_n}(x_n))) = inj_{(s'',s)}(f_{w'',s''}(inj_{(s_1,s''_1)}(x_1), \ldots, inj_{(s_n,s''_n)}(x_n)))$ for $f_{w',s'} \sim_F f_{w'',s''}$, where $w \leq w', w''$, $s', s'' \leq s$, $w = s_1, \ldots, s_n$, $w' = s'_1, \ldots, s'_n$, and $w'' = s''_1, \ldots, s''_n$ (function-monotonicity), and
  - $p_{w'}(inj_{(s_1,s'_1)}(x_1), \ldots, inj_{(s_n,s'_n)}(x_n)) \iff p_{w''}(inj_{(s_1,s''_1)}(x_1), \ldots, inj_{(s_n,s''_n)}(x_n))$ for $p_{w'} \sim_P p_{w''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, and $w'' = s''_1 \ldots s''_n$ (predicate-monotonicity).
- *Satisfaction* and the *satisfaction condition* are inherited from the many-sorted institution. Roughly speaking, a formula φ is satisfied in a model M iff it is satisfied w.r.t. all variable valuations into M.

In contrast to Maude, CASL's subsort relations may be interpreted by arbitrary injections $inj_{s,t}$, not only by subsets. We refer to [29] for details. We will only need the Horn clause fragment of first-order logic. For freeness, we will also need sort generation constraints, as well as the *second-order* extension of CASL with quantification over predicates; we show the details in Section 5.2.

4.3 Encoding Maude into CASL

We now present an encoding of Maude into CASL, which is formalized as an institution comorphism. The idea of the encoding of $Maude^{pre}$ in CASL is that we represent rewriting as a binary predicate and we axiomatize it as a preorder compatible with operations.

Every Maude signature $\Sigma = (K, F, kind : (S, \leq) \to K)$ is translated to the CASL theory $\Phi(\Sigma) = ((S', \leq', F, P), E)$, where S' is the disjoint union of K and S, $\leq'$ extends the relation $\leq$ on sorts with pairs $(s, kind(s))$, for each $s \in S$, $rew \in P_{s,s}$ for any $s \in S'$ is a binary predicate and E contains axioms stating that for any kind k, $rew \in P_{k,k}$ is a preorder compatible with the operations. The latter means that for any $f \in F_{s_1..s_n,s}$ and any x_i, y_i of sort $s_i \in S'$, $i = 1, .., n$, if $rew(x_i, y_i)$ holds, then $rew(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n))$ also holds.

Let Σ_i, $i = 1, 2$ be two Maude signatures and let $\varphi : \Sigma_1 \to \Sigma_2$ be a Maude signature morphism. Then its translation $\Phi(\varphi) : \Phi(\Sigma_1) \to \Phi(\Sigma_2)$ denoted ϕ, is defined as follows:

- for each $s \in S$, $\phi(s) := \varphi^{sort}(s)$ and for each $k \in K$, $\phi(k) := \varphi^{kind}(k)$.
- the subsort preservation condition of ϕ follows from the similar condition for φ.
- for each operation symbol σ, $\phi(\sigma) := \varphi^{op}(\sigma)$.
- *rew* is mapped identically.

The sentence translation map for each signature is obtained in two steps. While the equational atoms are translated as themselves, membership atoms $t : s$ are translated to CASL memberships t *in* s and rewrite atoms of the form $t \Rightarrow t'$ are translated as $rew(t, t')$. Then, any sentence of Maude of the form $(\forall x_i : k_i)H \implies C$, where H is a conjunction of Maude atoms and C is an atom is translated as $(\forall x_i : k_i)H' \implies C'$, where H' and C' are obtained by mapping all the Maude atoms as described before.

Given a Maude signature Σ, a model M' of its translated theory (Σ', E) is reduced to a Σ-model denoted M where:

- for each kind k, define $M_k = M'_k$ and the preorder relation on M_k is rew;
- for each sort s, define M_s to be the image of M'_s under the injection $inj_{s,kind(s)}$ generated by the subsort relation;
- for each f on kinds, let $M_f(x_1, .., x_n) = M'_f(x_1, .., x_n)$ and for each f on sorts of result sort s, let $M_f(x_1, .., x_n) = inj_{s,kind(s)}(M'_f(x_1, .., x_n))$. M_f is monotone because axioms ensure that M'_f is compatible with rew.

The reduct of model homomorphisms is the expected one. Let Σ be a Maude signature, M', N' be two $\Phi(\Sigma)$-models (in CASL) and let $h' : M' \to N'$ be a model homomorphism. Let us denote $M = \beta_\Sigma(M')$, $N = \beta_\Sigma(N')$ and let us define $h : M \to N$ as follows: for any kind k of Σ, $h_k := h'_k$ (this is correct because the domain and the codomain match, by definition of M and N). We need to show that h is indeed a Maude model homomorphism. For this, we need to show three things:

1. h_k is preorder preserving for any kind k.

   Assume $x \leq^M_k y$. By definition, the preorder on M_k is the one given by rew, so this means $M_{rew}(x, y)$ holds. By the homomorphism condition for h' we have $N_{rew}(h'(x), h'(y))$ holds, which means by definition of the preorder on N that $h'(x) \leq^N_k h'(y)$.

2. h is an algebra homomorphism.

   This follows directly from the definition of M_σ where σ is an operation symbol and from the homomorphism condition for operation symbols for h'.

3. for any sort s, $h_{k_s}(M_s) \subseteq N_s$, where k_s is the kind of s, by a slight notational abuse.

   By definition, $M_s = inj_{s,k_s}(M'_s)$. By the homomorphism condition for inj_{s,k_s}, which is an explicit operation symbol in CASL, we have that $h_{k_s}(M_s) = h_{k_s}(inj_{s,k_s}(M'_s)) = inj_{s,k_s}(h_s(M'_s))$. Since $h_s(M'_s) \subseteq N'_s$ by definition, we have that $inj_{s,k_s}(h_s(M'_s)) \subseteq inj_{s,k_s}(N'_s)$, which by definition is N_s.

5 Building development graphs

We describe in this section how Maude structuring mechanisms described in Section 2 are translated into development graphs. Then, we explain how these development graphs are normalized to deal with freeness constraints.

5.1 Creating the development graph

We describe here how Maude modules, theories, and views are translated into development graphs, illustrating it with an example.

5.1.1 Modules

Each Maude module generates two nodes in the development graph. The first one contains the theory equipped with the usual loose semantics. The second one, linked to the first one with a free definition link (whose signature morphism is detailed in Section 5.2), contains the same signature but no local axioms and stands for the free models of the theory. Note that Maude theories only generate one node, since their initial semantics is not used by Maude specifications.

The model class of parameterized modules consists of free extensions of the models of their parameters, that are persistent on sorts, but not on kinds. This notion of freeness has been studied in [4] under assumptions like existence of top sorts for kinds and sorted variables in formulas; our results hold under similar hypotheses. We use non-persistent free links to link these modules with their corresponding theories.

5.1.2 Module expressions

Maude module expressions allow to combine and modify the information contained in Maude modules:

- When the module expression is a simple identifier the development graph remains unchanged.
- The summation of the module expressions ME_1 and ME_2 generates a new node in the development graph ($ME_1 + ME_2$) with the union of the information in both summands. A definition link is also created between the original expressions and the resulting one.
- The renaming expression $ME*(R)$ creates a morphism with the information given in R that will be used to label the link between the node standing for the module expression and the node importing it.

5.1.3 Importations

As explained above, each Maude module generates two nodes in the development graph; when importing a module, we will select between these nodes depending on the chosen importation mode:

- The `protecting` mode generates a non-persistent free link between the current node and the node standing for the free semantics of the included one. We use the same links for the parameters in parameterized modules.
- The `extending` mode generates a global link with the annotation PCons?, that stands for proof-theoretic conservativity and that can be checked with a special conservativity checker that is integrated into Hets.
- The `including` mode generates a global definition link between the current node and the node standing for the loose semantics of the included one.

5.1.4 Views

Maude views have a theory as source and either a module or a theory as target. All the sorts and the operators declared in the source theory have to be mapped to sorts and operators in the target.

As seen in Section 2.5.3, a particular case of mapping between operators is the mapping between terms, that has the general form `op` e `to term` t. Since this shortcut allows to map operators with different profiles, in these cases it generates an auxiliary node with the signature of the target specification extended by an extra operator of the appropriate arity; this node will be used as new target.

Views generate a theorem link between the theory and the module satisfying it. Note that an instantiation generates some implicit morphisms and modifies the ones stated in the views, see Section 2.5 for details:

- Sorts and labels are qualified by the parameter name in order to distinguish different labels/sorts with the same name defined in different theories. Thus, the mapping indicated by the view (more specifically, the source sorts) is modified depending on the name of the parameter.
- As explained in Section 2.5.4, parameterized modules can define parameterized sorts, that is, sorts that use the parameters as part of the sort name and hence they are modified by the mapping in the view. Moreover, when the target of a view is a theory the identifiers of these sorts are extended with the name of the view and the name of the new parameter. Thus, the sort morphism is extended with these new renamings.

5.1.5 An example of development graph

We illustrate how to build the development graph with an example. Consider the following Maude specification:

```
fmod M1 is
 sort S1 .
 op _+_ : S1 S1 -> S1 [comm] .
endfm
```

```
fmod M2 is
  sort S2 .
endfm
```

```
th T is
```

```
mod M3{X :: T} is
```

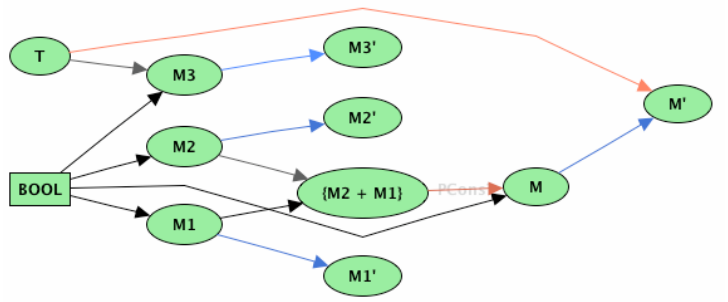

Figure 3: Development graph for Maude specifications

```
 sort S1 .
 op _._ : S1 S1 -> S1 .
 eq V1:S1 . V2:S1 = V2:S1 . V1:S1 [nonexec] .
endth

mod M is
 ex M1 + M2 * (sort S2 to S) .
endm
```

```
 sort S4 .
endm


view V from T to M is
 op _._ to _+_ .
endv
```

Hets builds the graph shown in Figure 3, where the following steps take place:

- First, the modules in the predefined Maude prelude generate their own graph. These nodes can be used by the ones of the current specification, like the `BOOL` node in the image (where the node has rectangular form because it hides part of its structure, like the modules it imports).
- Each module has generated a node with its name and another primed one that contains the initial model, while both of them are linked with a non-persistent free link (in blue in the illustration). Note that theory `T` did not generate this primed node.
- The summation expression has created a new node that includes the theories of `M1` and `M2`, importing the latter with a renaming; this new node, since it is imported in `extending` mode, uses a link with the PCons? annotation. This is the (unfortunately blurry) label in the link from `{M2 + M1}` to `M`.
- There is a theorem link (red link in the figure) between `T` and the free (here, initial) model of `M`. This link is labeled with the mapping defined in the view `V`, namely `op _._ to _+_ .`.
- The parameterized module `M3` includes the theory of its parameter with a renaming, that qualifies the sort. Note that these nodes are connected by means of a non-persistent free link.

It is straightforward to show:

Theorem 1 *The translation of Maude modules into development graphs is semantics-preserving.*

Once the development graph is built, we can apply the (logic independent) calculus rules that reduce global theorem links to local theorem links, which are in turn discharged by local theorem proving [25]. This can be used to prove Maude views, like "natural numbers are a total order." For example, we can automatically prove the view `V` above correct by using the first order automated provers SPASS or Vampire.

We show in the next section how we deal with the freeness constraints imposed by free definition links.

5.2 Normalization of free definition links

Maude uses initial and free semantics intensively. The semantics of freeness is, as mentioned, different from the one used in CASL in that the free extensions of models are required to be persistent only on sorts and new error elements can be added on the interpretation of kinds. As explained before, attempts to design the translation to CASL in such a way that Maude free links would be translated to usual free definition links in CASL have been unsuccessful, and thus we decided to use non-persistent free links. Hence, in order not to break the development graph calculus, we need a way to normalize them, by

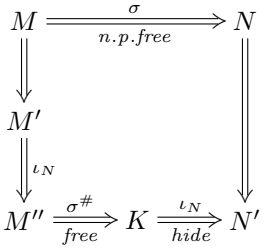

Figure 4: Normalization of Maude free links

replacing them with a semantically equivalent development graph in CASL. The main idea is to make a free extension persistent by duplicating parameter sorts appropriately, such that the parameter is always explicitly included in the free extension.

For any Maude signature Σ, let us define an extension $\Sigma^\# = (S^\#, \leq^\#, F^\#, P^\#)$ of the translation $\Phi(\Sigma)$ of Σ to CASL as follows:

- $S^\#$ adds the sorts of $\Phi(\Sigma)$ to the set $\{[s] \mid s \in \mathit{Sorts}(\Sigma)\}$;
- $\leq^\#$ extends the subsort relation $\leq$ with pairs $(s, [s])$ for each sort s and $([s], [s'])$ for any sorts $s \leq s'$;
- $F^\#$ adds the function symbols $\{f : [w] \to [s]\}$ for all function symbols on sorts $f : w \to s$;[4] and
- $P^\#$ adds the predicate symbol *rew* on all new sorts.

Now, we consider a Maude non-persistent free definition link and let $\sigma : \Sigma \to \Sigma'$ be the morphism labeling it.[5] We define a CASL signature morphism $\sigma^\# : \Phi(\Sigma) \to \Sigma'^\#$: on sorts, $\sigma^\#(s) := \sigma^{sort}(s)$ and $\sigma^\#([s]) := [\sigma^{sort}(s)]$; on operation symbols, we can define $\sigma^\#(f) := \sigma^{op}(f)$ and this is correct because the operation symbols were introduced in $\Sigma'^\#$; *rew* is mapped identically.

The normalization of Maude freeness is then illustrated in Figure 4. Given a non-persistent free definition link $M \xRightarrow[n.p.free]{\sigma} N$, with $\sigma : \Sigma \to \Sigma_N$, we first take the translation of the nodes to CASL (nodes M' and N'), and create the node M'', an extension (the morphism ι is a renaming to make the signature distinct from M) of M' where the signature has been extended with sorts $[s]$ for each sort $s \in \Sigma_M$, such that $s \leq [s]$ and $[s] \leq [s']$ if $s \leq s'$; function symbols have been extended with $f : [w] \to [s]$ for each $f : w \to s \in \Sigma_M$; and new *rew* predicates have been added for these sorts. Then, we introduce a new node, K, labeled with $\Sigma_N^\#$, a free definition link from M'' to K labeled with $\sigma^\#$ and a hiding definition link from K to N' labeled with the inclusion ι_N.[6]

Notice that the models of N are Maude reducts of CASL models of K, reduced along the inclusion ι_N.

Now we show how to eliminate CASL free definition links in a logic-independent way. The idea is to use a transformation specific to the second-order extension of CASL to normalize freeness. The intuition behind this construction is that it mimics the quotient term algebra construction, that is, the free model is specified as the homomorphic image of an absolutely free model (i.e. term model).

We are going to make use of the following known facts [33]:

Fact 1 *Extensions of theories in Horn form admit free extensions of models.*

Fact 2 *Extensions of theories in Horn form are monomorphic.*[7]

Given a free definition link $M \xRightarrow[free]{\sigma} N$, with $\sigma : \Sigma \to \Sigma^N$ such that $Th(M)$ is in Horn form, replace it with $M \xRightarrow{incl} K \xRightarrow[hide]{incl} N'$, where N' has the same signature and axioms as N, *incl* denote inclusions and the node K is constructed as follows.

[4] $[s_1 \dots s_n]$ is defined to be $[s_1] \dots [s_n]$.

[5] In Maude, this would usually be an injective renaming.

[6] The arrows without labels in Figure 4 correspond to heterogeneous links from Maude to CASL.

[7] That is, if N is an extension of K under a morphism σ, then every K-model has a σ-expansion to an N-model that is unique up to isomorphism.

The signature Σ^K consists of the signature Σ^M disjointly united with a copy of Σ^M, denoted $\iota(\Sigma_M)$ which makes all function symbols total (let us denote $\iota(f)$ the corresponding symbol in this copy for each symbol f from the signature Σ^M) and augmented with new operations $h : \iota(s) \to ? s$, for any sort s of Σ^M and $make_s : s \to \iota(s)$, for any sort s of the source signature Σ of the morphism σ labelling the free definition link.

The axioms ψ^K of the node K consist of:

- sentences imposing the bijectivity of *make*;
- axiomatization of the sorts in $\iota(\Sigma_M)$ as free types with all operations as constructors, including *make* for the sorts in $\iota(\Sigma)$;
- homomorphism conditions for h:

  $$h(\iota(f)(x_1, \ldots, x_n)) = f(h(x_1), \ldots, h(x_n))$$

  and

  $$\iota(p)(t_1, \ldots, t_n) \Rightarrow p(h(t_1), \ldots, h(t_n))$$

- surjectivity of homomorphisms:

  $$\forall y : s.\exists x : \iota(s).h(x) \stackrel{e}{=} y$$

- a second-order formula saying that the kernel of h ($ker(h)$) is the least partial predicative congruence[8] satisfying $Th(M)$. This is done by quantifying over a predicate symbol for each sort for the binary relation and one predicate symbol for each relation symbol as follows:

  $$\forall \{P_s : \iota(s), \iota(s)\}_{s \in Sorts(\Sigma_M)}, \{P_{p:w} : \iota(w)\}_{p:w \in \Sigma_M}$$
  $$. \ symmetry \wedge transitivity \wedge congruence \wedge satThM \implies largerThanKerH$$

  where *symmetry* stands for

  $$\bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s).P_s(x, y) \implies P_s(y, x),$$

  transitivity stands for

  $$\bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s), z : \iota(s).P_s(x, y) \wedge P_s(y, z) \implies P_s(x, z),$$

  congruence is the conjunction of

  $$\textstyle\bigwedge_{f_{w \to s} \in \Sigma^M} \forall x_1 \ldots x_n : \iota(w), y_1 \ldots y_n : \iota(w) .$$
  $$D(\iota(f_{w,s})(\bar{x})) \wedge D(\iota(f_{w,s})(\bar{y})) \wedge P_w(\bar{x}, \bar{y}) \implies P_s(\iota(f_{w,s})(\bar{x}), \iota(f_{w,s})(\bar{y}))$$

  and

  $$\textstyle\bigwedge_{p_w \in \Sigma^M} \forall x_1 \ldots x_n : \iota(w), y_1 \ldots y_n : \iota(w) .$$
  $$D(\iota(f_{w,s})(\bar{x})) \wedge D(\iota(f_{w,s})(\bar{y})) \wedge P_w(\bar{x}, \bar{y}) \implies P_{p:w}(\bar{x}) \Leftrightarrow P_{p:w}(\bar{y})$$

  where D indicates definedness. *satThM* stands for

  $$Th(M)[\stackrel{e}{=} / P_s; p : w / P_{p:w}; D(t) / P_s(t, t); t = u / P_s(t, u) \vee (\neg P_s(t, t) \wedge \neg P_s(u, u))]$$

  where, for a set of formulas Ψ, $\Psi[sy_1/sy'_1; \ldots; sy_n/sy'_n]$ denotes the simultaneous substitution of sy'_i for sy_i in all formulas of Ψ (while possibly instantiating the meta-variables t and u). Finally *largerThanKerH* stands for

  $$\textstyle\bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s).h(x) \stackrel{e}{=} h(y) \implies P_s(x, y)$$
  $$\textstyle\bigwedge \wedge_{p_w \in \Sigma^M} \forall \bar{x} : \iota(w).\iota(p : w)(\bar{x}) \implies P_{p:w}(\bar{x})$$

[8] A *partial predicative congruence* consists of a symmetric and transitive binary relation for each sort and a relation of appropriate type for each predicate symbol.

Proposition 1 *The models of the nodes N and N' are the same.*

Proof. Let n be an N-model. To prove that n is also an N'-model, we need to show that it has a K-expansion.

Let us define the following Σ_K model, denoted k:

- on Σ_M, k coincides with n;
- on $\iota(\Sigma_M)$, the interpretation of sorts and function symbols is given by the free types axioms (i.e., sorts are interpreted as set of terms, operations $\iota(f)$ map terms $t_1, \ldots, t_n$ to the term $\iota(f)(t_1, \ldots, t_n)$). We define the interpretation of predicates after defining h;
- *make* assigns to each x the term $make(x)$;
- the homomorphism h is defined inductively as follows:
  - $h(make(x)) = x$, if $x \in n_s$ and $s \in Sorts(\Sigma)$;
  - $h(make(t)) = h(t)$, otherwise;
  - $h(\iota(f)(t_1, \ldots, t_n))$ is defined iff $f(h(t_1), \ldots, h(t_n))$ is defined in n and then $h(\iota(f)(t_1, \ldots, t_n)) = f(h(t_1), \ldots, h(t_n))$;
- for predicates in $\iota(\Sigma_M)$ we define $\iota(p)(t_1, \ldots, t_n)$ iff $p(h(t_1), \ldots, h(t_n))$.

Notice that the first three types of axioms of the node K hold by construction and also notice that $ker(h)$ satisfies $Th(M)$ because n is an M-model. The surjectivity of h and the minimality of $ker(h)$ are exactly the "no junk" and the "no confusion" properties of the free model n.

For the other inclusion, let n' be a model of N', n_0 be its Σ-reduct and k' a K-expansion of n'. Using the fact that the theory of M is in Horn form, we get an expansion of n_0 to a σ-free model n. We have seen that all free models are also models of N' and moreover we have seen that $ker(k_h)$ is the least predicative congruence satisfying $Th(M)$. The free types axioms of K fix the interpretation of $\iota(\Sigma_M)$ and therefore $ker(k'_h)$ and $ker(k_h)$ are both minimal on the same set, and must be the same. This and the surjectivity of k_h and k'_h allow us to define easily an isomorphism between n and n' and because n' is isomorphic with a free model it must be free as well.

□

6 An example: Reversing lists

The example we are going to present is a standard specification of lists with empty lists, concatenation, and reversal. We want to prove that by reversing a list twice we obtain the original list. Since Maude syntax does not support marking sentences of a theory as theorems, the methodology to state proof obligations in Maude would normally be to write a view (*PROVEIDEM* in Figure 5, left side) from a theory containing the theorem (*REVIDEM*) to the module with the axioms defining *reverse* (*LISTREV*).

The first advantage that the integration of Maude in Hets brings in is that we can use heterogeneous Casl structuring mechanisms and the *%implies* annotation to obtain the same development graph in a shorter way—see the right side of Figure 5, whose development graph is shown in Figure 6, where the blue link[9] stands for freeness and the red one[10] for proof obligations, and only the rightmost node has name, while all the others are intermediate nodes introduced by the Hets constructors (`free`, used to indicate that the specification is free, and `then`, used to import the previous specification without assuming anything about it) used to structure the specification.

For our example, the development calculus rules are applied as follows.[11] First, the whole graph is translated to Casl; during this step, Maude non-persistent free links are normalized. The next step is to normalize Casl free links, using the `Freeness` rule. We then apply the `Normal-Form` rule which introduces normal forms for the nodes with incoming hiding links (introduced at the previous step) and then the `Theorem-Hide-Shift` rule which moves the target of any theorem link targeting a node with incoming hiding links to the normal form of the latter. Finally, calling `Automatic` the development graph in Figure 7 is obtained, where the proof obligation has been delegated to the normal form node (the red node in the upper right corner).

[9] The leftmost link, if you are reading a black-and-white paper.
[10] The one starting in `PROVEIDEM`.
[11] All the rules listed below are accessible in the Edit/Proof menu.

```
fmod MYLIST is
 sorts Elt List .
 subsort Elt < List .
 op nil : -> List [ctor] .
 op __ : List List -> List
    [ctor assoc id: nil] .
endfm
fmod MYLISTREV is
 pr MYLIST .
 op reverse : List -> List .
 var L : List .
 var E : Elt .
 eq reverse(nil) = nil .
 eq reverse(E L) = reverse(L) E .
endfm
fth REVIDEM  is
 pr MYLIST .
 op reverse : List -> List .
 var L : List .
 eq reverse(reverse(L)) = L .
endfth
view PROVEIDEM from REVIDEM
               to MYLISTREV is
 sort List to List .
 op reverse to reverse .
endv
```

logic MAUDE
spec PROVEIDEM =
free
{*sorts Elt List* .
subsort Elt < *List* .
op nil : −> *List* [*ctor*] .
op __ : *List List* −> *List* [*ctor assoc id*: *nil*] .
}
then {*op reverse* : *List* −> *List* .
var L : *List* . *var E* : *Elt* .
eq reverse(*nil*) = *nil* .
eq reverse(*E L*) = *reverse*(*L*) *E* .
} **then %implies**
{*var L* : *List* .
eq reverse(*reverse*(*L*)) = *L* .
}

Figure 5: Lists with reverse, in Maude (left) and CASL (right) syntax.

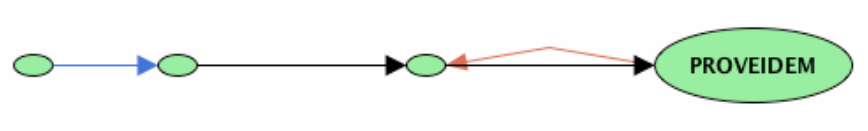

Figure 6: Development graph for the lists example

In this node, we now have a proof goal for a second-order theory. It can be discharged using the interactive theorem prover Isabelle/HOL [30]. We have set up a series of lemmas easing such proofs, and that can be adapted to other proofs over Maude specifications. First of all, normalization of freeness introduces sorts for the free model which are axiomatized to be the homomorphic image of a set of the absolutely free (i.e. term) model. A transfer lemma (that exploits surjectivity of the homomorphism) enables us to transfer any proof goal from the free model to the absolutely free model. Since the absolutely free model is term generated, we can use induction proofs here. For the case of datatypes with total constructors (like lists), we prove by induction that the homomorphism is total as well. Once these lemmas have been proved we prove the main theorem; to do that, two further lemmas on lists are proved by induction: (1) associativity of concatenation and (2) the reverse of a concatenation is the concatenation (in reverse order) of the reversed lists. This infrastructure then allows us to prove (again by induction) that $reverse(reverse(L)) = L$.

While proof goals in Horn clause form often can be proved by induction, other proof goals like the inequality of certain terms or extensionality of sets cannot. Here, we need to prove inequalities or equalities with more complex premises, and this calls for use of the special axiomatization of the kernel of the homomorphism. This axiomatization is rather complex, and we are currently setting up the infrastructure for easing such proofs in Isabelle/HOL.

6.1 An easier proof: Revisiting our initial example

Recalling the example in Section 5.1.5, we stated in a theory `T` that an operator `_._` over the sort `S1` fulfilling the equation

```
eq V1:S1 . V2:S2 = V2:S1 . V1:S2 [nonexec] .
```

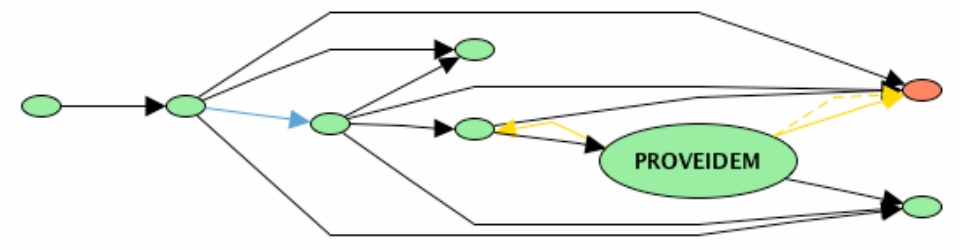

Figure 7: Development graph for the lists example after the transformation rules

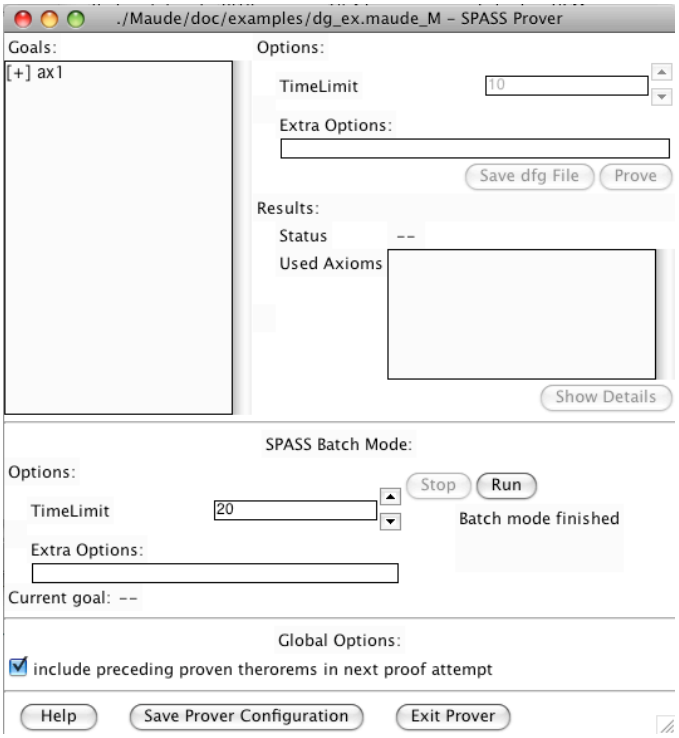

Figure 8: Axiom proved by SPASS

was required. We mapped this operator by means of a view to another one with syntax `_+_` and declared with the commutativity attribute `comm`, and thus we want to check that this view is correct. This easy proof can be automatically discarded by using provers such as SPASS just by transforming the development graph with the `Automatic` command, that "pushes" proof obligations to the appropriate nodes. We show in Figure 8 how it was proved (the `+` symbol indicates it was proved) by just using the `Run` button. This straightforward strategy can be applied to most Maude views (such as the ones from theories requiring different orders over the elements), making these proofs straightforward.

Of course, this proof can also be done in Isabelle. The code needed to prove it is:

```
theorem Ax1 :
"ALL (v1 :: kind_S1). ALL (v2 :: kind_S1). v1 +'' v2 = v2 +'' v1"
apply auto
apply (rule comm_Plus'_kind_S1)
done
```

where the theorem was introduced by HETS and `comm_Plus'_kind_S1` is the axiom automatically generated by the system for the commutative operator `_+_` in `M1`.

7 Implementation

We describe in this section how the integration described in the previous sections has been implemented. We have used Maude to parse Maude modules, taking advantage of the reflective capabilities of rewriting logic [7], while the rest of the system has been implemented in Haskell, the implementation language of HETS. Section 7.1 shows the abstract syntax used to represent Maude modules in Haskell, while Section 7.2 presents how these data structures are generated in Maude. Section 7.3 shows how Maude signatures, sentences, and morphisms are obtained, Section 7.4 explains how they are introduced into a development graph, and Section 7.5 describes the implementation of the comorphism between Maude and CASL. Finally, Section 7.6 outlines how the freeness constraints are implemented.

7.1 Abstract syntax

In this section we show how the abstract syntax for Maude specifications is defined in Haskell. This abstract syntax is based in the Maude grammar presented in [6, Chapter 24].

The main datatype of this abstract syntax is `Spec`, that distinguishes between the different specifications available in Maude: modules, theories, and views. Although both modules and theories contain the same information, their semantics are different and need different constructors:

```
data Spec = SpecMod Module
          | SpecTh Module
          | SpecView View
          deriving (Show, Read, Ord, Eq)
```

A `Module` is composed of the identifier of the module, a list of parameters, and a list of statements:

```
data Module = Module ModId [Parameter] [Statement]
            deriving (Show, Read, Ord, Eq)
```

while a `View` is composed of a module identifier, the source and target module expressions, and a list of renamings:

```
data View = View ModId ModExp ModExp [Renaming]
            deriving (Show, Read, Ord, Eq)
```

The `Parameter` type contains the identifier of the parameter, a sort (used as the parameter identifier), and its type (which is a module expression):

```
data Parameter = Parameter Sort ModExp
               deriving (Show, Read, Ord, Eq)
```

A `Statement` can be any of the Maude statements: importation, sort, subsort, and operator declarations, and equation, membership axiom, and rule statements:

```
data Statement = ImportStmnt Import
               | SortStmnt Sort
               | SubsortStmnt SubsortDecl
               | OpStmnt Operator
               | EqStmnt Equation
               | MbStmnt Membership
               | RlStmnt Rule
               deriving (Show, Read, Ord, Eq)
```

Importations consist of a module expression qualified by the type of import:

```
data Import = Including ModExp
            | Extending ModExp
            | Protecting ModExp
            deriving (Show, Read, Ord, Eq)
```

A subsort declaration keeps single relations between sorts, being the first one the subsort and the second one the supersort:

```
data SubsortDecl = Subsort Sort Sort
                 deriving (Show, Read, Ord, Eq)
```

Operator declarations are composed of the identifier of the operator, a list of types giving the arity of the operator, a type for its coarity, and a list of attributes:

```
data Operator = Op OpId [Type] Type [Attr]
              deriving (Show, Read, Ord, Eq)
```

Membership statements consist of a term, its sort, a list of conditions, and a list of statement attributes:

```
data Membership = Mb Term Sort [Condition] [StmntAttr]
                deriving (Show, Read, Ord, Eq)
```

Equations and rules share the same elements: the lefthand and righthand terms of the statement, a list of conditions, and a list of statement attributes:

```
data Equation = Eq Term Term [Condition] [StmntAttr]
              deriving (Show, Read, Ord, Eq)

data Rule = Rl Term Term [Condition] [StmntAttr]
          deriving (Show, Read, Ord, Eq)
```

We distinguish between the following module expressions:

- A single identifier:

  ```
  data ModExp = ModExp ModId
  ```

- A summation, that keeps the two module expressions involved:

  ```
                | SummationModExp ModExp ModExp
  ```

- A renaming, that contains the module expression renamed and the list of renamings:

  ```
                | RenamingModExp ModExp [Renaming]
  ```

- An instantiation, composed of the module instantiated and the list of view identifiers applied:

  ```
                | InstantiationModExp ModExp [ViewId]
                deriving (Show, Read, Ord, Eq)
  ```

The `Renaming` type distinguishes the different renamings available in Maude:

- Renaming of sorts, that indicates that the first sort identifier is changed to the second one:

  ```
  data Renaming = SortRenaming Sort Sort
  ```

- Renaming of labels, where the first label is renamed to the second one:

  ```
                  | LabelRenaming LabelId LabelId
  ```

- Renaming of operators, that can be of three kinds: renaming of operators without profile, with profile, or a map between terms, as explained in Section 2.5.3:

  ```
                  | OpRenaming1 OpId ToPartRenaming
                  | OpRenaming2 OpId [Type] Type ToPartRenaming
                  | TermMap Term Term
                  deriving (Show, Read, Ord, Eq)
  ```

where `ToPartRenaming` specifies the new operator identifier and the new attributes:

```
data ToPartRenaming = To OpId [Attr]
                    deriving (Show, Read, Ord, Eq)
```

The `Condition` type distinguishes between the different conditions available in Maude, namely equational conditions, membership conditions, matching conditions, and rewriting conditions:

```
data Condition = EqCond Term Term
               | MbCond Term Sort
               | MatchCond Term Term
               | RwCond Term Term
               deriving (Show, Read, Ord, Eq)
```

We define the type `Qid`, a synonym of `Token` that will be used for identifiers:

```
type Qid = Token
```

Terms are always represented in prefix notation. Notice that the case of an operator applied to a list of terms is slightly different to the Maude grammar because it also includes the type of the term. It will be used later in the implementation to rename operators whose profile has been specified:

```
data Term = Const Qid Type
          | Var Qid Type
          | Apply Qid [Term] Type
          deriving (Show, Read, Ord, Eq)
```

Finally, `Type` distinguishes between sorts and kinds:

```
data Type = TypeSort Sort
          | TypeKind Kind
          deriving (Show, Read, Ord, Eq)
```

7.2 Maude parsing

In this section we explain how the Maude specifications introduced in HETS are parsed in order to obtain a term following the abstract syntax described in the previous section. We are able to implement this parsing in Maude itself thanks to Maude's metalevel [6, Chapter 14], a module that allows the programmer to use Maude entities such as modules, equations, or rules as usual data by efficiently implementing the *reflective* capabilities of rewriting logic [7].

The function `haskellify` receives a module (the first parameter stands for the original module, while the second one contains the flattened one) and returns a list of quoted identifiers creating an object of type `Spec`, that can be read by Haskell since this data type derives the class `Read`:

```
  op haskellify : Module Module -> QidList .
  ceq haskellify(M, M') =
      'SpecMod ''( 'Module haskellifyHeader(H) ' '
      ''[ haskellifyImports(IL) comma(IL, SS)
          haskellifySorts(SS) comma(IL, SS, SSDS)
          haskellifySubsorts(SSDS) comma(IL, SS, SSDS, ODS)
          haskellifyOpDeclSet(M', ODS) comma(IL, SS, SSDS, ODS, MAS)
          haskellifyMembAxSet(M', MAS) comma(IL, SS, SSDS, ODS, MAS, EqS)
          haskellifyEqSet(M', EqS) ''] ''') '\n '@#$endHetsSpec$#@ '\n
    if fmod H is IL sorts SS . SSDS ODS MAS EqS endfm := M .
```

This function prints the keyword `SpecMod` and uses the `haskellify` auxiliary functions to print the different parts of the module. The functions `comma` introduce a comma whenever it is necessary. Since all the "haskellify" functions are very similar, we describe them by using `haskellifyImports` as example. This function traverses all the imports in the list and applies the auxiliary function `haskellifyImport` to each of them:

```
op haskellifyImports : ImportList -> QidList .
eq haskellifyImports(nil) = nil .
eq haskellifyImports(I IL) = 'ImportStmnt ' ''( haskellifyImport(I) '')
                             comma(IL) haskellifyImports(IL) .
```

This auxiliary function distinguishes between the importation modes, using the appropriate keyword for each of them:

```
op haskellifyImport : Import -> QidList .
eq haskellifyImport(protecting ME .) = 'Protecting haskellifyME(ME) .
eq haskellifyImport(including ME .) = 'Including haskellifyME(ME) .
eq haskellifyImport(extending ME .) = 'Extending haskellifyME(ME) .
```

where `haskellifyME` is in charge of printing the module expression. When it is just an identifier, it prints it preceded by the word `ModId`:

```
op haskellifyME : ModuleExpression -> QidList .
eq haskellifyME(Q) = ' ''( 'ModExp ' ''( 'ModId qid2token(Q) '') '') ' .
```

The summation module expression recursively prints the summands, and uses the keyword `SummationModExp` to indicate the type of module expression:

```
eq haskellifyME(ME + ME') = ' ''( 'SummationModExp haskellifyME(ME)
                            haskellifyME(ME')  '') ' .
```

To print a renaming we recursively apply `haskellifyME` for the inner module expression and then we use the auxiliary function `haskellifyMaps` to print the renamings. In this case we use the constant `no-module` as argument because it will only be used when parsing mappings from views, since it may be needed to parse terms:

```
eq haskellifyME(ME * (RNMS)) = ' ''( 'RenamingModExp haskellifyME(ME)
                                 ''[ haskellifyMaps(no-module, no-module, RNMS) '']  '') ' .
```

Finally, an instantiation is printed by using an auxiliary function `haskellifyPL` in charge of the parameters:

```
eq haskellifyME(ME {PL}) = ' ''( 'InstantiationModExp haskellifyME(ME)
                           ''[ haskellifyPL(PL)  '']  '') ' .
```

7.3 Logic

Once the Maude modules have been translated into their abstract syntax, we must implement the type classes `Language` and `Logic` provided by HETS, that define the types needed to represent each logic as an institution and the comorphisms between them. In the following section we describe the most important of these datatypes, while more details about them can be found in [19].

7.3.1 Signature

The signature defines types for the sorts, kinds, subsort relations, operators, and sentences, and a map relating each sort to its corresponding kind:

```
type SortSet = SymbolSet
type KindSet = SymbolSet
type SubsortRel = SymbolRel
type OpDecl = (Set Symbol, [Attr])
type OpDeclSet = Set OpDecl
type OpMap = Map Qid OpDeclSet
type Sentences = Set Sentence
type KindRel = Map Symbol Symbol
```

These types are used to define `Sign`, that stands for Maude signatures:

```
data Sign = Sign {
        sorts :: SortSet,
        kinds :: KindSet,
        subsorts :: SubsortRel,
        ops :: OpMap,
        sentences :: Sentences,
        kindRel :: KindRel
        } deriving (Show, Ord, Eq)
```

The function `fromSpec` extracts the signature from a module:

```
fromSpec :: Module -> Sign
```

This type class also provides functions to join and intersect signatures:

```
union :: Sign -> Sign -> Sign
intersection :: Sign -> Sign -> Sign
```

7.3.2 Sentences

The type for sentences distinguishes between membership axioms, equations, and rules:

```
data Sentence = Membership Membership
              | Equation Equation
              | Rule Rule
              deriving (Show, Read, Ord, Eq)
```

The sentences can be extracted from a module with `fromSpec`, that uses an auxiliary function to obtain them from the statements:

```
fromSpec :: Module -> [Sentence]
fromSpec (Module _ _ stmts) = fromStatements stmts
```

This auxiliary function generates the sentences inferred from the operator attributes such as `assoc` or `comm` with `fromOperator`, while the rest of statements are translated identically:

```
fromStatements :: [Statement] -> [Sentence]
fromStatements stmts = let
   convert stmt = case stmt of
       OpStmnt op -> fromOperator op
       MbStmnt mb -> [Membership mb]
       EqStmnt eq -> [Equation eq]
       RlStmnt rl -> [Rule rl]
       _ -> []
   in concatMap convert stmts
```

`fromOperator` traverses the attributes and generates the appropriate sentence for each of them:

```
fromOperator :: Operator -> [Sentence]
fromOperator (Op op dom cod attrs) = let
   name = getName op
   first = head dom
   second = head $ tail dom
   convert attr = case attr of
       Comm -> commEq name first second cod
       Assoc -> assocEq name first second cod
       Idem -> idemEq name first cod
       Id term -> identityEq name first term cod
       LeftId term -> leftIdEq name first term cod
       RightId term -> rightIdEq name first term cod
       _ -> []
   in concatMap convert attrs
```

We show how the sentence for commutativity is created; the rest of them are produced analogously. The function `commEq` generates two variables of the given sort, which are provided as arguments to the operator in different order, thus creating two terms that are made equal by means of an equation:

```
commEq :: Qid -> Type -> Type -> Type -> [Sentence]
commEq op ar1 ar2 co = [Equation $ Eq t1 t2 [] []]
   where v1 = mkVar "v1" $ type2Kind ar1
         v2 = mkVar "v2" $ type2Kind ar2
         t1 = Apply op [v1, v2] $ type2Kind co
         t2 = Apply op [v2, v1] $ type2Kind co
```

7.3.3 Morphisms

To define the Maude morphisms we first declare maps for sorts, kinds (induced from the previous maps), operators, and labels as map of symbols, a generic identifier for the different data that appears in Maude modules:

```
type SortMap = SymbolMap
type KindMap = SymbolMap
type OpMap = SymbolMap
type LabelMap = SymbolMap
```

We create our morphisms by adding to these data types the source and target signatures:

```
data Morphism = Morphism {
        source :: Sign,
        target :: Sign,
        sortMap :: SortMap,
        kindMap :: KindMap,
        opMap :: OpMap,
        labelMap :: LabelMap
    } deriving (Show, Ord, Eq)
```

Morphisms can be obtained from a list of renamings with `fromSignRenamings`:

```
fromSignRenamings :: Sign -> [Renaming] -> Morphism
```

The type class also provides functions to join and compose morphisms:

```
union :: Morphism -> Morphism -> Morphism
compose :: Morphism -> Morphism -> Result Morphism
```

and to generate an inclusion morphism, that is, an identity morphism between two signatures:

```
inclusion :: Sign -> Sign -> Morphism
inclusion src tgt = Morphism {
        source = src,
        target = tgt,
        sortMap  = Map.empty,
        kindMap  = Map.empty,
        opMap    = Map.empty,
        labelMap = Map.empty
   }
```

7.4 Development graph

We describe in this section the main functions used to draw the development graph for Maude specifications. The most important function is `anaMaudeFile`, that receives a record of all the options received from the command line (of type `HetcatsOpts`) and the path of the Maude file to be parsed and returns a pair with the library name and its environment. This environment contains two development graphs, the first one containing the modules used in the Maude prelude and another one with the user specification:

```
anaMaudeFile :: HetcatsOpts -> FilePath -> IO (Maybe (LibName, LibEnv))
anaMaudeFile _ file = do
    (dg1, dg2) <- directMaudeParsing file
    let ln = emptyLibName file
        lib1 = Map.singleton preludeLib $
                 computeDGraphTheories Map.empty $ markFree Map.empty $
```

```
                markHiding Map.empty dg1
        lib2 = Map.insert ln
                (computeDGraphTheories lib1 $ markFree lib1 $
                markHiding lib1 dg2) lib1
    return $ Just (ln, lib2)
```

This environment is computed with the function `directMaudeParsing`, that receives the path introduced by the user and returns a pair of development graphs. These graphs are obtained with the function `maude2DG`, that receives the predefined specifications (obtained with `predefinedSpecs`) and the user defined specifications (obtained with `traverseSpecs`):

```
directMaudeParsing :: FilePath -> IO (DGraph, DGraph)
directMaudeParsing fp = do
  ml <- getEnvDef "MAUDE_LIB" ""
  if null ml then error "environment variable MAUDE_LIB is not set" else do
    ns <- parse fp
    let ns' = either (const []) id ns
    (hIn, hOut, hErr, procH) <- runMaude
    exitCode <- getProcessExitCode procH
    case exitCode of
      Nothing -> do
              hPutStrLn hIn $ "load " ++ fp
              hFlush hIn
              hPutStrLn hIn "."
              hFlush hIn
              hPutStrLn hIn "in Maude/hets.prj"
              psps <- predefinedSpecs hIn hOut
              sps <- traverseSpecs hIn hOut ns'
              (ok, errs) <- getErrors hErr
              if ok
                  then do
                        hClose hIn
                        hClose hOut
                        hClose hErr
                        return $ maude2DG psps sps
                  else do
                        hClose hIn
                        hClose hOut
                        hClose hErr
                        error errs
      Just ExitSuccess -> error "maude terminated immediately"
      Just (ExitFailure i) -> error $ "calling maude failed with exitCode: " ++ show i
```

The function `maude2DG` first computes the data structures associated to the predefined specifications and then uses them to compute the development graph related to the specifications introduced by the user. These data structures are computed with `insertSpecs`:

```
maude2DG :: [Spec] -> [Spec] -> (DGraph, DGraph)
maude2DG psps sps = (dg1, dg2)
   where (_, tim, vm, tks, dg1) = insertSpecs psps emptyDG Map.empty
                                              Map.empty Map.empty [] emptyDG
         (_,_, _, _, dg2) = insertSpecs sps dg1 tim Map.empty vm tks emptyDG
```

Before describing this function, we briefly explain the data structures used during the generation of the development graph:

- The type `ParamSort` defines a pair with a symbol representing a sort and a list of tokens indicating the parameters present in the sort, so for example the sort `List{X, Y}` generates the pair `(List{X, Y}, [X,Y])`:

  ```
  type ParamSort = (Symbol, [Token])
  ```

- The information of each node introduced in the development graph is stored in the tuple `ProcInfo`, that contains the following information:

  - The identifier of the node.
  - The signature of the node.
  - A list of symbols standing for the sorts that are not instantiated.
  - A list of triples with information about the parameters of the specification, namely the name of the parameter, the name of the theory, and the list of not instantiated sorts from this theory.
  - A list with information about the parameterized sorts.

```
type ProcInfo = (Node, Sign, Symbols, [(Token, Token, Symbols)], [ParamSort])
```

- Each `ProcInfo` tuple is associated to its corresponding module expression in the `TokenInfoMap` map:

```
type TokenInfoMap = Map.Map Token ProcInfo
```

- When a module expression is parsed a `ModExpProc` tuple is returned, containing the following information:
  - The identifier of the module expression.
  - The `TokenInfoMap` structure updated with the data in the module expression.
  - The morphism associated to the module expression.
  - The list of sorts parameterized in this module expression.
  - The development graph thus far.

```
type ModExpProc = (Token, TokenInfoMap, Morphism, [ParamSort], DGraph)
```

- When parsing a list of importation statements we return a `ParamInfo` tuple, containing:
  - The list of parameter information: the name of the parameter, the name of the theory, and the sorts that are not instantiated.
  - The updated `TokenInfoMap` map.
  - The list of morphisms associated with each parameter.
  - The updated development graph.

```
type ParamInfo = ([(Token, Token, Symbols)], TokenInfoMap, [Morphism], DGraph)
```

- Data about views is kept in a separated way from data about theories and modules. The `ViewMap` map associates to each view identifier a tuple with:
  - The identifier of the target node of the view.
  - The morphism generated by the view.
  - The list of renamings that generated the morphism.
  - A Boolean value indicating whether the target is a theory (`True`) or a module (`False`).

```
type ViewMap = Map.Map Token (Node, Token, Morphism, [Renaming], Bool)
```

- Finally, we describe the tuple `InsSpecRes`, used to return the data structures updated when a specification or a view is introduced in the development graph. It contains:
  - Two values of type `TokenInfoMap`. The first one includes all the information related to the specification, including the one from the predefined modules, while the second one only contains information related to the current development graph.
  - The updated `ViewMap`.
  - A list of tokens indicating the theories introduced thus far.

- The new development graph.

```
type InsSpecRes = (TokenInfoMap, TokenInfoMap, ViewMap, [Token], DGraph)
```

The function `insertSpecs` traverses the specifications updating the data structures and the development graph with `insertSpec`:

```
insertSpecs :: [Spec] -> DGraph -> TokenInfoMap -> TokenInfoMap -> ViewMap -> [Token] -> DGraph
               -> InsSpecRes
insertSpecs [] _ ptim tim vm tks dg = (ptim, tim, vm, tks, dg)
insertSpecs (s : ss) pdg ptim tim vm ths dg = insertSpecs ss pdg ptim' tim' vm' ths' dg'
              where (ptim', tim', vm', ths', dg') = insertSpec s pdg ptim tim vm ths dg
```

The behavior of `insertSpec` is different for each type of Maude specification. When the introduced specification is a module, the following actions are performed:

- The parameters are parsed:
  - The list of parameter declarations is obtained with the auxiliary function `getParams`.
  - These declarations are processed with `processParameters`, that returns a tuple of type `ParamInfo` described above.
  - Given the parameters names, we traverse the list of sorts to check whether the module defines parameterized sorts with `getSortsParameterizedBy`.
  - The links between the theories in the parameters and the current module are created with `createEdgesParams`.
- The importations are handled:
  - The importation statements are obtained with `getImportsSorts`. Although this function also returns the sorts declared in the module, in this case they are not needed and its value is ignored.
  - These importations are handled by `processImports`, that returns a list containing the information of each parameter.
  - The definition links generated by the imports are created with `createEdgesImports`.
- The final signature is obtained with `sign_union_morphs` by merging the signature in the current module with the ones obtained from the morphisms from the parameters and the imports.

```
insertSpec :: Spec -> DGraph -> TokenInfoMap -> TokenInfoMap -> ViewMap -> [Token] -> DGraph
              -> InsSpecRes
insertSpec (SpecMod sp_mod) pdg ptim tim vm ths dg = (ptimUp, tim5, vm, ths, dg6)
      where ps = getParams sp_mod
            (il, _) = getImportsSorts sp_mod
            up = incPredImps il pdg (ptim, tim, dg)
            (ptimUp, timUp, dgUp) = incPredParams ps pdg up
            (pl, tim1, morphs, dg1) = processParameters ps timUp dgUp
            top_sg = Maude.Sign.fromSpec sp_mod
            paramSorts = getSortsParameterizedBy (paramNames ps) (Set.toList $ sorts top_sg
            ips = processImports tim1 vm dg1 il
            (tim2, dg2) = last_da ips (tim1, dg1)
            sg = sign_union_morphs morphs $ sign_union top_sg ips
            ext_sg = makeExtSign Maude sg
            nm_sns = map (makeNamed "") $ Maude.Sentence.fromSpec sp_mod
            sens = toThSens nm_sns
            gt = G_theory Maude ext_sg startSigId sens startThId
            tok = HasName.getName sp_mod
            name = makeName tok
            (ns, dg3) = insGTheory dg2 name DGBasic gt
            tim3 = Map.insert tok (getNode ns, sg, [], pl, paramSorts) tim2
            (tim4, dg4) = createEdgesImports tok ips sg tim3 dg3
            dg5 = createEdgesParams tok pl morphs sg tim4 dg4
            (_, tim5, dg6) = insertFreeNode tok tim4 morphs dg5
```

When the specification inserted is a theory the process varies slightly:

- Theories cannot be parameterized in Core Maude, so the parameter handling is not required.
- The specified sorts have to be qualified with the parameter name when used in a parameterized module. These sorts are extracted with `getImportsSorts` and kept in the corresponding field of `TokenInfoMap`.

```
insertSpec (SpecTh sp_th) pdg ptim tim vm ths dg = (ptimUp, tim3, vm, tok : ths, dg3)
      where (il, ss1) = getImportsSorts sp_th
            (ptimUp, timUp, dgUp) = incPredImps il pdg (ptim, tim, dg)
            ips = processImports timUp vm dgUp il
            ss2 = getThSorts ips
            (tim1, dg1) = last_da ips (tim, dg)
            sg = sign_union (Maude.Sign.fromSpec sp_th) ips
            ext_sg = makeExtSign Maude sg
            nm_sns = map (makeNamed "") $ Maude.Sentence.fromSpec sp_th
            sens = toThSens nm_sns
            gt = G_theory Maude ext_sg startSigId sens startThId
            tok = HasName.getName sp_th
            name = makeName tok
            (ns, dg2) = insGTheory dg1 name DGBasic gt
            tim2 = Map.insert tok (getNode ns, sg, ss1 ++ ss2, [], []) tim1
            (tim3, dg3) = createEdgesImports tok ips sg tim2 dg2
```

The introduction of views into the development graph follows these steps:

- The function `isInstantiated` checks whether the target of the view is a theory or a module. This value will be used to decide whether the sorts have to be qualified when this view is used.
- A morphism is generated between the signatures of the source and target specifications.
- If there is a renaming between terms the function `sign4renamings` generates the extra signature and sentences needed. These values, kept in `new_sign` and `new_sens` are used to create an inner node with the function `insertInnerNode`.
- Finally, a theorem link stating the proof obligations generated by the view is introduced between the source and the target of the view with `insertThmEdgeMorphism`.

```
insertSpec (SpecView sp_v) pdg ptim tim vm ths dg = (ptimUp, tim3, vm', ths, dg4)
      where View name from to rnms = sp_v
            (ptimUp, timUp, dgUp) = incPredView from to pdg (ptim, tim, dg)
            inst = isInstantiated ths to
            tok_name = HasName.getName name
            (tok1, tim1, morph1, _, dg1) = processModExp timUp vm dgUp from
            (tok2, tim2, morph2, _, dg2) = processModExp tim1 vm dg1 to
            (n1, _, _, _, _) = fromJust $ Map.lookup tok1 tim2
            (n2, _, _, _, _) = fromJust $ Map.lookup tok2 tim2
            morph = fromSignsRenamings (target morph1) (target morph2) rnms
            morph' = fromJust $ maybeResult $ compose morph1 morph
            (new_sign, new_sens) = sign4renamings (target morph1) (sortMap morph) rnms
            (n3, tim3, dg3) = insertInnerNode n2 tim2 tok2 morph2 new_sign new_sens dg2
            vm' = Map.insert (HasName.getName name) (n3, tok2, morph', rnms, inst) vm
            dg4 = insertThmEdgeMorphism tok_name n3 n1 morph' dg3
```

We describe now the main auxiliary functions used above. Module expressions are parsed following the guidelines outlined in Section 5.1.2:

- When the module expression is a simple identifier its signature and its parameterized sorts are extracted from the `TokenInfoMap` and returned, while the generated morphism is an inclusion:

```
processModExp :: TokenInfoMap -> ViewMap -> DGraph -> ModExp -> ModExpProc
processModExp tim _ dg (ModExp modId) = (tok, tim, morph, ps, dg)
                      where tok = HasName.getName modId
                            (_, sg, _, _, ps) = fromJust $ Map.lookup tok tim
                            morph = Maude.Morphism.inclusion sg sg
```

- The parsing of the summation expression performs the following steps:
  - The information about the module expressions is recursively computed with `processModExp`.
  - The signature of the resulting module expression is obtained with the `union` of signatures.
  - The morphism generated by the summation is just an inclusion.
  - A new node for the summation is introduced with `insertNode`.
  - The target signature of the obtained morphisms is substituted by this new signature with `setTarget`.
  - These new morphisms are used to generate the links between the summation and its summands in `insertDefEdgeMorphism`.

```
processModExp tim vm dg (SummationModExp modExp1 modExp2) = (tok, tim3, morph, ps', dg5)
          where (tok1, tim1, morph1, ps1, dg1) = processModExp tim vm dg modExp1
                (tok2, tim2, morph2, ps2, dg2) = processModExp tim1 vm dg1 modExp2
                ps' = deleteRepeated $ ps1 ++ ps2
                tok = mkSimpleId $ concat ["{", show tok1, " + ", show tok2, "}"]
                (n1, _, ss1, _, _) = fromJust $ Map.lookup tok1 tim2
                (n2, _, ss2, _, _) = fromJust $ Map.lookup tok2 tim2
                ss1' = translateSorts morph1 ss1
                ss2' = translateSorts morph1 ss2
                sg1 = target morph1
                sg2 = target morph2
                sg = Maude.Sign.union sg1 sg2
                morph = Maude.Morphism.inclusion sg sg
                morph1' = setTarget sg morph1
                morph2' = setTarget sg morph2
                (tim3, dg3) = insertNode tok sg tim2 (ss1' ++ ss2') [] dg2
                (n3, _, _, _, _) = fromJust $ Map.lookup tok tim3
                dg4 = insertDefEdgeMorphism n3 n1 morph1' dg3
                dg5 = insertDefEdgeMorphism n3 n2 morph2' dg4
```

- The renaming module expression recursively parses the inner expression, computes the morphism from the given renamings with `fromSignRenamings`, taking special care of the renaming of the parameterized sorts with `applyRenamingParamSorts`. Once the values are computed, the final morphism is obtained from the composition of the morphisms computed for the inner expression and the one computed from the renamings:

```
processModExp tim vm dg (RenamingModExp modExp rnms) = (tok, tim', comp_morph, ps', dg')
              where (tok, tim', morph, ps, dg') = processModExp tim vm dg modExp
                    morph' = fromSignRenamings (target morph) rnms
                    ps' = applyRenamingParamSorts (sortMap morph') ps
                    comp_morph = fromJust $ maybeResult $ compose morph morph'
```

- The parsing of the instantiation module expression works as follows:
  - The information of the instantiated parameterized module is obtained with `processModExp`.
  - The parameter names are obtained by applying `fstTpl`, that extracts the first component of a triple, to the information about the parameters of the parameterized module.
  - Parameterized sorts are instantiated with `instantiateSorts`, that returns the new parameterized sorts, in case the target of the view is a theory, and the morphism associated.
  - The view identifiers are processed with `processViews`. This function returns the token identifying the list of views, the morphism to be applied from the parameterized module, a list of pairs of nodes and morphisms, indicating the morphism that has to be used in the link from each view, and a list with the updated information about the parameters due to the views with theories as target.
  - The morphism returned is the inclusion morphism.
  - The links between the targets of the views and the expression are created with `updateGraphViews`.

```
processModExp tim vm dg (InstantiationModExp modExp views) =
                                        (tok'', tim'', final_morph, new_param_sorts, dg'')
        where (tok, tim', morph, paramSorts, dg') = processModExp tim vm dg modExp
              (_, _, _, ps, _) = fromJust $ Map.lookup tok tim'
              param_names = map fstTpl ps
              view_names = map HasName.getName views
              (new_param_sorts, ps_morph) = instantiateSorts param_names
                                                             view_names vm morph paramSorts
              (tok', morph1, ns, deps) = processViews views (mkSimpleId "") tim'
                                                    vm ps (ps_morph, [], [])
              tok'' = mkSimpleId $ concat [show tok, "{", show tok', "}"]
              sg2 = target morph1
              final_morph = Maude.Morphism.inclusion sg2 sg2
              (tim'', dg'') = if Map.member tok'' tim
                              then (tim', dg')
                              else updateGraphViews tok tok'' sg2 morph1 ns tim' deps dg'
```

We present the function `insertNode` to describe how the nodes are introduced into the development graph. This function receives the identifier of the node, its signature,[12] the `TokenInfoMap` map, a list of sorts, and information about the parameters, and returns the updated map and the new development graph. First, it checks whether the node is already in the development graph. If it is in the graph, the current map and graph are returned. Otherwise, the extended signature is computed with `makeExtSign` and used to create a graph theory that will be inserted with `insGTheory`, obtaining the new node information and the new development graph. Finally, the map is updated with the information received as parameter and the node identifier obtained when the node was introduced:

```
insertNode :: Token -> Sign -> TokenInfoMap -> Symbols -> [(Token, Token, Symbols)]
              -> DGraph -> (TokenInfoMap, DGraph)
insertNode tok sg tim ss deps dg = if Map.member tok tim
                     then (tim, dg)
                     else let
                            ext_sg = makeExtSign Maude sg
                            gt = G_theory Maude ext_sg startSigId noSens startThId
                            name = makeName tok
                            (ns, dg') = insGTheory dg name DGBasic gt
                            tim' = Map.insert tok (getNode ns, sg, ss, deps, []) tim
                          in (tim', dg')
```

The function `insertDefEdgeMorphism` describes how the definition links are introduced into the development graph. It receives the identifier of the source and target nodes, the morphism to be used in the link, and the current development graph. The morphism is transformed into a development graph morphism indicating the current logic (`Maude`) and the type (`globalDef`) and is introduced in the development graph with `insLEdgeDG`:

```
insertDefEdgeMorphism :: Node -> Node -> Morphism -> DGraph -> DGraph
insertDefEdgeMorphism n1 n2 morph dg = snd $ insLEdgeDG (n2, n1, edg) dg
                     where mor = G_morphism Maude morph startMorId
                           edg = globDefLink (gEmbed mor) SeeTarget
```

Theorem links are introduced with `insertThmEdgeMorphism` in the same way, but specifying with `globalThm` that the link is a theorem link. This function receives as extra argument the name of the view generating the proof obligations, that is used to name the link:

```
insertThmEdgeMorphism :: Token -> Node -> Node -> Morphism -> DGraph -> DGraph
insertThmEdgeMorphism name n1 n2 morph dg = snd $ insLEdgeDG (n2, n1, edg) dg
                     where mor = G_morphism Maude morph startMorId
                           edg = defDGLink (gEmbed mor) globalThm
                                 (DGLinkView name $ Fitted [])
```

The function `insertFreeEdge` receives the names of the nodes and the `TokenInfoMap` and builds an inclusion morphism to use it in the `FreeOrCofreeDefLink` link:

[12] Note that when the function `insertNode` is used there are no sentences.

```
insertFreeEdge :: Token -> Token -> TokenInfoMap -> DGraph -> DGraph
insertFreeEdge tok1 tok2 tim dg = snd $ insLEdgeDG (n2, n1, edg) dg
          where (n1, _, _, _, _) = fromJust $ Map.lookup tok1 tim
                (n2, sg2, _, _, _) = fromJust $ Map.lookup tok2 tim
                mor = G_morphism Maude (Maude.Morphism.inclusion Maude.Sign.empty sg2) startMorId
                dgt = FreeOrCofreeDefLink NPFree $ EmptyNode (Logic Maude)
                edg = defDGLink (gEmbed mor) dgt SeeTarget
```

7.5 Comorphism

We show in this section how the comorphism from Maude to CASL described in Section 4 is implemented. The function in charge of computing the comorphism is `maude2casl`, that returns the CASL signature and sentences given the Maude signature and sentences:

```
maude2casl :: MSign.Sign -> [Named MSentence.Sentence] -> (CSign.CASLSign,
                                                           [Named CAS.CASLFORMULA])
```

This function splits the work into different stages:

- The function `rewPredicates` generates the `rew` predicates for each sort to simulate the rewrite rules in the Maude specification.
- The function `rewPredicatesSens` creates the formulas associated to the `rew` predicates created above, stating that they are reflexive and transitive.
- The CASL operators are obtained from the Maude operators:
  - The function `translateOps` splits the Maude operator map into a tuple of CASL operators and CASL associative operators, (which are required for parsing purposes).
  - Since CASL does not allow the definition of polymorphic operators, these operators are removed from the map with `deleteUniversal` and for each one of these Maude operators we create a set of CASL operators with all the possible profiles with `universalOps`.
- CASL sentences are obtained from the Maude sentences and from predefined CASL libraries:
  - In the computation of the CASL formulas we split Maude sentences in equations defined without the `owise` attribute, equations defined with `owise`, and the rest of statements with the function `splitOwiseEqs`.
  - The equations defined without the `owise` attribute are translated as universally quantified equations, as shown in Section 4, with `noOwiseSen2Formula`.
  - Equations with the `owise` attribute are translated using a negative existential quantification, as we will show later, with the function `owiseSen2Formula`. This function requires as additional parameter the definition of the formulas defined without the `owise` attribute, in order to state that the equations defined with `owise` are applied when the rest of possible equations cannot.
  - The rest of statements, namely memberships and rules, are translated with the function `mb_rl2formula`.
  - There are some built-in operators in Maude that are not defined by means of equations. To allow the user to reason about them we provide some libraries with the definitions of these operators as CASL formulas, obtained with `loadLibraries`.
- Finally, the CASL symbols are created:
  - The kinds are translated to symbols with `kinds2syms`.
  - The operators are translated with `ops2symbols`.
  - The symbol predicates are obtained with `preds2syms`.

```
maude2casl msign nsens = (csign { CSign.sortSet = cs,
                                  CSign.sortRel = sbs',
                                  CSign.opMap = cops',
                                  CSign.assocOps = assoc_ops,
                                  CSign.predMap = preds,
                                  CSign.declaredSymbols = syms }, new_sens)
   where csign = CSign.emptySign ()
         ss = MSign.sorts msign
         ss' = Set.map sym2id ss
         mk = kindMapId $ MSign.kindRel msign
         sbs = MSign.subsorts msign
         sbs' = maudeSbs2caslSbs sbs mk
         cs = Set.union ss' $ kindsFromMap mk
         preds = rewPredicates cs
         rs = rewPredicatesSens cs
         ops = deleteUniversal $ MSign.ops msign
         ksyms = kinds2syms cs
         (cops, assoc_ops, _) = translateOps mk ops
         cops' = universalOps cs cops $ booleanImported ops
         rs' = rewPredicatesCongSens cops'
         pred_forms = loadLibraries (MSign.sorts msign) ops
         ops_syms = ops2symbols cops'
         (no_owise_sens, owise_sens, mbs_rls_sens) = splitOwiseEqs nsens
         no_owise_forms = map (noOwiseSen2Formula mk) no_owise_sens
         owise_forms = map (owiseSen2Formula mk no_owise_forms) owise_sens
         mb_rl_forms = map (mb_rl2formula mk) mbs_rls_sens
         preds_syms = preds2syms preds
         syms = Set.union ksyms $ Set.union ops_syms preds_syms
         new_sens = concat [rs, rs', no_owise_forms, owise_forms,
                            mb_rl_forms, pred_forms]
```

The `rew` predicates are declared with the function `rewPredicates`, that traverses the set of sorts applying the function `rewPredicate`:

```
rewPredicates :: Set.Set Id -> Map.Map Id (Set.Set CSign.PredType)
rewPredicates = Set.fold rewPredicate Map.empty
```

This function defines a binary predicate using as identifier the constant `rewID` and the sort as type of the arguments:

```
rewPredicate :: Id -> Map.Map Id (Set.Set CSign.PredType)
                -> Map.Map Id (Set.Set CSign.PredType)
rewPredicate sort m = Map.insertWith (Set.union) rewID ar m
   where ar = Set.singleton $ CSign.PredType [sort, sort]
```

Once these predicates have been declared, we have to introduce formulas to state their properties. The function `rewPredicatesSens` accomplishes this task by traversing the set of sorts and applying `rewPredicateSens`:

```
rewPredicatesSens :: Set.Set Id -> [Named CAS.CASLFORMULA]
rewPredicatesSens = Set.fold rewPredicateSens []
```

This function generates the formulas for each sort:

```
rewPredicateSens :: Id -> [Named CAS.CASLFORMULA] -> [Named CAS.CASLFORMULA]
rewPredicateSens sort acc = ref : trans : acc
        where ref = reflSen sort
              trans = transSen sort
```

We describe the formula for reflexivity, being the formula for transitivity analogous. A new variable of the required sort is created with the auxiliary function `newVar`, then the qualified predicate name is created with the `rewID` constant and applied to the variable. Finally, the formula is named with the prefix `rew_refl_` followed by the name of the sort:

```
reflSen :: Id -> Named CAS.CASLFORMULA
reflSen sort = makeNamed name $ quantifyUniversally form
        where v = newVar sort
              pred_type = CAS.Pred_type [sort, sort] nullRange
              pn = CAS.Qual_pred_name rewID pred_type nullRange
              form = CAS.Predication pn [v, v] nullRange
              name = "rew_refl_" ++ show sort
```

The function `translateOps` traverses the map of Maude operators, applying to each of them the function `translateOpDeclSet`:

```
translateOps :: IdMap -> MSign.OpMap -> OpTransTuple
translateOps im = Map.fold (translateOpDeclSet im) (Map.empty, Map.empty, Set.empty)
```

Since the values in the Maude operator map are sets of operator declarations, the auxiliary function `translateOpDeclSet` has to traverse these sets, applying `translateOpDecl` to each operator declaration:

```
translateOpDeclSet :: IdMap -> MSign.OpDeclSet -> OpTransTuple -> OpTransTuple
translateOpDeclSet im ods tpl = Set.fold (translateOpDecl im) tpl ods
```

The function `translateOpDecl` receives an operator declaration, that consists of all the operators declared with the same profile at the kind level. The function traverses these operators, transforming them into CASL operators with the function `ops2pred` and returning a tuple containing the operators, the associative operators, and the constructors:

```
translateOpDecl :: IdMap -> MSign.OpDecl -> OpTransTuple -> OpTransTuple
translateOpDecl im (syms, ats) (ops, assoc_ops, cs) = case tl of
                     [] -> (ops', assoc_ops', cs')
                     _ -> translateOpDecl im (syms', ats) (ops', assoc_ops', cs')
      where sym = head $ Set.toList syms
            tl = tail $ Set.toList syms
            syms' = Set.fromList tl
            (cop_id, ot, _) = fromJust $ maudeSym2CASLOp im sym
            cop_type = Set.singleton ot
            ops' = Map.insertWith (Set.union) cop_id cop_type ops
            assoc_ops' = if any MAS.assoc ats
                         then Map.insertWith (Set.union) cop_id cop_type assoc_ops
                         else assoc_ops
            cs' = if any MAS.ctor ats
                  then Set.insert (Component cop_id ot) cs
                  else cs
```

As said above, Maude equations that are not defined with the `owise` attribute are translated to CASL with `noOwiseSen2Formula`. This function extracts the current equation from the named sentence, translates it with `noOwiseEq2Formula` and creates a new named sentence with the resulting formula:

```
noOwiseSen2Formula ::  IdMap -> Named MSentence.Sentence -> Named CAS.CASLFORMULA
noOwiseSen2Formula im s = s'
       where MSentence.Equation eq = sentence s
             sen' = noOwiseEq2Formula im eq
             s' = s { sentence = sen' }
```

The function `noOwiseEq2Formula` distinguishes whether the equation is conditional or not. In both cases, the Maude terms in the equation are translated into CASL terms with `maudeTerm2caslTerm`, and a strong equation is used to create a formula. If the equation has no conditions this formula is universally quantified and returned as result, while if it has conditions each of them generates a formula and their conjunction, computed with `conds2formula`, will be used as premise of the equational formula:

```
noOwiseEq2Formula :: IdMap -> MAS.Equation -> CAS.CASLFORMULA
noOwiseEq2Formula im (MAS.Eq t t' [] _) = quantifyUniversally form
      where ct = maudeTerm2caslTerm im t
            ct' = maudeTerm2caslTerm im t'
            form = CAS.Strong_equation ct ct' nullRange
noOwiseEq2Formula im (MAS.Eq t t' conds@(_:_) _) = quantifyUniversally form
```

```
 where ct = maudeTerm2caslTerm im t
       ct' = maudeTerm2caslTerm im t'
       conds_form = conds2formula im conds
       concl_form = CAS.Strong_equation ct ct' nullRange
       form = createImpForm conds_form concl_form
```

`maudeTerm2caslTerm` is defined for each Maude term:

- Variables are translated into qualified CASL variables, and their type is translated to the corresponding type in CASL:

```
maudeTerm2caslTerm :: IdMap -> MAS.Term -> CAS.CASLTERM
maudeTerm2caslTerm im (MAS.Var q ty) = CAS.Qual_var q ty' nullRange
        where ty' = maudeType2caslSort ty im
```

- Constants are translated as functions applied to the empty list of arguments:

```
maudeTerm2caslTerm im (MAS.Const q ty) = CAS.Application op [] nullRange
        where name = token2id q
              ty' = maudeType2caslSort ty im
              op_type = CAS.Op_type CAS.Total [] ty' nullRange
              op = CAS.Qual_op_name name op_type nullRange
```

- The application of an operator to a list of terms is translated into another application, translating recursively the arguments into valid CASL terms:

```
maudeTerm2caslTerm im (MAS.Apply q ts ty) = CAS.Application op tts nullRange
        where name = token2id q
              tts = map (maudeTerm2caslTerm im) ts
              ty' = maudeType2caslSort ty im
              types_tts = getTypes tts
              op_type = CAS.Op_type CAS.Total types_tts ty' nullRange
              op = CAS.Qual_op_name name op_type nullRange
```

The conditions are translated into a conjunction with `conds2formula`, that traverses the conditions applying `cond2formula` to each of them, and then creates the conjunction of the obtained formulas:

```
conds2formula :: IdMap -> [MAS.Condition] -> CAS.CASLFORMULA
conds2formula im conds = CAS.Conjunction forms nullRange
       where forms = map (cond2formula im) conds
```

- Both equality and matching conditions are translated into strong equations:

```
cond2formula :: IdMap -> MAS.Condition -> CAS.CASLFORMULA
cond2formula im (MAS.EqCond t t') = CAS.Strong_equation ct ct' nullRange
       where ct = maudeTerm2caslTerm im t
             ct' = maudeTerm2caslTerm im t'
cond2formula im (MAS.MatchCond t t') = CAS.Strong_equation ct ct' nullRange
       where ct = maudeTerm2caslTerm im t
             ct' = maudeTerm2caslTerm im t'
```

- Membership conditions are translated into CASL memberships by translating the term and the sort:

```
cond2formula im (MAS.MbCond t s) = CAS.Membership ct s' nullRange
      where ct = maudeTerm2caslTerm im t
            s' = token2id $ getName s
```

- Rewrite conditions are translated into formulas by using both terms as arguments of the corresponding `rew` predicate:

```
cond2formula im (MAS.RwCond t t') = CAS.Predication pred_name [ct, ct'] nullRange
       where ct = maudeTerm2caslTerm im t
             ct' = maudeTerm2caslTerm im t'
             ty = token2id $ getName $ MAS.getTermType t
             kind = Map.findWithDefault (errorId "rw cond to formula") ty im
             pred_type = CAS.Pred_type [kind, kind] nullRange
             pred_name = CAS.Qual_pred_name rewID pred_type nullRange
```

The equations defined with the `owise` attribute are translated with `owiseSen2Formula`, that traverses them and applies `owiseEq2Formula` to the inner equation:

```
owiseSen2Formula ::  IdMap -> [Named CAS.CASLFORMULA] -> Named MSentence.Sentence
                     -> Named CAS.CASLFORMULA
owiseSen2Formula im owise_forms s = s'
       where MSentence.Equation eq = sentence s
             sen' = owiseEq2Formula im owise_forms eq
             s' = s { sentence = sen' }
```

This function receives all the formulas defined without the `owise` attribute and, for each formula with the same operator in the lefthand side as the current equation (obtained with `getLeftApp`), it generates with `existencialNegationOtherEqs` a negative existential quantification stating that the arguments do not match or the condition does not hold that is used as premise of the equation:

```
owiseEq2Formula :: IdMap -> [Named CAS.CASLFORMULA] -> MAS.Equation -> CAS.CASLFORMULA
owiseEq2Formula im no_owise_form eq = form
      where (eq_form, vars) = noQuantification $ noOwiseEq2Formula im eq
            (op, ts, _) = fromJust $ getLeftApp eq_form
            ex_form = existencialNegationOtherEqs op ts no_owise_form
            imp_form = createImpForm ex_form eq_form
            form = CAS.Quantification CAS.Universal vars imp_form nullRange
```

The translation from sorts, operators, and predicates to symbols works in a similar way to the transformations shown above, so we only describe how the predicate symbols are obtained. The function `preds2syms` traverses the map of predicates and inserts each obtained symbol into the set with `pred2sym`:

```
preds2syms :: Map.Map Id (Set.Set CSign.PredType) -> Set.Set CSign.Symbol
preds2syms = Map.foldWithKey pred2sym Set.empty
```

This function traverses the set of predicate types and creates the symbol corresponding to each one with `createSym4id`:

```
pred2sym :: Id -> Set.Set CSign.PredType -> Set.Set CSign.Symbol -> Set.Set CSign.Symbol
pred2sym pn spt acc = Set.fold (createSym4id pn) acc spt
```

`createSym4id` generates the symbol and inserts it into the accumulated set:

```
createSym4id :: Id -> CSign.PredType -> Set.Set CSign.Symbol -> Set.Set CSign.Symbol
createSym4id pn pt acc = Set.insert sym acc
      where sym = CSign.Symbol pn $ CSign.PredAsItemType pt
```

7.6 Freeness constraints

We describe here how the freeness constraints introduced in Section 5.2 have been implemented. This implementation has been performed for general CASL theories, so it is not specific to Maude specifications. Since this transformation is quite complex, we focus in this section on the second-order formula for the kernel of h (the functions added for each $\iota(s)$). As explained before, this formula is split into several subformulas:

- The formula for *symmetry* of each sort is implemented by the function `symmetry_ax`. It first generates two fresh variables, `v1` and `v2`, of the given sort, and a binary predicate `ps` ranging in this sort; then we create the righthand (`rhs`) and lefthand (`lhs`) sides of the implication, that finally is universally quantified and returned:

```
symmetry_ax :: SORT -> CASLFORMULA
symmetry_ax s = quant
     where free_sort = mkFreeName s
           v1@(Qual_var n1 _ _) = newVarIndex 1 free_sort
           v2@(Qual_var n2 _ _) = newVarIndex 2 free_sort
           pt = Pred_type [free_sort, free_sort] nullRange
           name = phiName s
           ps = Qual_pred_name name pt nullRange
           lhs = Predication ps [v1, v2] nullRange
           rhs = Predication ps [v2, v1] nullRange
           inner_form = Implication lhs rhs True nullRange
           vd = [Var_decl [n1, n2] free_sort nullRange]
           quant = Quantification Universal vd inner_form nullRange
```

- The formulas for *transitivity* are generated with `transitivity_ax`, that works in a similar way to the function above: it first creates the fresh variables `v1`, `v2`, and `v3`, which are used in the `ps` predicate to build the formulas `fst_form` ($\Phi(\mathtt{v1}, \mathtt{v2})$), `snd_form` ($\Phi(\mathtt{v2}, \mathtt{v3})$), and `thr_form` ($\Phi(\mathtt{v1}, \mathtt{v3})$). The first two formulas are put together in the conjunction `conj` and imply the third one in `imp`. Finally, the formula is universally quantified and returned as `quant`:

```
transitivity_ax :: SORT -> CASLFORMULA
transitivity_ax s = quant
     where free_sort = mkFreeName s
           v1@(Qual_var n1 _ _) = newVarIndex 1 free_sort
           v2@(Qual_var n2 _ _) = newVarIndex 2 free_sort
           v3@(Qual_var n3 _ _) = newVarIndex 3 free_sort
           pt = Pred_type [free_sort, free_sort] nullRange
           name = phiName s
           ps = Qual_pred_name name pt nullRange
           fst_form = Predication ps [v1, v2] nullRange
           snd_form = Predication ps [v2, v3] nullRange
           thr_form = Predication ps [v1, v3] nullRange
           conj = mk_conj [fst_form, snd_form]
           imp = Implication conj thr_form True nullRange
           vd = [Var_decl [n1, n2, n3] free_sort nullRange]
           quant = Quantification Universal vd imp nullRange
```

- Formulas stating *congruence* are more complex. The function `congruence_ax` computes the axioms for each operator identifier by using the auxiliary function `congruence_ax_aux`:

```
congruence_ax :: Id -> Set.Set OpType -> [CASLFORMULA] -> [CASLFORMULA]
congruence_ax name sot acc = set_forms
     where set_forms = Set.fold ((:) . (congruence_ax_aux name)) acc sot
```

  This auxiliary function, after renaming the operator and the sorts to obtain the free identifiers, creates the arrays of variables, `xs` and `ys`, and the two terms where they are applied, `fst_term` and `snd_term`. Definedness formulas of the form $D(t)$ are built as $P_s(t, t)$ in `fst_form` and `snd_form`, while the third part of the conjunction is built with `congruence_ax_vars` (not shown here) and kept in `vars_forms`. These formulas are put together as a conjunction and used in the final implication, which is returned once it is universally quantified:

```
congruence_ax_aux :: Id -> OpType -> CASLFORMULA
congruence_ax_aux name ot = cong_form'
     where OpType _ args res = ot
           free_name = mkFreeName name
           free_args = map mkFreeName args
           free_res = mkFreeName res
           free_ot = Op_type Total free_args free_res nullRange
           free_os = Qual_op_name free_name free_ot nullRange
           lgth = length free_args
           xs = createVars 1 free_args
```

```
        ys = createVars (1 + lgth) free_args
        fst_term = Application free_os xs nullRange
        snd_term = Application free_os ys nullRange
        phi = phiName res
        pt = Pred_type [free_res, free_res] nullRange
        ps = Qual_pred_name phi pt nullRange
        fst_form = Predication ps [fst_term, fst_term] nullRange
        snd_form = Predication ps [snd_term, snd_term] nullRange
        vars_forms = congruence_ax_vars args xs ys
        conj = mk_conj $ fst_form : snd_form : vars_forms
        concl = Predication ps [fst_term, snd_term] nullRange
        cong_form = Implication conj concl True nullRange
        cong_form' = quantifyUniversally cong_form
```

- The formulas for *satThM* are obtained by applying `free_formula`, which performs the substitutions shown in Section 5.2, to the formulas in the current theory:

```
sat_thm_ax :: [Named CASLFORMULA] -> CASLFORMULA
sat_thm_ax forms = final_form
     where forms' = map (free_formula . sentence) forms
           final_form = mk_conj forms'
```

- Finally, *largerThanKerH* formulas are computed with `larger_than_ker_h`. We split the work between the function `ltkh_sorts`, in charge of the formulas related to the sorts, and `ltkh_preds`, in charge of the formulas related to the predicates:

```
larger_than_ker_h :: Set.Set SORT -> Map.Map Id (Set.Set PredType) -> CASLFORMULA
larger_than_ker_h ss mis = conj
     where ltkhs = ltkh_sorts ss
           ltkhp = ltkh_preds mis
           conj = mk_conj (ltkhs ++ ltkhp)
```

  We describe how the `ltkh_sorts` function works. This function traverses all the sorts and applies `ltkh_sort` to each of them:

```
ltkh_sorts :: Set.Set SORT -> [CASLFORMULA]
ltkh_sorts = Set.fold ((:) . ltkh_sort) []
```

  This auxiliary function creates variables of the given sort and uses them in the homomorphism function. The terms thus obtained are used in an existential equation to create the premise of the implication, while the conclusion is a predicate with these variables as arguments. Finally, the formula is universally quantified and returned:

```
ltkh_sort :: SORT -> CASLFORMULA
ltkh_sort s = imp'
     where free_s = mkFreeName s
           v1 = newVarIndex 1 free_s
           v2 = newVarIndex 2 free_s
           phi = phiName s
           pt = Pred_type [free_s, free_s] nullRange
           ps = Qual_pred_name phi pt nullRange
           ot_hom = Op_type Partial [free_s] s nullRange
           name_hom = Qual_op_name homId ot_hom nullRange
           t1 = Application name_hom [v1] nullRange
           t2 = Application name_hom [v2] nullRange
           prem = Existl_equation t1 t2 nullRange
           concl = Predication ps [v1, v2] nullRange
           imp = Implication prem concl True nullRange
           imp' = quantifyUniversally imp
```

8 Concluding remarks and future work

We have presented how Maude has been integrated into Hets, a parsing, static analysis, and proof management tool that combines various tools for different specification languages. To achieve this integration, we consider preordered algebra semantics for Maude and define an institution comorphism from Maude to Casl. This integration allows to prove properties of Maude specifications like those expressed in Maude views. We have also implemented a normalization of the development graphs that allows us to prove freeness constraints. We have used this transformation to connect Maude to Isabelle [30], a Higher Order Logic prover, and have demonstrated a small example proof about reversal of lists. Moreover, this encoding is suited for proofs of e.g. extensionality of sets, which require first-order logic, going beyond the abilities of existing Maude provers like ITP.

Since interactive proofs are often not easy to conduct, future work will make proving more efficient by adopting automated induction strategies like rippling [11]. We also have the idea to use the automatic first-order prover SPASS for induction proofs by integrating special induction strategies directly into Hets.

We have also studied the possible comorphisms from Casl to Maude. We distinguish whether the formulas in the source theory are confluent and terminating or not. In the first case, that we plan to check with the Maude termination [12] and confluence checker [13], we map formulas to equations, whose execution in Maude is more efficient, while in the second case we map formulas to rules.

Finally, we also plan to relate Hets' Modal Logic and Maude models in order to use the Maude model checker [6, Chapter 13] for linear temporal logic.

Acknowledgments We wish to thank Francisco Durán for discussions regarding freeness in Maude, Martin Kühl for cooperation on the implementation of the system presented here, and Maksym Bortin for help with the Isabelle proofs. This work has been supported by the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe), the German Research Council (DFG) under grant KO-2428/9-1, the Comunidad de Madrid project *PROMETIDOS* (S2009/TIC–1465), the MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01), and the Spanish Ministerio de Educación grant AP2005-007.

References

[1] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer*, 3(1):66–77, 2009.

[2] T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, editors. *Proceedings of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference*, volume 3273 of *Lecture Notes in Computer Science*. Springer, 2004.

[3] M. Bidoit and P. D. Mosses. Casl *User Manual*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[5] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1):386–414, 2006.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[8] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages.

[9] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):189–210, 2002.

[10] R. Diaconescu. *Institution-Independent Model Theory*. Birkhäuser Basel, 2008.

[11] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2004.

[12] F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, IJCAR 2008, Sydney, Australia, August 12-15*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.

[13] F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85, 2010.

[14] N. Een and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[15] A. Fuchs. Darwin: A Theorem Prover for the Model Evolution Calculus. Master's thesis, University of Koblenz-Landau, 2004.

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

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

[18] M. Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics (SIGSAM)*, 34(2):43–48, 2000.

[19] M. Kühl. Integrating Maude into Hets. Master's thesis, Universität Bremen, 2010.

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

[21] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[22] M. W. Moskewicz and C. F. Madigan. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.

[23] T. Mossakowski. Relating Casl with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[24] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Technical report, Universität Bremen, 2005. Habilitation thesis.

[25] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - proof management for structured specifications. *Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques*, 67(1-2):114–145, 2006.

[26] T. Mossakowski, C. Maeder, M. Codescu, and D. Lücke. Hets user guide – Version 0.97 –, January 2011.

[27] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2007.

[28] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In B. Beckert, editor, *VERIFY 2007, 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 119–135, 2007.

[29] P. Mosses, editor. Casl *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

[30] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[31] M. Norrish and K. Slind. The HOL system tutorial, September 2010.

[32] B. C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[33] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford University Press, 1987.

[34] A. Tarlecki. Institutions: An abstract framework for formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, pages 105–130. Springer, 1999.

[35] D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks. Using Vampire to reason with OWL. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference, ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 2004.

[36] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[37] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction, CADE 2009*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145. Springer, 2009.