Programming and Symbolic Computation in Maude

Francisco Durán[a], Steven Eker[b], Santiago Escobar[c], Narciso Martí-Oliet[d], José Meseguer[e], Rubén Rubio[d], Carolyn Talcott[b]

[a] *Universidad de Málaga, Spain.*
[b] *SRI International, CA, USA.*
[c] *Universitat Politècnica de València, Spain.*
[d] *Universidad Complutense de Madrid, Spain.*
[e] *University of Illinois at Urbana-Champaign, IL, USA.*

Abstract

Rewriting logic is both a flexible *semantic framework* within which widely different concurrent systems can be naturally specified and a *logical framework* in which widely different logics can be specified. Maude programs are exactly rewrite theories. Maude has also a formal environment of verification tools. *Symbolic computation* is a powerful technique for reasoning about the correctness of concurrent systems and for increasing the power of formal tools. We present several new symbolic features of Maude that enhance formal reasoning about Maude programs and the effectiveness of formal tools. They include: (i) very general *unification* modulo user-definable equational theories, and (ii) *symbolic reachability analysis* of concurrent systems using narrowing. The paper does not focus just on symbolic features: it also describes several other new Maude features, including: (iii) Maude's *strategy language* for controlling rewriting, and (iv) *external objects* that allow flexible interaction of Maude object-based concurrent systems with the external world. In particular, *meta-interpreters* are external objects encapsulating Maude interpreters that can interact with many other objects. To make the paper self-contained and give a reasonably complete language overview, we also review the basic Maude features for equational rewriting and rewriting with rules, Maude programming of concurrent object systems, and reflection. Furthermore, we include many examples illustrating all the Maude notions and features described in the paper.

Keywords: Maude, rewriting logic, functional modules, system modules, parameterization, strategies, object-oriented programming, external objects, unification, narrowing, symbolic model checking, reflection, meta-interpreters.

Email addresses: `duran@lcc.uma.es` (Francisco Durán), `eker@csl.sri.com` (Steven Eker), `sescobar@dsic.upv.es` (Santiago Escobar), `narciso@ucm.es` (Narciso Martí-Oliet), `meseguer@illinois.edu` (José Meseguer), `rubenrub@ucm.es` (Rubén Rubio), `clt@cs.stanford.edu` (Carolyn Talcott)

Contents

1. Introduction

What is Maude? The Maude book's title [28] describes it as a *High-Performance Logical Framework* and adds: *How to Specify, Program and Verify Systems in Rewriting Logic*. Maude is indeed a declarative programming language based on rewriting logic [90, 19, 98].

So, what is rewriting logic? It is a logic ideally suited to specify *and* execute computational systems in a simple and natural way. Since nowadays most computational systems are concurrent, rewriting logic is particularly well suited to specify concurrent systems without making any a priori commitments about the model of concurrency in question, which can be synchronous or asynchronous, and can vary widely in its shape and nature: from a Petri net [128] to a process calculus [134, 126], from an object-based system [92] to asynchronous hardware [75], from a mobile ad hoc network protocol [79] to a cloud-based storage system [14], from a web browser [21, 122] to a programing language with threads [107, 108], from a distributed control system [105, 11] to a model of mammalian cell pathways [54, 130], and so on. And all *without any encoding*: what you see and get is a direct definition of the system itself, not some crazy Turing machine or Petri net encoding of it. All this means that rewriting logic is a flexible *semantic framework* to define and program computational systems. But since in rewriting logic

$$\textit{Computation} = \textit{Deduction}$$

the exact same flexibility can be used to specify any *logic* in rewriting logic, used now as a *logical framework*. Indeed, a logic's inference system can be naturally specified as a rewrite theory whose (possibly conditional) rewrite rules are exactly the logic's inference rules. Again, logics as different as linear logic, first-order logic, various modal logics, or all the higher-order logics in Barendregt's lambda cube can be specified in rewriting logic (and mechanized in Maude) *without any encoding* [83, 127, 112, 98]. This explains the "Logical Framework" part in the Maude book's title.

What about the "High-Performance" description? You should not take our word for it. Instead, you may wish to take a look at the paper [62], where a thorough benchmarking by H. Garavel and his collaborators at INRIA Rône-Alpes of a wide range of functional and rule-based declarative languages based on a large suite of benchmarks expressed in a language-independent manner and mapped into each language is reported. Although Maude is an interpreted language, it ranks second in overall performance for that suite, closely after Haskell.

What are Maude Programs? Rewrite theories. A rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory, with function symbols Σ and equations[1] E, specifying a concurrent sytem's states as an algebraic data

[1] As we explain in Section 2, the equational theory may also contain *membership axioms* specifying the typing of some expressions. For the moment think of E as containing both

type, and R is a set of rewrite rules that specify the *local concurrent transitions* that the concurrent system can perform. In Maude this is declared as a *system module* with syntax `mod` (Σ, E, R) `endm`. The degenerate case when $R = \emptyset$ gives rise to Maude's functional sublanguage of *functional modules*, which are declared with syntax `fmod` (Σ, E) `endfm` and specify the algebraic data type defined by E for the function symbols in Σ. Of course, this means that when writing and verifying Maude programs we *never* leave the realm of mathematics. This explains the qualification: "How to Specify, Program and Verify Systems in Rewriting Logic" in Maude book's title. Maude is not just a language: it has a formal environment of *verification tools*, some internal to the language and others built as language extensions (more on this in Sections 3, 7, and 9).

Why another paper on Maude? Maude is in her mid 20s. The first conference paper on Maude appeared in 1996 [29]. This was expanded into the 2002 journal paper [27], which to this date remains the most cited journal reference for the language. Important new advances were reported in the 2007 Maude book [28], which is the most highly cited reference on Maude to date. But a lot has happened since 2007. From time to time we have reported on new advances in a piecemeal way in a sequence of tool papers; but they are both quite brief and scattered over numerous publications: no unified account of the present state of Maude actually exists. That is why we decided to write this paper.

What is it like? On the one hand *repetition* of material already available in previous publications should be avoided; but on the other hand this paper should be a good entry point to learn about Maude as it is in 2019 *without assuming prior acquaintance with Maude*. Therefore, we have tried to strike a balance between: (i) making the paper self-contained and providing a reasonably complete *overview* of the language; and (ii) making sure that all the important *new features* now available in Maude are explained and illustrated. The way this balance between generality and novelty is attempted is reflected in the paper's organization. The most basic introduction to the language is given in Section 2 on functional modules and Section 3 on system modules. The first important new feature is Maude's *strategy language*, treated in Section 4. Section 5 on *object-based programming* is a mixture of old and new: on the one hand we introduce new readers to the basic ideas on how distributed object systems are programmed declaratively in Maude. On the other hand we explain several important new features on how Maude objects can now interact with various *external objects*. Another mixture of old and new is provided by Section 8 on reflection and meta-interpreters: reflection is a long-standing and crucial feature of both rewriting logic and Maude; but meta-interpreters are an entirely new feature. A very important additional theme with a host of new language features is reflected in the paper's title, namely, Maude's current support for *symbolic computation*. This theme is developed along three sections: Section 6 discusses unification, variants, and equational narrowing features; Section 7 dis-

equations and memberships.

cusses narrowing-based symbolic reachability analysis; and Section 9.1 discusses symbolic reasoning tools and applications.

Strategies. Most concurrent systems are intrinsically *non-deterministic*, so that different transitions may lead the system into widely different states. The obvious consequence is that an expression t in a system module can be evaluated by its rules R in many different ways. For example, the rewrite theory (Σ, E, R) may describe the game of chess, or the inference system of a theorem prover. But many chess moves may be stupid ones, and many inference steps may be useless. In both cases we need a *strategy* to apply the rules R in a way that achieves *our* intended goals. This is what Maude's *strategy language*, explained in Section 4, makes possible.

Specification and Deployment of Concurrent Object Systems. Although, as already mentioned, Maude can naturally express a wide variety of concurrent systems, many such systems are best expressed as collections of concurrent *objects* which communicate with each other by message passing. Section 5 explains how concurrent object systems can be programmed declaratively in Maude. But this leaves open two issues: (i) the object-based view, by its very nature, should allow interactions with *any* kind of object, including the user seen as an "object," and (ii) the Maude interpreter runs on a single machine, therefore the concurrent system defined by the program can be *simulated* and analyzed in a Maude interpreter; but how can it be *deployed* as a distributed system? Both issues are addressed by means of several kinds of *external objects* with which standard Maude objects can interact. In particular, using socket external objects, Maude programs can be deployed as distributed systems running on several machines.

Reflection and Meta-Interpreters. Rewriting logic is a *reflective* logic. This means that its *meta-theory*, including notions such as theory and term, can be *represented as data* at the so-called *object level* of the logic in a *universal theory*. It also means that such a universal theory, like in the case of universal Turing machines, can *simulate* any other theory, including itself. This is extremely powerful for (meta-)programming purposes and is efficiently supported by Maude's `META-LEVEL` module. Section 8 explains reflection, and also illustrates how meta-programming can be used to easily build advanced new tools such as an Eqlog [64] functional-logic programming interpreter. It also explains a very powerful new reflective feature, namely, *meta-interpreters*, which open the possibility of creating and interacting in a reflective manner with a hierarchy of Maude interpreters as *external objects*.

Maude and Symbolic Computation. Because Maude is a programming language *and* a logical framework in which many different logics and formal tools can be mechanized *and* has itself a formal environment of verification tools, support for *symbolic reasoning* is very important both for advanced formal reasoning about Maude programs and to use Maude as a *formal meta-tool* to build many other tools in other logics. From 2007 to the present, a sustained effort has been made to endow Maude with powerful symbolic reasoning capabilities.

At the *equational logic* level, they focus around the topic of Section 6, namely, *unification modulo an equational theory*, that is, solving systems of equations *modulo* an equational theory. Maude's unification features are extremely general in *three orthogonal dimensions*, corresponding to three aspects of an equational theory, which in Maude can have the form $(\Sigma, E \cup B)$, where Σ is an order-sorted signature (more on this in Section 2), B are common equational axioms such as associativity and/or commutativity and/or identity, and E are equations that are assumed convergent (more on this in Section 2) modulo B. The first dimension of generality is Σ: since order-sorted signatures are strictly more general than many-sorted ones, which are way more general than unsorted ones, order-sorted unification algorithms are much more general than the usual unsorted ones. The second dimension is unification modulo axioms B, which in Maude can be *any* combination of associativity and/or commutativity and/or identity axioms. The third dimension of generality is support for order-sorted unification modulo *any* theory $E \cup B$, where the axioms B are as explained and the equations E are convergent modulo B. Some very hard problems had to be solved to make $E \cup B$-unification practical and to characterize the cases when it terminates. They were solved in [60] thanks to the notion of *variant*, as also explained in Section 6.

At the *rewriting logic level*, Section 7 explains how the just-described support for unification modulo $E \cup B$ becomes a key symbolic lever to support narrowing-based *symbolic reachability analysis* for a rewrite theory (system module) $\mathcal{R} = (\Sigma, E \cup B, R)$, where $E \cup B$ has the so-called *finite variant property* [34], ensuring that $E \cup B$-unification terminates. Such reachability analysis provides a powerful form of *symbolic model checking* for $\mathcal{R}$, where possibly infinite sets of states are described by symbolic expressions. Using the new symbolic reasoning features described in Sections 6–7, many symbolic reasoning tools can be developed covering many applications. Section 9.1 focuses on those tools and applications most directly related to Maude itself; but similar formal tools can likewise be developed (and are developed) for many other logics [98].

Core Maude vs. Full Maude. Maude is also referred to as *Core Maude*. This is done to distinguish it from *Full Maude*. But what is Full Maude? What Maude is not yet but *will* be. Most new features presented in this paper — from the strategy language to variants, from unification algorithms to symbolic model checking, from object-oriented features to parameterized modules— first cut their teeth as features prototyped in Full Maude. How does Full Maude work? By *reflection*. That is, Full Maude is a *reflective Maude program* extending Maude itself with new language features [28]. As explained in [48], since many Maude verification tools need to manipulate Maude modules reflectively and should be well integrated with Maude itself, they can be built quite easily as *extensions* of Full Maude. Full Maude is not directly discussed in this paper; but, as Alfred Hitchcock in his movies, makes some interesting cameo appearances. A nice one takes place in Sections 8.2, where we show how the Eqlog [64] functional-logic language can be easily implemented using reflection and Maude's narrowing-based symbolic reachability and can be given an execution environment as an extension of Full Maude.

Differences with the Conference Paper [40]. This paper is a loose and very large extension of the conference paper [40]. Usually one says what has been added, but that would take too long. It is much shorter to say what has been loosely imported from [40], namely, some of the material in the "symbolic" Sections 6–7. But even that needs a few grains of salt. For example, since this paper focuses on language features and what they are good for, we have omitted the detailed description of Maude's order-sorted unification algorithm modulo *associativity* given in [40].

Examples and Maude Executables. The examples in the paper run on a Maude alpha version that will become a Maude release in the near future. The executables for that alpha version are available at `http://maude.sip.ucm.es/strategies/`. The Maude code for all the examples in the paper can be found at `http://maude.lcc.uma.es/maude28`.

2. Functional Modules

Maude is a declarative language based on rewriting logic; but rewriting logic has membership equational logic [95, 16] as its functional sublogic. From the computational point of view the key difference between rewriting logic and its membership equational sublogic is that between: (i) the *non-determinism* of rewrite theories, and (ii) the *determinism* of equational ones. That is, an equational program is a functional program in which a functional expression (called a *term*) is evaluated using the equations in the program as left-to-right rewrite rules, which are assumed *confluent* [35]. If such an evaluation terminates, it returns a *unique* computed value (determinism), namely, its *normal form* after simplifying it with the (oriented) equations. Instead, a rewrite theory usually models a *non-deterministic* and often *concurrent* system, which may never terminate and where the notion of a computed value may be meaningless.

In this section we present Maude *functional modules*, which are conditional membership equational theories of the form $(\Sigma, M \cup E \cup B)$ specifying functional programs, where: (i) Σ is the *signature* specifying: the types, here called *sorts*, the subtype, i.e., *subsort*, inclusions, and the function symbols and constants used in the theory; (ii) E is a collection of (possibly conditional) equations which are used as left-to-right rewrite rules to evaluate terms; (iii) B is a collection of equational axioms, such as associativity and/or commutativity and/or identity satisfied by some of the function symbols in Σ; such axioms are viewed as *structural axioms*, so that rewriting with the equations E is performed *modulo* the axioms B; and (iv) M is a collection of (possibly conditional) *memberships*, which can lower the sort of a term if a membership's condition is satisfied (more on this below). In Maude, the functional module defined by $(\Sigma, M \cup E \cup B)$ is declared within keywords `fmod` and `endfm`, and is also given a name, say, `FOO`, so that its declaration has the form: `fmod FOO is` $(\Sigma, M \cup E \cup B)$ `endfm`.

Maude's syntax for Σ, E, and M is self-explanatory: it is in essence the ASCII version of the standard textbook notation (see below). Instead, the structural axioms B *are declared together with the function symbols satisfying such axioms*. Furthermore, the syntax for the signature Σ, i.e., the *names* and syntactic form of the sorts, constants, and function symbols for Σ, is completely *user-definable*. For example, if `Nat` is the name we have chosen for the sort of natural numbers, we may choose any syntax we wish to declare a natural number addition function in Σ, and, furthermore, we may declare such a function as enjoying some structural axioms B. Suppose that `0` and `1` have been declared as constants with the declaration:

```
ops 0 1 : -> Nat [ctor] .
```

where the `ctor` declaration makes it clear that the constants `0` and `1` are *data constructors* that will not be evaluated away to other values by some equations. Then, we can choose any syntax we wish for the addition function. The less imaginative choice is to adopt a *prefix syntax*, such as `+(1,0)`, or `plus(1,0)`. But we may wish to use the more readable *infix syntax*, so as to be able to write the term `1 + 0`. Suppose we decide to give addition such an infix syntax and to declare it as enjoying the *associativity* axiom $(x + y) + z = x + (y + z)$, the *commutativity* axiom $x + y = y + x$, and the *identity* axiom $x + 0 = x = 0 + x$. Then, we can give this declaration in Maude as follows:

```
op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
```

where the two underbar symbols indicate where the first and second argument of the addition function must be placed before and after the `+` character. As before, the `ctor` declaration makes it clear that in this representation the addition symbol is a *data constructor*, which will not be evaluated away, except if one of its arguments is `0`, so that the identity axiom can be used. That is, in this representation the natural numbers are: $0,\ 1,\ 1+1, \ldots, 1+ \overset{n}{\ldots} +1, \ldots$. Instead, had we chosen the prefix syntax, say, `plus`, we would have given the alternative declaration:

```
op plus : Nat Nat -> Nat [ctor assoc comm id: 0] .
```

Order-sorted equational logic [66, 95] is a very useful sublogic of membership equatiognal logic. An order-sorted equational theory is a membership equational theory $(\Sigma, M \cup E \cup B)$ such that $M = \emptyset$, i.e., it has the form $(\Sigma, E \cup B)$, where $\Sigma = ((S, <), F)$ consists of a partially ordered set $(S, \leq)$ of sorts, where $\leq$ denotes *subsort inclusion*, and where F is a set of function symbols and constants *typed* with sorts in S. Function symbols in F can be *subsort overloaded* (also called *subtype polymorphic*). For example, we may introduce a sort `NzNat` of non-zero natural numbers as a subsort of `Nat`, declare instead `1` as a constant of sort `NzNat`, and add the additional declaration:

```
op _+_ : NzNat NzNat -> NzNat [ctor assoc comm id: 0] .
```

The only requirement is that, as done above, all subsort polymorphic function declarations must satisfy the *same* structural axioms.

Example 1. *Consider the following order-sorted specification of terms in prefix form, with an arbitrary number of constant and function symbols, as elements* *of a sort* `Term` *having two subsorts,* `Var` *of variables, and* `NvTerm` *of non-variable terms. Assuming that we import Maude's built-in modules* `NAT` *of natural numbers, with main sort* `Nat`*, and* `QID` *of quoted identifiers, with main sort* `Qid`*, both modules in* protecting *mode (i.e., the sorts in* `NAT` *and* `QID` *are not modified, but they are protected, in such an importation [28]) we can then define such a data type of terms as follows:*

```
fmod TERM is
  protecting NAT + QID .

  sort Variable .
  op x{_} : Nat -> Variable [ctor] .

  sorts Term NvTerm .
  subsort Qid < NvTerm < Term .
  subsort Variable < Term .
  op _[_] : Qid NeTermList -> NvTerm [ctor prec 40] .

  sort NeTermList .
  subsort Term < NeTermList .
  op _,_ : NeTermList NeTermList -> NeTermList [ctor assoc] .
endfm
```

where, since no equations have been declared (only the associativity structural axiom for non-empty lists of terms), all operators are data constructors*. For example, assuming a countable set of variables, say,* $x_1, x_2, \ldots, x_n \ldots$ *and arbitrary names for constants and function symbols, the term* $f(g(x_3, b, x_1), k(x_2))$ *is here represented as the term:* `'f['g[x{3},'b,x{1}],'k[x{2}]]` *of least sort*[2] `NvTerm`*. Instead,* `'b` *has least sort* `Qid`*, and* `x{3}` *has least sort* `Variable`*. But of course all these terms share the common supersort* `Term`*.*

Note that any finite poset, and in particular the poset of sorts $(S, \leq)$, can be viewed as the reflexive-transitive closure of a directed acyclic graph (DAG), and that the set of nodes of such a DAG breaks into a set of *connected components*. For example, in the signature $\Sigma = ((S, \leq), F)$ of the `TERM` module there are *three* connected components: (i) one involving the sort `Bool`, since the Booleans are imported by `NAT`, (ii) another involving the sort `Nat` and its subsort `NzNat`, and (iii) yet another involving the sorts `Qid`, `Var`, `NvTerm`, and `Term`. Maude automatically adds a new so-called *kind* supersort at the top of each connected component in the poset $(S, \leq)$ declared by the user, where kinds are indicated with a bracket notation. For this example, Maude will add kinds `[Bool]`, `[Nat]` and `[Term]` at the top of each of these three components. Furthermore, for each function symbol, say $f : s_1 \ldots s_n \rightarrow s$, in Σ a new subsort-overloaded symbol $f : [s_1] \ldots [s_n] \rightarrow [s]$ is also added by Maude at the kind level. Intuitively, terms whose least sort is a kind are viewed as *error terms*. For example, the least sort of the term `‘f[‘a][‘g[‘c,x{2}],’b]` is the kind `[Term]`, because `‘f[‘a]` is *not* a quoted identifier and therefore cannot be used as a function symbol. This is very useful to *give functional expressions the benefit of the doubt*, because at

[2]Under a simple syntactic condition on Σ checked by Maude, called *preregularity* [66] (more generally, preregularity *modulo* the structural axioms B [28]), any Σ-term t always has a smallest possible typing with a sort called its *least sort* and denoted $ls(t)$.

parse time only partial type information may be available, but as a computation progresses some typing problems may go away. For example, in a data type `RAT` of rational numbers an expression like `3 / (2 - 7)` can only be parsed with least sort `[Rat]`, but will happily evaluate to `- 3 / 5` with least sort `NzRat`. Instead, the evaluation of the term `3 / (7 - (4 + 3))` will yield the error term `3 / 0`, whose least sort is `[Rat]`.

2.1. Predicate Subtyping with Membership Predicates

The full generality of Maude functional modules as membership equational theories can be illustrated by means of the following module.

Example 2. *We define in Figure 1 a functional module* `PFUN` *of (finite) partial functions on the natural numbers.*

```
fmod PFUN is
  protecting NAT .

  sorts Pair PFun Rel Nat? .
  subsorts Pair < PFun < Rel .
  subsort Nat < Nat? .

  op undef : -> Nat? [ctor] .

  vars N M K : Nat .
  var R : Rel .
  var F : PFun .

  op [_,_] : Nat Nat -> Pair [ctor] .
  op null : -> PFun [ctor] .      *** empty relation
  op _,_ : [Rel] [Rel] -> [Rel] [ctor assoc comm id: null] .

  eq [N,M], [N,M] = [N,M] .       *** idempotency

  op def : Nat Rel -> Bool .      *** is number defined in relation
  eq def(N, null) = false .
  eq def(N, ([M,K], R)) = if N == M then true else def(N, R) fi .

  cmb ([N,M], F) : PFun if def(N, F) = false .

  op _{_} : PFun Nat -> Nat? .    *** partial function application
  eq null{N} = undef .
  eq ([N,K], F){M} = if N == M then K else F{M} fi .
endfm
```

Figure 1: `PFUN` module

A few things are worth mentioning about this example. First of all, an if-then-else operator with "mix-fix" syntax `if_then_else_fi` and with the obvious equational definition is added automatically by Maude to any module importing the `BOOL` module, which is always imported by default unless the user indicates otherwise [28]. Second, a built-in equality predicate `_==_` is also automatically added by Maude for each connected component. However, neither of these built-in operators are really needed: the user can easily define his/her own if-then-else, as well as an equality predicate for natural numbers, or for many other data types (see [70]). These two built-in operators have been used in the definitions of the `def` predicate and of partial function application. Third, because of the idempotency equation, all terms of sort `Rel` in normal form are finite *sets* of pairs, and therefore finite relations in the mathematical sense.

The key new feature used here is the *conditional membership* (introduced with keyword `cmb`) defining the sort `PFun` of finite partial functions. This sort is defined by three cases. A partial function `F` is either: (i) the empty relation, or (ii) a relation `R` that is a partial function and to which a new pair `[N,K]` has been added, provided `R` is undefined for the input `N`. Case (i) is covered by the unconditional membership and case (ii) is specified by the conditional membership. Membership predicates are unary predicates in postfix notation $\_ : s$, where $s \in S$ is a sort. Applied to a term t whose least sort belongs to the connected component of s (and could even be its kind sort), the predicate states that t has sort s. A single such membership predicate is called an *unconditional membership* and is introduced with the keyword `mb`. In general, both equations and memberships can be *conditional*, and have, respectively, the general form:

$$\texttt{ceq}\ t = t' \ \ \textit{if} \ \ u_1 = v_1 \ \wedge \ \ldots \ \wedge \ u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_j : s_j \ .$$

$$\texttt{cmb}\ t : s \ \ \textit{if} \ \ u_1 = v_1 \ \wedge \ \ldots \ \wedge \ u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_j : s_j \ .$$

That is, both equations and memberships may appear in their conditions. Using ASCII symbols, the conjunction symbol $\wedge$ is rendered in Maude as: `/\`. Since an equation $t = t'$ will be applied as a left-to-right rewrite rule $t \to t'$ to simplify terms, sort information should *increase* as such simplification proceeds. This is captured by the requirement that all equations $t = t'$ (conditional or not) in a functional module should be *sort-decreasing*. That is, for any substitution θ we should have $ls(t\theta) \geq ls(t'\theta)$, where $t\theta$ and $t'\theta$ denote the respective instantiations of t and t' by θ. In the most common cases, all the variables appearing in such formulas also appear in the term t at the left of the equation $t = t'$ or the membership $t : s$. Furthermore, the equations and memberships in a condition can appear in different orders. However, for greater expressiveness Maude allows conditional equations and memberships whose conditions *can have extra variables* that are *incrementally instantiated* by matching, provided they obey the syntactic requirements explained in [24, 28]. We explain the incremental evaluation of conditions by means of Example 8 in Section 3.

In any confluent and operationally terminating functional module (more on this below), any term can be evaluated to its unique *normal form* having a least possible sort by applying to it both the module's equations as left-to-right rewrite rules, and the memberships to lower its sort, where equations and memberships are applied *modulo* the axioms B. For example, the idempotency equation `[N,M],[N,M] = [N,M]` can be applied modulo the associativity-commutativity axioms for `_,_` to simplify the term `[1,2],[3,7],[1,2]` to the term `[1,2],[3,7]`, even though the two instances of `[1,2]` are not contiguous. This evaluation to normal form is performed with the `reduce` command (which can be abbreviated to `red`). For example, in the above functional module in Example 2 above we can perform the following evaluations:

```
reduce in PFUN : [1,2],[1,2],[3,7],[5,17],[3,7] .
result PFun: [1,2],[3,7],[5,17]

reduce in PFUN : [1,2],[3,7],[5,17] {3} .
result NzNat: 7
```

2.2. Initial Algebra Semantics

What is the *mathematical* meaning of a Maude functional module, say, `fmod FOO is` $(\Sigma, M \cup E \cup B)$ `endfm`? That is, what does such a module declaration *denote*? The answer is simple: Maude has an *initial algebra semantics* for such modules, so that what `FOO` denotes is the initial algebra [95] $T_{\Sigma/M \cup E \cup B}$ of the theory $(\Sigma, M \cup E \cup B)$. There are two possible descriptions of $T_{\Sigma/M \cup E \cup B}$, one more abstract, and another very concrete. In the abstract description an element $[t] \in T_{\Sigma/M \cup E \cup B}$ is the $=_{E \cup B}$-equivalence class of a ground Σ-term t (i.e., t has no variables), where $=_{E \cup B}$ is the *provable equality* equivalence relation in the theory $(\Sigma, M \cup E \cup B)$ [95]. Under the already-mentioned executability conditions of: (i) sort-decreasingness, (ii) operational termination, and (iii) confluence[3] modulo B —where properties (i)–(iii) are summarized saying that $(\Sigma, M \cup E \cup B)$ is *convergent* modulo B— the more concrete and informative description is given by the isomorphic algebra $C_{\Sigma/M \cup E,B} \cong T_{\Sigma/M \cup E \cup B}$, called the *canonical term algebra* of $(\Sigma, M \cup E \cup B)$, whose elements $[u] \in C_{\Sigma/M \cup E,B}$ are $=_B$-equivalence classes of ground terms u that are in normal form by the equations E and the memberships M modulo B. $C_{\Sigma/M \cup E,B}$ provides the most concrete possible semantics for `FOO`, since it is just the semantics of Maude's `reduce` command in the following sense: a ground term t that is evaluated to a term u by Maude's `reduce` command has as its *value* the B-equivalence class $[u] \in C_{\Sigma/M \cup E,B}$. Furthermore, thanks to the Church-Rosser Theorem for membership equational logic [16], what the isomorphism $C_{\Sigma/M \cup E,B} \cong T_{\Sigma/M \cup E \cup B}$ ensures is the *full agreement* between the *mathematical semantics* provided by $T_{\Sigma/M \cup E \cup B}$ and the rewriting-based *operational semantics* (for details see [16], and for the conditional order-sorted subcase modulo B see [82]), whose algebra of normal forms is precisely $C_{\Sigma/M \cup E,B}$.

2.3. Theories, Views and Parameterized Functional Modules

Maude, like its OBJ3 predecessor [67], supports a very expressive form of *parametric polymorphism* [129] by means of its parameterized modules. The extra expressiveness has to do with the fact that *parameters* are *not* just parametric *types*, but are instead specified by *parameter theories*. That is, not only types

[3] Under the operational termination assumption, confluence modulo B just means that, up to B-equality, the normal form of any term t by simplification with the equations E and memberships M modulo B is *unique*. Therefore, confluence means that evaluation to normal form is a *deterministic* computation.

(sorts) can be parametric: constants and function symbols can also be parametric, and, furthermore, parameter theories impose *semantic requirements*, in the form of logical axioms, that must be satisfied by any instantiation of a parameter theory with actual parameters to be correct. Roughly speaking,[4] a *parameter theory* called, say, `FOO`, is a membership equational theory $(\Sigma, M \cup E \cup B)$, which is declared in Maude with syntax: `fth FOO is` $(\Sigma, M \cup E \cup B)$ `endfth`.

What is the mathematical semantics of such a functional theory `FOO`? Unlike the case of a functional module, whose semantics is the initial algebra $T_{\Sigma/M \cup E \cup B}$, the semantics of a functional theory is the *class* of all $(\Sigma, M \cup E \cup B)$-algebras, denoted $\mathbf{Alg}_{(\Sigma, M \cup E \cup B)}$. That is, functional theories have a "loose semantics" that specifies all the possible instantiations of the parameter theory $(\Sigma, M \cup E \cup B)$ by an algebra $A \in \mathbf{Alg}_{(\Sigma, M \cup E \cup B)}$ as an actual parameter.

Let us illustrate the extra power of parameterized theories, as opposed to just parameterized sorts, by describing some examples at a high level (further details can be found in [28]). The case of having just a parameterized sort is handled by the trivial parameter theory `TRIV`:

```
fth TRIV is
  sort Elt .
endfth
```

Note that the class of algebras of this theory, $\mathbf{Alg}_{\mathtt{TRIV}} = \mathbf{Set}$, is precisely *the class* **Set** *of all sets*. Therefore, `TRIV` is exactly the theory of a *parametric type* in the standard sense. For example, the parameterized functional module of lists `LIST{X :: TRIV}` can be instantiated by any set, say A, as actual parameter to obtain the data type of lists with elements in A. Let us consider two other examples where the parameter theories are nontrivial. The functional module `SORTING{X :: TOSET}` provides a parameterized functional module to sort lists of elements for any totally ordered set $(A, \leq)$, that is, for any $(A, \leq) \in \mathbf{Alg}_{\mathtt{TOSET}}$, where `TOSET` is the functional theory of totally ordered sets.[5] Yet a third example is the functional module `POLY{R :: RING, X :: TRIV}` of *polynomials*, which has *two* parameter theories. The first if the theory `RING` of commutative rings, so that its actual parameters are commutative rings $(R, -, +, *, 0, 1) \in \mathbf{Alg}_{\mathtt{RING}}$ providing the ring of coefficients used in the polynomials. Instead, the actual parameters for the second theory `TRIV` are precisely sets $X \in \mathbf{Set}$ providing the set of *variables* used in the polynomial expressions. Of course, for parameter instantiations to be *correct*, all the *axioms* in theories such as `TOSET` or `RING` must be *satisfied* by their actual parameters, $(A, \leq)$ or $(R, -, +, *, 0, 1)$. Maude does not check the semantic correctness of instantia-

[4] In fact, parameter theories may also contain *initiality constraints* in the sense of, e.g., [63, 45], which can impose the requirement that some sorts and functions must be interpreted as the initial model of an imported subtheory. For example, a theory T may import the theory `NAT` of natural numbers in `protecting` mode, so that only models where `NAT` is interpreted as the natural numbers are accepted.

[5] The fact that, as explained in Footnote 4, a functional theory can include initiality constraints is useful in this case, since `TOSET` can be easily defined by importing the functional module `BOOL` in `protecting` mode (see [28]).

tions. However, tools like the Maude's Inductive Theorem Prover (ITP) [32] can be used for this purpose.

But how are parameter theories *instantiated* in Maude? By *theory interpretations*! Suppose that we want to instantiate the parameterized module `POLY{R :: RING, X :: TRIV}` to polynomials with rational coefficients and with quoted identifiers as variables. We can, for example, use Maude's modules `RAT` of rational numbers and `QID` of quoted identifiers in Maude's standard prelude as actual parameters. But any functional module *is* a theory, namely, a membership equational theory *with the initiality constraint* that a model belongs to its class of models iff it is an initial algebra for the theory. This means that not only the axioms explicitly mentioned in the functional module, but also all its *inductive consequences* are true in such models and therefore valid under the initiality constraint. So we just need two theory interpretations: `ring2RAT` : `RING` $\rightarrow$ `RAT` to get the actual ring of coefficients, and `Qid` : `TRIV` $\rightarrow$ `QID` to select the sort `Qid` in module `QID` as the set of variables. What is a theory interpretation? Given two membership equational theories, say $(\Sigma, M \cup E \cup B)$ and $(\Sigma', M' \cup E' \cup B')$, a *theory interpretation* (called a *view* in Maude) $V : (\Sigma, M \cup E \cup B) \rightarrow (\Sigma', M' \cup E' \cup B')$ is a *signature map* $V : \Sigma \rightarrow \Sigma'$ that preserves all the axioms $M \cup E \cup B$, in the sense that the translated axioms $V(M) \cup V(E) \cup V(B)$ are *logical consequences* of the theory $(\Sigma', M' \cup E' \cup B')$. What is a signature map $V : \Sigma \rightarrow \Sigma'$? It is a mapping of sorts and function symbols such that: (i) If $(S, \leq)$ and $(S', \leq')$ are the posets of sorts for Σ and Σ', then V is a *monotonic function* on sorts, and (ii) each constant a in Σ of sort s in S is mapped to a ground Σ'-term $V(a)$ with $ls(V(a)) \leq V(s)$, and each function symbol $f : s_1 \ldots s_n \rightarrow s$ in Σ is mapped to a Σ'-*term* $V(f) = t'$ with $ls(t') \leq V(s')$ and with variables among the $x_1 : V(s_1), \ldots, x_n : V(s_n)$ in such a way that V preserves subtype polymorphism. In Maude such theory interpretations are defined with syntax of the form (see [28] for more details):

```
view V from T to T' is
  sort S1 to S'1 .
  ...
  op f(X1:S1,...,Xn:Sn) to term t'(X1:V(S1),...,Xn:V(Sn)) .
  ...
endv
```

A parameterized functional module `M{X1 :: T1,..., Xm :: Tm}` can be *instantiated* by *replacing* its formal parameter theories `T1`, ..., `Tm` by corresponding views `V1`, ..., `Vm` from `T1`, ..., `Tm` to `T'1`, ..., `T'm`, where the `T'1`, ..., `T'm` need not be all different. In this way, we get the instance `M{V1, ... ,Vm}`. For example, polynomials with rational coefficients and quoted identifiers as variables are defined as follows:

```
fmod RAT-POLY is protecting POLY{Ring2RAT,Qid} . endfm
```

We say that a parameterized module `M{X1 :: T1, ... , Xm :: Tm}` is *fully instantiated* by the views `V1`, ..., `Vm` if their target theories `T'1`, ..., `T'm` are all (unparameterized) functional modules. But this is not the only possibility: a module may be instantiated in an *incremental* way. For example, we can define a view:

```
view triv2TOSET from TRIV to TOSET is
  sort Elt to Elt .
endv
```

to instantiate the list module `LIST{X :: TRIV}` to the module `LIST{triv2TOSET}` which is still parameterized, but now by the `TOSET` theory; and we can then use `LIST{triv2TOSET}` as part of the definition of a `SORTING{X :: TOSET}` module.

Let us see an example illustrating all the ideas discussed so far.

Example 3. *A parameterized module for finite partial functions generalizing the* `PFUN` *module of Example 2 can be found in Figure 2. This module is such a straightforward* generalization *of the* `PFUN` *module of Example 2 that not much needs to be said about it, except, perhaps, for some syntax details. First of all, note that* `PFUN{X :: TRIV, Y :: TRIV}` *has* two *parameters, both with parameter theory* `TRIV`*, but of course these two occurrences of* `TRIV` *are different and can be instantiated quite differently. This means that two different* copies *of* `TRIV` *must be used to avoid a confusion of sorts. In the first copy, the sort* `Elt` *in* `TRIV` *is automatically renamed to* `X$Elt`*, and in the second copy to* `Y$Elt`*. Furthermore, the sorts* `Pair{X,Y}`*,* `PFun{X,Y}` *and* `Rel{X,Y}` *are now* parametric *on both* `X` *and* `Y`*. Finally, the role formerly played by the supersort* `Nat < Nat?`*, where the* `undef` *constant was added in* `FUN`*, is now played by the supersort* `Y$Elt < ?{Y}`*, which is of course parametric on* `Y`*.*

```
fmod PFUN{X :: TRIV, Y :: TRIV} is
  sorts Pair{X,Y} PFun{X,Y} Rel{X,Y} ?{Y} .
  subsorts Pair{X,Y} < PFun{X,Y} < Rel{X,Y} .
  subsort Y$Elt < ?{Y} .

  op undef : -> ?{Y} [ctor] .

  vars X1 X2 : X$Elt .
  vars Y1 Y2 : Y$Elt .
  var R : Rel{X,Y} .
  var F : PFun{X,Y} .

  op [_,_] : X$Elt Y$Elt -> Pair{X,Y} [ctor] .
  op null : -> PFun{X,Y} [ctor] .   *** empty relation
  op _,_ : [Rel{X,Y}] [Rel{X,Y}] -> [Rel{X,Y}] [ctor assoc comm id: null] .

  eq [X1,Y1],[X1,Y1] = [X1,Y1] .      *** idempotency

  op def : X$Elt Rel{X,Y} -> Bool .    *** is element defined in relation

  eq def(X1,null) = false .
  eq def(X1,([X2,Y1],R)) = if X1 == X2 then true else def(X1,R) fi .

  cmb ([X1,Y1],F) : PFun{X,Y} if def(X1,F) = false .

  op _{_} : PFun{X,Y} X$Elt -> ?{Y} .   *** partial function application

  eq null{X1} = undef .
  eq ([X1,Y1],F){X2} = if X1 == X2 then Y1 else F{X2} fi .
endfm
```

Figure 2: `PFUN` module

Since a view from `TRIV` into any theory `T` is fully determined by the name `Foo` of the sort in `T` to which the sort `Elt` is mapped, Maude has a collection of such views already predefined in its standard prelude. Therefore, to define a module of partial functions from the natural numbers to the rationals we can just write:

```
fmod Nat2RaT-PFUN is
  protecting PFUN{Nat,Rat} .
endfm
```

and we can then evaluate some expressions in this module as follows:

```
reduce in Nat2RaT-PFUN : [1,1/2],[1,1/2],[3,1/7],[5,1/17],[3,1/7] .
result PFun{Nat,Rat}: [1,1/2],[3,1/7],[5,1/17]

reduce in Nat2RaT-PFUN : [1,1/2],[3,1/7],[5,1/17] {3} .
result PosRat: 1/7
```

Further Reading. Besides [28], further details on the executability conditions for a (possibly conditional) functional module can be found in: (i) for operational termination [43]; for confluence and sort-decreasingness [16, 47]; for rewriting modulo axioms B, the canonical term algebra $C_{\Sigma/M\cup E,B}$, and the agreement between mathematical and operational semantics [82, 99]. For the semantics of parameterized functional modules see [63, 45, 97].

3. System Modules

Maude system modules model concurrent systems as conditional rewrite theories [90, 19] of the form $\mathcal{R} = (\Sigma, M \cup E \cup B, R, \phi)$, where: (i) $(\Sigma, M \cup E \cup B)$ is a membership equational theory satisfying the executability conditions of a functional module (i.e., convergence), (ii) R is a set of (possibly conditional) rewrite rules specifying the system's concurrent transitions, and (iii) ϕ is a frozenness map (more on this below). In Maude, a system module for the above theory named `FOO` is declared with syntax: `mod FOO is` $(\Sigma, M \cup E \cup B, R, \phi)$ `endm`.

What is the concurrent system defined by $\mathcal{R}$? The membership equational theory $(\Sigma, M \cup E \cup B)$ defines the *states* of such a system as the elements of the algebraic data type $C_{\Sigma/M\cup E,B}$. We can call this aspect the *static* part of the specification $\mathcal{R}$. Instead, its *dynamics*, i.e., how states *evolve*, is described by the rewrite rules R, which specify the possible *local* concurrent transitions of the system thus specified. The system's concurrency is naturally modeled by the fact that in a given state $[u] \in C_{\Sigma/M\cup E,B}$ several rewrite rules in R may be applied concurrently to different subterms of u, producing several concurrent local state changes, and that rewriting logic itself models those concurrent transitions as logical deductions (see [90, 19] and the later discussion on semantics). The only restrictions imposed when applying rules in R are specified by the *frozenness map* $\phi : \Sigma \longrightarrow \mathcal{P}(\mathbb{N})$, which assigns to each operator $f\ :\ k_1 \dots k_n \to k$ in Σ the subset $\phi(f) \subseteq \{1, \dots, n\}$ of its *frozen arguments*, that is, those argument positions under which rewriting with rules in R is forbidden.

The rules in R can be unconditional rewrite rules of the form $t \to t'$, where t, t' are Σ-terms of the same kind. They are then specified in Maude with syntax `rl` t `=>` t' `.` but, by making them conditional, rules can become considerably more expressive. Conditional rules in R have the general form:

$$\texttt{crl}\, t \to t' \;\; \textit{if} \; u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m \wedge l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k \;.$$

where in Maude the $\to$ symbol is rendered in ASCII as `=>` and the conjunction $\wedge$ as `/\` and where t and t' are Σ-terms of the same kind, $u_i = v_i, 1 \leq i \leq n$ are Σ-equations, $w_i : s_i, 1 \leq i \leq m$ are memberships, and $l_i \to r_i, 1 \leq i \leq k$ are rewrite conditions understood as *reachability predicates*, that is, the arrow in them (rendered in ASCII as `=>`) should be implicitly understood in a reflexive-transitive closure sense as $l_i \to^* r_i$. Of course, in their full generality, so that, for example, new variables may appear in a rule's condition in an arbitrary manner, a conditional rule may not be executable in Maude. However, Maude allows conditional rules to have extra variables in their conditions provided they appear in a disciplined manner (spelled out in detail in [28]) that allows such extra variables to be *incrementally instantiated* by incrementally evaluating the conditions in the rule from left to right. We postpone a more detailed explanation of conditional rule evaluation until after Example 8.

Besides the syntactic requirements for a conditional rule in R having extra variables in its condition to be executable in Maude spelled out in [28], some further *executability requirements* are needed: (i) first of all, the equational part $(\Sigma, M \cup E \cup B)$ must meet the requirements of a functional module, so the (oriented) equations E should be sort-decreasing, confluent and operationally terminating modulo B; and (ii) the rules R should "commute" with the equations E modulo B in the precise sense of having the *ground coherence property.* This exactly means that if t is a ground term having a normal form $[u] \in C_{\Sigma/M \cup E,B}$, and we can perform a rewrite $t \to t'$ with a rule in R, then we can also perform a rewrite $u \to t''$ with a rule in R so that t' and t'' have the *same* normal form, say, $[w] \in C_{\Sigma/M \cup E,B}$. Ground coherence can be checked by Maude's ChC tool [47]. This allows Maude to always normalize terms with the equations E modulo B *before* performing a transition with R, under the assurance that no state transitions will ever be missed by following this strategy.

Let us explain how terms are evaluated in a system module. As pointed out at the beginning of Section 2, the key difference between an equational program (functional module) and a rewriting logic one (system module) is that evaluation to normal form of a term t in a functional module by means of the `reduce` command yields a *unique* result (determinism) under the confluence and operational termination assumptions. Instead, rewrite theories are intrinsically *non-deterministic*. What should Maude do to evaluate a term t in a system module? t can be rewritten in many different ways to many different terms, and the process may never terminate. Maude offers the following options:

- A *rule fair* sequence of rewrite steps starting from a term t can be obtained by giving the command: `rewrite` t `.` but since such a rewrite sequence may not terminate, a bound limiting the number of rewrite steps can be specified (see [28]). Rule fairness means that if more than one rule is applicable to the terms of a rewrite sequence, different rules are applied, avoiding the repetition of the very same rule for each term in a rewrite sequence.

- A *rule and position fair* sequence of rewrite steps starting from a term t can be obtained by giving the command: `frewrite` t `.` and a bound limiting the number of rewrite steps can likewise be specified (see [28]). Position fairness is similar to rule fairness but refers to the positions of a term where a rule is applicable.

- The entire, possibly infinite, state space of terms reachable from a term t by a sequence of rewrite steps can be explored with Maude's `search` command, which searches such a state space in a breadth-first manner. The general form of the command is: `search` t `=>`$\diamond$ t' `s.t.` C `.` where t' is a term pattern, so that we are looking for terms reachable from t that are instances of t' by a substitution θ, and C is an equational condition such that only reachable terms of the form $t'\theta$ such that $C\theta$ holds are selected. The $\diamond$ symbol is a place holder for the options: $\diamond =$ `1` (exactly one rewrite step), $\diamond =$ `+` (one or more steps), $\diamond =$ `*` (zero or more steps), and $\diamond =$ `!` (terminating states). Since the search may either never terminate and/or find an infinite number of solutions, two *bounds* can be added to a `search` command: one bounding the number of solutions requested, and another bounding the depth of the rewrite steps from the initial term t (see the examples in Example 9 and [28] for details). Note that Maude's `search` command provides a quite expressive form of model checking by *reachability analysis*, in addition to its LTL-based model checking.

Example 4. *Let us consider a simple example of a system module. The* `HANOI` *module in Figure 3 specifies the Towers of Hanoi puzzle, invented by the French mathematician Éduard Lucas in 1883. His story tells that in an Asian temple there are three diamond posts; the first is surrounded by sixty-four golden disks of increasing size from the bottom to the top. The monks are committed to move them from one post to another respecting two rules: only a disk can be moved at a time, and they can only be laid either on a bigger disk or on the floor. Their objective is to move all of them to the third post, and then the world will end.*

In the `HANOI` *module, the golden disks are modeled as natural numbers describing their size, and the posts are described as lists of disks in bottom-up order. In general, we have terms describing the states of the game and rules that model the moves allowed by the game. Rewriting with these rules allows going from a given initial state to other states, hopefully including the desired final state. In [28, Chapter 7] there is a collection of game examples that follow this general pattern.*

If we try to rewrite the initial puzzle setting

```
Maude> rewrite in HANOI : (0)[3 2 1] (1)[nil] (2)[nil] .
```

the command does not terminate because the disks are being moved in a loop. We can instead rewrite with a bound on the number of rewrites, like 23 in the following command example

```
mod HANOI is
  protecting NAT-LIST .
  sorts Tower Hanoi .
  subsort Tower < Hanoi .
  op (_)[_] : Nat NatList -> Tower [ctor] .
  op empty : -> Hanoi [ctor] .
  op __ : Hanoi Hanoi -> Hanoi [ctor assoc comm id: empty] .
  vars S T D1 D2 : Nat .   vars L1 L2 : NatList .

  crl [move] : (S) [L1 D1]  (T) [L2 D2]
    => (S) [L1]             (T) [L2 D2 D1]
    if D2 > D1 .
  rl [move] :  (S) [L1 D1]  (T) [nil]
    =>         (S) [L1]     (T) [D1] .
endm
```

Figure 3: `HANOI` module

```
Maude> rewrite [23] in HANOI : (0)[3 2 1] (1)[nil] (2)[nil] .
result Hanoi: (0)[3 2] (1)[1] (2)[nil]
```

Even if the example has non-terminating rewrite sequences, as shown above with the rewrite command, the number of reachable configurations is finite and the following search command terminates.

```
Maude> search in HANOI : (0)[3 2 1] (1)[nil] (2)[nil] =>* H .
.... 27 solutions
```

Indeed, the solution to the initial configuration `(0)[3 2 1] (1)[nil] (2)[nil]` *is* `(0)[nil] (1)[nil] (2)[3 2 1]` *and is found at state* 26*.*

In Section 4 we will consider again this example with the help of strategies.

3.1. Logic Programming Running Example

One of the key strengths of rewriting logic, inherited by Maude, is that a wide range of concurrent and non-deterministic systems can be naturally specified as rewrite theories and executed and analyzed as system modules in Maude. Such systems include: (i) a very wide range of concurrency models [94, 98], (ii) the executable semantic definition of concurrent programming languages [107, 123, 108], and (iii) a very wide range of logics, specified as rewrite theories using rewriting logic as a *logical framework* [83, 98]. Section 5 will give examples of how concurrent object systems can be naturally specified in Maude. Here we give an example that straddles cases (ii)–(iii) above, namely, a computational logic (Horn Logic [73]) that can at the same time be used as a programming language. Since this logic programming example uses symbolic computation in an essential manner, we will present several variants of it at various places in the paper to illustrate various Maude symbolic computation features.

Example 5 (LP-Syntax). *To define the semantics of logic programs as a system module we first specify an* `LP-SYNTAX` *functional module that imports the* `TERM` *functional module with sort* `Term` *in Example 1. An atomic predicate is defined as a* `Qid` *symbol applied to a non-empty list of terms in parentheses:*

```
  sort Predicate .
  op _‘(_‘) : Qid NeTermList -> Predicate [ctor] .
```

Since for Horn clauses we need both empty and non-empty lists of predicates, we define them in Maude as follows:

```
  pr (LIST * (sort List{X} to PredicateList,
              sort NeList{X} to NePredicateList,
              op __ to _‘,_ [prec 50])) {Predicate} .
```

We define a Horn clause using the standard symbol `:-` *for* $\Leftarrow$. *We express an axiom, e.g.* `'father('john,'peter)`, *as a Horn clause with an empty body, e.g.* `'father('john,'peter) :- nil`.

```
  sort Clause .
  op _:-_ : Predicate PredicateList -> Clause [ctor prec 60] .
```

First, we should provide some basic notion of substitution.

Example 6 (LP-Substitution Module). *Using the syntax for predicates and Horn clauses given in Example 5, we define a substitution as a partial function according to Example 3. We need to define two views from the sort* `Variable` *to* `Elt` *and from the sort* `Term` *to* `Elt`.

```
view Variable from TRIV to TERM is
  sort Elt to Variable .
endv

view Term from TRIV to TERM is
  sort Elt to Term .
endv
```

And we import the parametric module `PFUN` *instantiated to the views* `Variable` *and* `Term` *but rename sort* `PFun{Variable,Term}` *into* `Substitution`, *operator* `_,_` *for combining bindings into* `_;_`, *and operator* `[_,_]` *into the more standard syntax* `_->_` *for substitution bindings.*

```
fmod LP-SUBSTITUTION is
  protecting (PFUN * (op _,_ to _;_,
                      op [_,_] to _->_)) {Variable,Term}
                * (sort PFun{Variable,Term} to Substitution,
                   sort Pair{Variable,Term} to Binding) .
endfm
```

Now, we are able to provide some basic syntactic unification functionality.

Example 7 (LP-Unification Module). *Continuing Example 6, module* `LP-UNIFICATION` *in Figure 4 defines a syntactic unification procedure, without the occurs check, for both predicates and terms that will be used by our Horn logic interpreter in Example 8 below. Note that the failure to unify is represented by an expression at the kind* `[Substitution]`.

Example 8 (LP System Module). *Using the LP syntax defined in Example 5 and the unification algorithm in Example 7, the system module* `LP-SEMANTICS` *in Figure 5 defines the semantics of logic programs and providing an interpreter with a breadth-first strategy. We first define logic programming*

```
fmod LP-UNIFICATION is
  protecting LP-SYNTAX .
  protecting LP-SUBSTITUTION .

  op unify : Predicate Predicate Substitution -> [Substitution] .
  eq unify(F(NeTL1), F(NeTL2), S)
   = unify(NeTL1, NeTL2,S) .

  vars C F : Qid .
  var V : Variable .
  vars NeTL1 NeTL2 : NeTermList .
  var NVT : NvTerm .
  var S : Substitution .
  vars T T1 T2 : Term .

  op unify : NeTermList NeTermList Substitution -> [Substitution] .
  eq unify(C, C, S) = S .
  eq unify(V, T1, (V -> T2) ; S) = unify(T1, T2, (V -> T2) ; S) .
 ceq unify(T1, V, (V -> T2) ; S) = unify(T1, T2, (V -> T2) ; S) if not T1 ::
      Variable .
 ceq unify(V, T, S) = (V -> T) ; S if not def(V, S) .
 ceq unify(NVT, V, S) = (V -> NVT) ; S if not def(V, S) .
  eq unify(F[NeTL1], F[NeTL2], S) = unify(NeTL1, NeTL2, S) .
 ceq unify((T1,NeTL1), (T2, NeTL2), S)
   = unify(NeTL1, NeTL2, unify(T1, T2, S))
   if unify(T1, T2, S) :: Substitution .
endfm
```

Figure 4: `LP-UNIFICATION` module

configurations to hold the execution state, essentially composed of a predicate list and a substitution `PL $ S` *that represents the pending objectives and the bindings carried from already executed clauses. The execution of a query predicate w.r.t. a logic program will be defined as transition rules transforming a configuration. Notice that the final configuration is extended with a natural number used for renaming clauses before unification, and the logic program. Its semantics is defined by a single conditional rule that invokes a variable renaming function* `rename` *not shown here. Finally, we add an initialization function to evaluate a predicate list* `PL` *in* `Pr`*.*

Since the condition in the above conditional rule only involves equations, its incremental evaluation is exactly the same as if it were the condition of a conditional equation or membership in a functional module. Therefore, we can use this example to explain the incremental evaluation of conditions in all cases. After this explanation we will briefly summarize the more general case of a conditional rule that also has rewrite conditions. To syntactically indicate the fact that extra variables appear in a condition, the equality sign `:=` is used instead of the usual sign `=`. At the left of the `:=` sign a *pattern with new variables* is placed. Here three such patterns are given, one for each condition, namely, the term `P3 :- PL3` and the variables `S2` and `N2`. Operationally, the substitution instantiating the new variables in these three patterns is obtained by *incrementally* (one condition at a time, from left to right): (i) reducing to normal form with the module's *equations and memberships* the substitution instance of the condition's righthand side. But instance under *which* substitution? For the first condition

```
mod LP-SEMANTICS is
  protecting LP-UNIFICATION .
  sort Configuration .
  op <_|_$_|_> : Nat PredicateList Substitution Program -> Configuration .

 crl [clause] :
   < N1 | P1, PL1 $ S1 | Pr1 ; P2 :- PL2 ; Pr2 >
   => < N2 | PL3, PL1 $ S2 | Pr1 ; P2 :- PL2 ; Pr2 >
   if P3 :- PL3 := rename(P2 :- PL2, N1)
   /\ S2 := unify(P1, P3, S1)
   /\ N2 := max(N1, last$(P3 :- PL3)) .

  ...

  op <_|_> : PredicateList Program -> Configuration .
  eq < PL | Pr > = < last$(PL) | PL $ null | Pr > .
endm
```

Figure 5: `LP-SEMANTICS` module

—here the condition `P3 :- PL3 := rename(P2 :- PL2,N1)`— its righthand side variables —here `P2`, `PL2` and `N1`— must always appear in the conditional rule's lefthand side. Therefore, the righthand side `rename(P2 :- PL2,N1)` is instantiated by the *matching substitution* θ_0 with wich we have instantiated the rule's lefthand side to attempt a rewrite. Suppose, for example, that in θ_0 `P2` was instantiated to `'p(x{1})`, `PL2` to `nil`, and `N1` to 5. Then we *reduce* to normal form with the module's equations the instance by θ_0 of the condition's righthand side `rename(P2 :- PL2,N1)`, that is, the term `rename('p(x{1}) :- nil,5)`. Since the rename function just renames the variables to fresh ones above the given index, we will get the result `'p(x{6}) :- nil`. Then, (ii) we now incrementally extend θ_0 by instantiating the new variables in the condition's lefthand side `P3 :- PL3` by *matching* `'p(x{6}) :- nil` against it. That is, `P3` is instantiated to `'p(x{6})` and `PL3` is instantiated to `nil`, thus getting an extended substitution $\theta_0 \uplus \theta_1$. But this extended substitution can now instantiate all the variables in the righthand side of the *second* condition `S2 := unify(P1,P3,S1)`, so that we can again reduce the instance by $\theta_0 \uplus \theta_1$ of its righthand side to normal form and then instantiate `S2` to *that* normal form to obtain a new extended substitution $\theta_0 \uplus \theta_1 \uplus \theta_2$. We then proceed in the same manner to evaluate the third condition using $\theta_0 \uplus \theta_1 \uplus \theta_2$ and in that way we finally get an extended substitution $\theta_0 \uplus \theta_1 \uplus \theta_2 \uplus \theta_3$ with which we can now instantiate the conditional rule's righthand side, thus ending the conditional rule's evaluation. Note that the second and third condition exactly correspond to `where` clauses in some functional languages.

Since rewrite conditions are not used in this example (but are used in Figure 13 below), we briefly explain their use and execution. Further details can be found in [28]. In general, a reachability condition $l \to r$ may have extra variables in its righthand side r. It succeeds for a substitution instance $l\theta$ (where, given the incremental way conditions are evaluated, θ will extend the original substitution θ_0 obtained by matching the rule's lefthand side) iff $l\theta$ can be rewritten in 0, 1 or more steps with the module's rules R to a term whose normal form by the module's equations E and memberships M is a substitution instance of r up to B-equality. In this way, if $l \to r$ was the kth condition, we obtain an extended substitution $\theta \uplus \theta_k$ that we then use to evaluate condition $k+1$ or, if no more conditions are left, to instantiate the rule's righthand side. Note that, in general, the equalities, memberships, and reachability conditions in a rule's condition need not appear in any particular order, provided that the variables appearing in either the right side of an equational condition $u := v$ or the left side of a rewrite condition $l \to r$ have already appeared in *previous* conditions and/or in the rule's lefthand side.

We can now evaluate some logic programs using our semantic definition as a breadth-first logic programming interpreter. An interesting question is what pattern t' to use in a search from an initial state `< PL | Pr >`, where `PL` is the list of predicates that we wish to find a solution for, and `Pr` is the given Horn logic program. The answer is that we should look for a pattern of the form: `< N | nil $ S1 | Pr >`, indicating that the list of objectives has become empty and therefore the substitution `S1` is a solution. Therefore, calls to our interpreter will have the general form: `search < PL | Pr > =>* < N | nil $ S1 | Pr > .`

Example 9 (Search LP-evaluation). *Consider the following logic program defining several family relations between Jane, Mike, Sally, John, and Tom.*

```
mother(jane, mike) .
mother(sally, john) .
father(tom, sally) .
father(tom, erica) .
father(mike, john) .
sibling(X1, X2) :- parent(X3,X1), parent(X3,X2)
parent(X1, X2) :- father(X1,X2)
parent(X1, X2) :- mother(X1,X2)
relative(X1, X2) :- parent(X1,X3), parent(X3,X2)
relative(X1, X2) :- sibling(X1,X3), relative(X3,X2)
```

This logic program is expressed using the syntax of Example 5 as follows:

```
'mother('jane, 'mike) :- nil ;
'mother('sally, 'john) :- nil ;
'father('tom, 'sally) :- nil ;
'father('tom, 'erica) :- nil ;
'father('mike, 'john) :- nil ;
'sibling(x{1}, x{2}) :- 'parent(x{3}, x{1}), 'parent(x{3}, x{2}) ;
'parent(x{1}, x{2}) :- 'father(x{1}, x{2}) ;
'parent(x{1}, x{2}) :- 'mother(x{1}, x{2}) ;
'relative(x{1}, x{2}) :- 'parent(x{1}, x{3}),'parent(x{3}, x{2}) ;
'relative(x{1}, x{2}) :- 'sibling(x{1}, x{3}),'relative(x{3}, x{2})
```

We can now evaluate different initial calls for this program by specifying specific lists of atoms that we seek a solution for. We can do so by searching for a set of configurations reachable from the given initial call that contains a solution. As already explained, a solution will correspond to a configuration of the form `nil $ Sub`*. Depth and solution bounds are automatically provided by Maude's* `search` *command if desired. In order to simplify the presentation, we have abbreviated our program to* `P` *and we do not show all the bindings returned by the search command, just the binding associated to the logic-programming computed substitution.*

First, we can ask whether Sally and Erica are sisters; the associated reachability graph is finite and no bound is needed.

```
Maude> search < 'sibling('sally, 'erica)  | P > =>* < N | nil $ S1 | Pr > .
Solution 1 (state 7)
S1 --> (x{1} -> 'sally) ; (x{2} -> 'erica) ; (x{3} -> x{4}) ;
       (x{4} -> 'tom) ; (x{5} -> 'sally) ; (x{6} -> x{4}) ; x{7} -> 'erica
```

Who are the siblings of Erica? Sally and herself.

```
Maude> search < 'sibling(x{1},'erica) | P > =>* < N | nil $ S1 | Pr > .
Solution 1 (state 19)
S1 --> (x{1} -> x{2}) ; (x{2} -> x{6}) ; (x{3} -> 'erica) ; (x{4} -> x{5}) ;
       (x{5} -> 'tom) ; (x{6} -> 'sally) ; (x{7} -> x{5}) ; x{8} -> 'erica

Solution 2 (state 20)
S1 --> (x{1} -> x{2}) ; (x{2} -> x{6}) ; (x{3} -> 'erica) ; (x{4} -> x{5}) ;
       (x{5} -> 'tom) ; (x{6} -> 'erica) ; (x{7} -> x{5}) ; x{8} -> 'erica
```

How many possible siblings are there? Sally and Sally, Sally and Erica, Erica and Sally, Erica and Erica, John and John, and Mike and Mike.

```
Maude> search < 'sibling(x{1},x{2}) | P > =>* < N | nil $ S1 | Pr > .
Solution 1 (state 19)
S1 --> (x{1} -> x{3}) ; (x{2} -> x{4}) ; (x{3} -> x{7}) ; (x{4} -> x{9}) ;
       (x{5} -> x{6}) ; (x{6} -> 'tom) ; (x{7} -> 'sally) ; (x{8} -> x{6}) ;
       x{9} -> 'sally
...

Solution 7 (state 25)
S1 --> (x{1} -> x{3}) ; (x{2} -> x{4}) ; (x{3} -> x{7}) ; (x{4} -> x{9}) ;
       (x{5} -> x{6}) ; (x{6} -> 'sally) ; (x{7} -> 'john) ; (x{8} -> x{6}) ;
       x{9} -> 'john
```

Seven solutions are given. Are Jane and John relatives? Yes

```
Maude> search < 'relative('jane,'john) | P > =>* < N | nil $ S1 | Pr > .
Solution 1 (state 11)
S1 --> (x{1} -> 'jane) ; (x{2} -> 'john) ; (x{3} -> x{5}) ; (x{4} -> 'jane) ;
       (x{5} -> 'mike) ; (x{6} -> x{5}) ; x{7} -> 'john
```

Who are the relatives of John? Tom and Jane.

```
Maude> search [2] < 'relative(x{1},'john) | P > =>* < N | nil $ S1 | Pr > .
Solution 1 (state 28)
S1 --> (x{1} -> x{2}) ; (x{2} -> x{5}) ; (x{3} -> 'john) ; (x{4} -> x{6}) ;
       (x{5} -> 'tom) ; (x{6} -> 'sally) ; (x{7} -> x{6}) ; x{8} -> 'john

Solution 2 (state 29)
S1 --> (x{1} -> x{2}) ; (x{2} -> x{5}) ; (x{3} -> 'john) ; (x{4} -> x{6}) ;
       (x{5} -> 'jane) ; (x{6} -> 'mike) ; (x{7} -> x{6}) ; x{8} -> 'john
```

This last call produces an infinite search and we must restrict the search, by asking for two solutions only.

3.2. Initial Model Semantics and Parameterization

What is the mathematical meaning of a system module `mod FOO is` $(\Sigma, M \cup E \cup B, R, \phi)$ `endm`, that is, of a rewrite theory $\mathcal{R} = (\Sigma, M \cup E \cup B, R, \phi)$? It is its *initial model* $\mathcal{T}_{\mathcal{R}}$ in the class (indeed, category) of all models of $\mathcal{R}$ [90, 19]. What does $\mathcal{T}_{\mathcal{R}}$ look like? Why, of course, it models a concurrent system! Its *states*, as already pointed out, are the normal forms $[w] \in C_{\Sigma/M \cup E,B}$ in the canonical term algebra of $(\Sigma, M \cup E \cup B)$. What about its transitions? $\mathcal{T}_{\mathcal{R}}$ provides a *true concurrency semantics*. That is, not only are one-step transitions modeled: concurrent *computations* are also modeled. Furthermore, $\mathcal{T}_{\mathcal{R}}$ provides a notion of *equivalence* between two different descriptions of the *same* concurrent computation. Mathematically, what all this means is that for each kind $[s]$ in Σ the concurrent computations of $\mathcal{R}$ form a *category* [90, 19], whose objects/states are precisely the normal forms in $C_{\Sigma/M\cup E,B,[s]}$. But how are concurrent computations modeled? They coincide with rewriting logic *proofs* in $\mathcal{R}$, with the category structure providing a natural notion of *proof equivalence*. This is what one should expect, since any declarative programming language worth its salt should satisfy the equivalence:

$$\textit{Computation} = \textit{Deduction}$$

and of course this is exactly what also happens at the equational logic level of functional modules, where the initial algebra $T_{\Sigma/M\cup E\cup B}$ is built up out of the proof theory of membership equational logic [95].

What about *parameterized* system modules? They are completely analogous to parameterized functional ones. That is, a *parameterized system module* `M{X1 :: T1, ... , Xm :: Tm}` is a rewrite theory with `T1`, ..., `Tm` its parameter theories and is instantiated by views `V1`, ..., `Vm` from `T1`, ..., `Tm` to `T'1`, ..., `T'm` to the instance `M{V1, ... , Vm}`. The only new feature is that, although often the parameter theories are functional, some of the theories among the `T1`, ..., `Tm` may be *system theories*, i.e., theories of the form `th FOO is` $(\Sigma, M \cup E \cup B, R, \phi)$ `endth`, where, as done for functional theories, the rewrite theory $\mathcal{R} = (\Sigma, M \cup E \cup B, R, \phi)$ is given a "loose semantics," so that actual parameter instances of this formal parameter range over the class of all models of $\mathcal{R}$. Of course, if the parameter theory `Ti` is a rewrite theory, then the target theory `T'i` should also be a rewrite theory, and the view `Vi : Ti` $\to$ `T'i` is a *theory interpretation* between rewrite theories.

Further Reading. Besides [28], the following references may be helpful: (i) for the semantics of rewrite theories [90, 19]; for the modeling of concurrent systems, programming languages, and logical systems in rewriting logic [98, 94, 107, 123, 108, 83]; (iii) for the ground coherence property and how to check it [47]; (iv) for automatically transforming a rewrite theory into a semantically equivalent one that is ground coherent [100]; and for an even more general notion of rewrite theory well suited for symbolic computation [100].

4. The Maude Strategy Language

As described in Section 3, rule rewriting is a highly non-deterministic process: in general, at every step many rules could be applied at various places. A finer control on rule application may sometimes be desirable and, although this may be accomplished in different ways, a strategy language has been proposed [84, 55, 85] as a specification layer above rewrite theories, providing a cleaner way to control the rewriting process and respecting the *separation of concerns* principle. That is, the rewrite theory is not modified in any way: strategies provide an additional specification level above that of rules, so that the same system module may be executed according to different strategy specifications. The design of this strategy language was influenced, among others, by ELAN [15] and Stratego [18].

The usual Maude command for rewriting with strategies is:

srewrite [⟨*Bound*⟩] in ⟨*ModuleName*⟩ : ⟨*Term*⟩ by ⟨*StrategyExpr*⟩ .

It rewrites the term according to the given strategy, which need not be deterministic, and prints all the results. Like in the standard rewriting commands, the module in which the rewriting is performed can be optionally selected with the `in` keyword, the command can be shortened to `srew`, and a bound on the number N of solutions to be obtained can be imposed by declaring the optional [N] just after the command keyword. For example, going back to the Towers of Hanoi example introduced in Section 3 (page 18), we can consider a basic strategy which is just rule application, invoked by mentioning the rule label, as follows:

```
Maude> srew [3] in HANOI : (0)[3 2 1] (1)[nil] (2)[nil] using move .

Solution 1
rewrites: 1
result Hanoi: (0)[3 2] (1)[1] (2)[nil]

Solution 2
rewrites: 2
result Hanoi: (0)[3 2] (1)[nil] (2)[1]

No more solutions.
rewrites: 2
```

The two results of applying the `move` rule to the initial term are shown, and the interpreter tells us that there are no more solutions, because we have requested a bound of 3 exceeding the possible one-step moves. In order to use more elaborate strategies we will need to introduce the complete strategy language, but before doing that, let us note that in this way we have a standard approach to model a game: terms represent states, rules represent allowed moves, and the strategy language can be used to model a (winning) strategy for solving or playing the game.

As we have seen, rule application is the basic building block of the strategy language. Besides the rule label, further restrictions can be imposed; its most general syntax has the form:

$$\textit{label}\ \texttt{[}X_1\ \texttt{<-}\ t_1, \ldots, X_n\ \texttt{<-}\ t_n\texttt{]}\ \texttt{\{}\alpha_1, \ldots, \alpha_m\texttt{\}}$$

All rules with the given label and exactly m rewriting conditions will be executed. Rewriting in these conditions is controlled by the strategies $\alpha_1, \ldots, \alpha_m$ between curly brackets, which must be omitted if $m = 0$. Between square brackets, we can optionally specify an initial ground substitution to be applied to both sides of the rule.

The other basic element of the language are the tests, which can be used for testing a condition on the subject term. Their syntax has the form: `match` P `s.t.` C where P is a pattern and C is an equational condition. On a successful match and condition check, the result is the initial term; otherwise, the test does not provide any solution. For example, we can check whether the Towers of Hanoi puzzle is solved with tests:

```
Maude> srew (0)[nil] (1)[nil] (2)[3 2 1] using
            match (N)[3 2 1] H s.t. N =/= 0 .

Solution 1
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
```

We present now various combinators that build more complex strategies out of rule applications and tests. The concatenation α`;`β executes the strategy α and then the strategy β on each α result. The disjunction or alternative α`|`β executes α or β; in other words, the results of α`|`β are both those of α and those of β. The iteration α`*` runs α zero or more times consecutively. These combinators resemble similar constructors for regular expressions. The empty word and empty language constants are here represented by the `idle` and `fail` operators; the result of applying `idle` is always the initial term, while `fail` generates no solution.

We say that a strategy *fails* when no solution is obtained. Remember that failures can happen in less explicit situations: when a rule cannot be applied to the term, when a test fails, etc. Thus, strategies can explore rewriting paths that will later be discarded, i.e., they can make tentative attempts.

Conditionals are written α `?` β `:` γ. It executes α and then β on its results, but if α does not produce any, it executes γ on the initial term. That is, α is the *condition*; β the *positive branch*, which applies to the results of α; and γ the *negative branch*, which is applied only if α fails. Some common patterns are defined as derived operators with their own names such as:

- The `or-else` combinator is defined by α `or-else` $\beta \equiv \alpha$ `? idle :` β. It executes β only if α has failed.
- The *negation* is defined as `not(`α`)` $\equiv \alpha$ `? fail : idle`. It fails when α succeeds and succeeds as an `idle` when α fails.
- The *normalization operator* α`!` $\equiv \alpha$`* ; not(`α`)` applies α until it cannot be further applied.

The *match and rewrite* operator `matchrew` restricts the application of a strategy to a specific subterm of the subject term. Moreover, we can use it to obtain information about the subject term by means of pattern matching or equational calculations, binding new variables if necessary; such data can then be used to parameterize its substrategies. Its syntax is

`matchrew` $P(X_1, \ldots, X_n)$ `s.t.` C `by` X_1 `using` $\alpha_1, \ldots, X_n$ `using` α_n

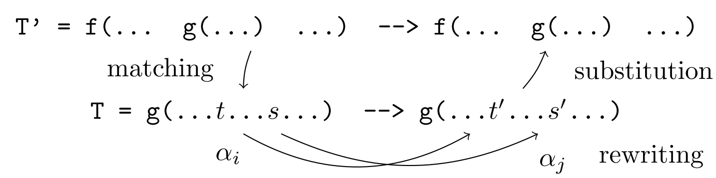

Figure 6: Behavior of `matchrew`

```
mod HANOI-AUX is
  protecting SET{Nat} .
  op third : Nat Nat Nat -> Nat .
  var N M K : Nat .
  ceq third(N, M) = K if N, M, K := 0, 1, 2 .
endm
```

Figure 7: `HANOI-AUX` module

where P is a pattern with variables $X_1, \ldots, X_n$ among others, and C is an optional equational condition. The `using` clauses associate variables in the pattern, which are matched by subterms of the matched term, with strategies that will be used to rewrite them. These variables must be distinct and must appear in the pattern. The semantics of this operator is illustrated in Figure 6. All matches of the pattern in the subject term are tried, checking the condition if any. If none succeeds, the strategy fails. Otherwise, for each match of the main pattern the subterms matching each X_i are extracted, rewritten according to α_i in *parallel*, and finally the term is reassembled with the results of rewriting each subterm in the place of the original subterm.

Strategies can be given a name and are defined in strategy modules. Named strategies make the use and description of complex strategies more convenient, and they extend the expressiveness of the language by means of recursive and mutually recursive strategies. As for functional and system modules, a *strategy module* is declared in Maude using the keywords

smod $\langle$*ModuleName*$\rangle$ **is** $\langle$*DeclarationsAndStatements*$\rangle$ **endsm**

A typical strategy module imports the system module whose rewriting it will control, declares some strategies and specifies their definitions.

To complete the Towers of Hanoi example, we define first the operator `third` which gives the third of two posts in an auxiliary functional module `HANOI-AUX` (Figure 7), and then define in the strategy module `HANOI-SOLVE` in Figure 8 a recursive strategy `moveAll` with three arguments (two posts and one number of disks), which we can use to solve the puzzle.

We can now repeat the above rewrite as follows:

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using moveAll(0, 2, 3) .

Solution 1
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
```

```
smod HANOI-SOLVE is
  protecting HANOI-RULES .
  protecting HANOI-AUX .

  strat moveAll : Nat Nat Nat @ Hanoi .

  var S T C : Nat .

  sd moveAll(S, S, C) := idle .
  sd moveAll(S, T, 0) := idle .
  sd moveAll(S, T, s(C)) := moveAll(S, third(S, T), C) ;
                            move[S <- S, T <- T] ;
                            moveAll(third(S, T), T, C) .
endsm
```

Figure 8: `HANOI-SOLVE` module

Although strategies control and restrict rewriting, they do not make the process deterministic. A strategy may allow multiple rewriting paths, produced by alternative local decisions that may appear during its execution. The `rewrite` command solves this non-determinism by choosing a single alternative using a fixed criterion. On the contrary, the `search` command explores all the possible rule applications looking for a term matching the given goal. The `srewrite` command coincides with `search` in the exhaustive exploration of the alternatives, looking, in this case, for strategy solutions. Hence, `srewrite` can be seen as a search for solutions, not in the complete rewriting tree of `search`, but in a subtree pruned by the effect of the strategy. How this tree is explored has relevant implications for the command output and performance.

The `srewrite` command explores the rewriting graph following a fair policy which ensures that all solutions that are reachable in a finite number of steps are eventually found, unless the interpreter runs out of memory. Without being a breadth-first search, multiple alternative paths are explored in parallel. An alternative rewriting command `dsrewrite` (*depth strategic rewrite*) explores the strategy rewriting graph in depth. It is written like a `srewrite` command but with a different starting keyword, which can be abbreviated to `dsrew`.

`dsrewrite [`⟨*Bound*⟩`] in` ⟨*ModuleName*⟩ `:` ⟨*Term*⟩ `by` ⟨*StrategyExpr*⟩ `.`

The disadvantage is that the depth-first exploration is incomplete, because it could go down an infinite branch before finding some reachable solutions, which would be missed; see next section for an example. The advantage is that `dsrewrite` can be more efficient than `srewrite`.

4.1. Logic Programming Running Example

In the same way that we can use the Maude strategy language to specify strategies to solve games, like the previous Towers of Hanoi example, another typical application of strategies is the execution of operational semantics for programming languages, which are specified by means of rules which require in many cases to be executed in a specific way. Again, language expressions become terms in its Maude representation, operational semantics rules become rewrite rules, and strategies are used to control rewriting in the appropriate way. For example, the Maude strategy language has been used in this sense in order to define the operational semantics for the parallel functional programming language Eden [72, 87], and also in the definition of modular structural operational semantics with strategies [17]. In this section, the strategy language is used to define the semantics of a logic programming language similar to Prolog [33], continuing with Examples 5 and 8 in Section 3.1. Strategies will be used to discard failed proofs, to enforce the Prolog search strategy, and to implement advanced features like negation. Although it is also possible to implement the Prolog cut in this way, this feature will not be shown here (see [24]).

The `rewrite` command is not useful as a logic programming interpreter because it simply explores a single rewriting path, thus a single proof path. This is clearly not enough to show multiple solutions, but it may also be insufficient to find even a single one. An admissible logic programming interpreter must consider all possible proof paths and be able to resume them when the execution arrives to a dead path. In Example 8 we used the `search` command as a possible solution. Here, strategies will be used instead.

First, we define an auxiliary predicate `isSolution` in module `LP-EXTRA` in Figure 9 to decide whether a given configuration is a solution. We then define in the strategy module `PROLOG` in Figure 10 the recursive strategy `solve` that applies the `clause` rule in Example 8 until a solution is found, and rejects any rewriting path that does not end in one. The exhaustive search of the `srewrite` command shows all reachable solutions for the initial predicate.

```
mod LP-EXTRA is
  protecting LP-SEMANTICS .
  op isSolution : Configuration -> Bool .
  var N : Nat .           var S : Substitution .
  var Pr : Program .      var Conf : Configuration .
  eq isSolution(< N | nil $ S | Pr >) = true .
  eq isSolution(Conf) = false [owise] .
endm
```

Figure 9: `LP-EXTRA` module

```
smod PROLOG is
  protecting LP-EXTRA .
  strat solve @ Configuration .
  var Conf : Configuration .
  sd solve := match Conf s.t. isSolution(Conf)
                       ? idle : (clause ; solve) .
endsm
```

Figure 10: `PROLOG` module

Now, the `solve` strategy can be applied to the previous examples:

```
Maude> srew < 'parent('tom, x{1}) | family > using solve .

Solution 1
rewrites: 453
```

```
mod PL-SIMPLIFIER-BASE is
  including LP-SEMANTICS .
  sort VarSet .
  subsort Variable < VarSet .
  op empty : -> VarSet .
  op _;_ : VarSet VarSet -> VarSet [ctor assoc comm id: empty] .
  ....
  op occurs : Configuration -> VarSet .
  op simplify : Substitution VarSet -> Substitution .
  op solution : Substitution -> Configuration [ctor format (g! o)] .

  var N : Nat .          var S : Substitution .
  var Pr : Program .     var VS : VarSet .

  rl [solution] : < N | nil $ S | Pr >
                => solution(simplify(S, VS)) [nonexec] .
endm
```

Figure 11: PL-SIMPLIFIER-BASE module (notice the ellipsis)

```
result Configuration:
  < 3 | nil $ x{1} -> x{3} ; x{2} -> 'tom ; x{3} -> 'sally
      | (omitted) >

Solution 2
rewrites: 489
result Configuration:
  < 3 | nil $ x{1} -> x{3} ; x{2} -> 'tom ; x{3} -> 'erica
      | (omitted) >

No more solutions.
rewrites: 605
```

As the example above shows, the resulting configurations are not easily readable: they are overloaded with intermediate data like the full program, which has been omitted here, and mappings on variables that do not occur in the initial predicate. To display solutions in a more readable form, we will use a wrapper strategy. First, some auxiliary functions are defined in the PL-SIMPLIFIER-BASE system module in Figure 11 along with a solution rule which restricts the substitution to the variables in the given set, after resolving them by transitivity.

The strategy wsolve in the strategy module PROLOG-SIMPLIFIER in Figure 12 records the variables that occur in the initial configuration predicate, then executes the previous solve strategy, and finally applies the solution rule with the initial variable set, thus restricting the substitution to those variables.

Now, the previous example can be rerun with wsolve, obtaining more concise and clearer answers.

```
Maude> srew < 'parent('tom, x{1}) | family > using wsolve .

Solution 1
rewrites: 511
result Configuration: solution(x{1} -> 'sally)

Solution 2
rewrites: 569
result Configuration: solution(x{1} -> 'erica)
```

```
smod PROLOG-SIMPLIFIER is
  protecting PL-SIMPLIFIER-BASE .
  protecting PROLOG .

  strat wsolve @ Configuration .

  var Conf : Configuration .   var VS : VarSet .

  sd wsolve := matchrew Conf s.t. VS := occurs(Conf)
                 by Conf using (solve ; solution[VS <- VS]) .
endsm
```

Figure 12: PROLOG-SIMPLIFIER module

```
No more solutions.
rewrites: 641
```

We can also observe that the order in which solutions appear depends on the way the rewriting tree is explored. With the `dsrewrite` command the results will appear in the same order as in Prolog, because both explore the derivation tree in depth. However, the `srewrite` command will often obtain shallower solutions first.

```
Maude> dsrew < 'p(x{1}) | 'p(x{1}) :- 'q(x{1}) ; 'p('a) :- nil ;
                             'q('b) :- nil > using wsolve .

Solution 1
rewrites: 82
result Configuration: solution(x{1} -> 'b)

Solution 2
rewrites: 111
result Configuration: solution(x{1} -> 'a)

No more solutions.
rewrites: 117

Maude> srew < 'p(x{1}) | 'p(x{1}) :- 'q(x{1}) ; 'p('a) :- nil ;
                           'q('b) :- nil > using wsolve .

Solution 1
rewrites: 105
result Configuration: solution(x{1} -> 'a)

Solution 2
rewrites: 117
result Configuration: solution(x{1} -> 'b)

No more solutions.
rewrites: 95
```

The benefit of using `srewrite` is that all reachable solutions are shown. In Prolog and with `dsrewrite` some of them may be hidden by going down a non-terminating branch.

```
Maude> dsrew < 'p(x{1}) | 'p(x{1}) :- 'p(x{1}) ; 'p('a) :- nil >
         using wsolve .

Debug(1)> abort .  *** non-terminating

Maude> srew < 'p(x{1}) | 'p(x{1}) :- 'p(x{1}) ; 'p('a) :- nil >
         using wsolve .
```

```
mod LP-EXTRA-NEGATION is
  including LP-EXTRA .

  var Q : Qid .                  var PL : PredicateList .
  var Conf : Configuration .     var NeTL : NeTermList .

  crl [negation] :
    < N | '\+(Q[NeTL]), PL $ S | Pr >
    => < N | PL $ S | Pr >
    if < N | Q(NeTL) $ S | Pr > => Conf .
endm
```

Figure 13: `LP-EXTRA-NEGATION` module

```
smod PROLOG-NEGATION is
  protecting LP-EXTRA-NEGATION .

  strat solve-neg @ Configuration .

  var Conf : Configuration .

  sd solve-neg := match Conf s.t. isSolution(Conf) ? idle :
    ((clause | negation{not(solve-neg)}) ; solve-neg) .
endsm
```

Figure 14: `PROLOG-NEGATION` module

```
Solution 1
rewrites: 109 in 0ms cpu (3ms real) (~ rewrites/second)
result Configuration: solution(x{1} -> 'a)

Debug(1)> abort .  *** non-terminating
```

We consider now the negation feature. In logic programming, the concept of negation is complicated: facts and predicates express positive knowledge, so we could either explicitly assert what is false or assume that any predicate that cannot be derived from the facts is considered as false. The last approach is known as *negation as failure*: the negation of a predicate holds if the predicate cannot be proved, no matter the values its variables take. This cannot be expressed with Horn clauses but can be easily implemented using strategies and an extra rewriting rule, added to `LP-EXTRA` system module in Figure 9. Like in ISO Prolog, in the system module `LP-EXTRA-NEGATION` in Figure 13 negation is represented as a normal predicate[6] named `\+`, which can be seen as the ASCII representation of the *not provable* sign $\nvdash$.

The `negation` rule only removes the negation predicate from the objective lists if its rewriting condition holds. By its own semantics, negation never binds variables, so the substitution remains unchanged. The initial term of the rewriting condition contains the negated predicate as its only objective. Whether

[6] Hence, its argument is written as a term, i.e., brackets should be used instead of parentheses.

this term can be rewritten to a solution configuration determines whether the negated predicate can be satisfied. Hence, we need to control the condition with a strategy that fails whenever that happens. We do so in the strategy module `PROLOG-NEGATION` in Figure 14. The strategy is similar to the original `solve` strategy, but the `negation` rule can be applied when a negated predicate is on top of the objective list. The strategy `not(solve-neg)` fails if `solve-neg` finds a solution for the negated predicate. Otherwise, it behaves like an `idle`, triggering the rule application. Thus, it is a suitable strategy for the rewriting condition.

We can illustrate the negation feature using the family tree example. Again, to obtain simplified results, we use the strategy `wsolve-neg`, defined from `solve-neg` as `wsolve` was defined from `solve` in `PROLOG-SIMPLIFIER` (a generic implementation is possible using parameterized strategy modules, see [24]). A predicate `'no-children` claims that someone does not have descendants:

```
Maude> srew < 'no-children('erica) | family ;
  'no-children(x{1}) :- '\+('parent[x{1}, x{2}]) > using solve-neg .

Solution 1
rewrites: 887
result Configuration: solution(empty)

No more solutions.
rewrites: 887

Maude> srew < 'no-children('mike) | family ;
  'no-children(x{1}) :- '\+('parent[x{1}, x{2}]) > using solve-neg .

No more solutions.
rewrites: 894
```

As mentioned at the beginning of this section, the Prolog cut has also been implemented using strategies in this way (details will appear in [24]). These examples show that our framework can be used to fully realize Kowalski's motto "Algorithm = Logic + Control" [78], putting into practice the separation of concerns allowed by our strategy language. The logic of a system (be it a game or a language operational semantics or whatever) is declaratively specified by means of equations and rules. The concrete, controlled way of executing such rules, when desired or when necessary, is written as a strategy on top of them. The separation between logic and control allows us to have different controls for the same logic, like, for example, having a logic programming interpreter which is complete because it uses breadth-first search instead of the standard depth-first search used by Prolog.

Further Reading. The Maude strategy language was introduced in a series of conference papers [84, 55, 85] and applied in different areas, including operational semantics (see [72, 87, 17], among others). More current work includes the extension of the language to include parameterized strategies, and the development of model-checking techniques for systems controlled by strategies. More examples and further details can be found in [24].

5. Object-Based Programming

In the design of distributed systems, the motto *think globally, act locally* expresses the essential philosophy. Each object in a distributed system has only a quite limited partial view of the global state and can only *act locally*, typically by communicating with other objects and changing its local state, to achieve some *global* system goals. A well-designed distributed system uses such local actions to achieve a desired global behavior.

Rewriting logic [90] is precisely a logic to express local actions in a concurrent system by means of rewrite rules. As explained in the Introduction, the concurrent systems that can be specified in rewriting logic, and therefore in Maude, can be widely different. In this sense, rewriting logic and Maude are completely ecumenical, since they do not prescribe any particular style of concurrent, synchronous or asynchronous, interaction at all: any such style can be supported. Nevertheless, the overwhelming majority of distributed systems and communication protocols can be most naturally expressed as made up of *concurrent objects* having their own *local states* that communicate with each other by *message passing*. Given the great importance of distributed object-based systems, Maude provides special support for such systems in the following ways: (i) a special notation is supported both in Maude and in its Full Maude extension; (ii) the `frewrite` command, when applied to object-based systems, provides an object and message fair rewriting strategy for simulation purposes; (iii) several kinds of *external objects* allow regular Maude objects to interact with the external world; and (iv) using such external objects, a Maude object-based distributed system design can be seamlessly transformed (within Maude) into an actual distributed system *implementation*. We discuss all these aspects in this section.

5.1. Modeling Concurrent Object Systems in Maude

To begin with, we explain below Maude's syntax support for concurrent objects, and illustrate how a concurrent object system design can be expressed in Maude using such a syntax. As a running example we consider the goal of designing a communication protocol that can achieve in-order, fault-tolerant communication in an asynchronous medium where messages can arrive out-of-order and can furthermore be lost. The first order of business is to specify the *distributed states* of such a system that we will call *configurations*. After this is done, we can then specify its *concurrent behavior* by means of rewrite rules that define the *local actions* that each object in such a system can perform to achieve in-order fault-tolerant communication.

A system's distributed state or configuration can be naturally understood as a "soup" or "ether" medium in which both objects and messages "float." In such a fluid medium, objects and messages can come together and participate in concurrent *actions*. We can model such a fluid medium mathematically by means of *structural axioms* of associativity and commutativity. That is, we can think of a configuration as a *multiset* of objects and messages. Since each object should have a unique identifier, the objects in the system should form a *set*. However, there can be several copies of a message floating around in the system. All these ideas are succinctly captured by the `CONFIGURATION` module in Maude's standard prelude, shown in Figure 15, which can be used as a basis for defining many different concurrent object-based systems.

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
    [ctor assoc comm id: none] .

  sorts Oid Cid Object Msg Portal Configuration .
  subsort Object Msg Portal < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
    [ctor config assoc comm id: none] .
  op <> : -> Portal [ctor] .
endm
```

Figure 15: `CONFIGURATION` module

The essential facts about concurrent object configurations are all stated in the `CONFIGURATION` module. They are multisets of objects and messages belonging, respectively, to the subsorts `Object` and `Msg`. These multisets are built with the "empty syntax" (juxtaposition) associative-commutative union operator `__`, having `none` (empty configuration) as its identity element. Objects themselves are record-like structures having a *name* or object identifier of sort `Oid`, belonging to an *object class* whose name has sort `Cid` of class identifiers, and having an associative-commutative *set* of attribute-value pairs of sort `AttributeSet` built with the associative-commutative set union operator `_,_` with empty set `none` as its identity. Each such attribute-value pair has sort `Attribute` and can have any syntax we like. Likewise, we can use any syntax we like to define different kinds of messages. However, a message operator should be of sort `Msg` or a subsort of it, should have the attribute `msg`, and the *first argument* of any message operator should be the `Oid` of the message's *addressee*.

All these ideas can be illustrated by defining configurations of objects and messages for our fault-tolerant communication protocol in the functional module `FT-COMM-CONF` in Figure 16. Objects belong to one of the classes `Sender` or `Receiver`. In addition to importing the `CONFIGURATION` module, the `FT-COMM-CONF` module imports `QID-LIST` from Maude's prelude. This functional module provides a sort `QidList` formed from elements of sort `Qid` using the associative concatenation (empty syntax) operator `__`, having `nil` (empty list) as its identity element.

Here is a typical configuration for our desired fault-tolerant in-order communication protocol:

```
  < 'Alice : Sender | buff: 'a 'b 'c 'd, rec: 'Bob, cnt: 0 >
  < 'Bob : Receiver | buff: 'a, snd: 'Alice, cnt: 1 >
  (to 'Alice from 'Bob ack 0)
```

```
mod FT-COMM-CONF is
  extending CONFIGURATION .
  protecting NAT + QID-LIST .

  ops Sender Receiver : -> Cid [ctor] .
  subsort Qid < Oid .

  op cnt:_ : Nat -> Attribute [ctor gather (&)] .
  op buff:_ : QidList -> Attribute [ctor gather (&)] .
  op snd:_ : Oid -> Attribute [ctor gather (&)] .
  op rec:_ : Oid -> Attribute [ctor gather (&)] .

  op to_from_val_cnt_ : Oid Oid Qid Nat -> Msg [ctor msg] .
  op to_from_ack_ : Oid Oid Nat -> Msg [ctor msg] .
endm
```

Figure 16: `FT-COMM-CONF` module

In this configuration, there is a sender object `'Alice` and a receiver object `'Bob` of respective classes `Sender` and `Receiver`. Both senders and receivers have a buffer attribute `buff:` whose value is a list of quoted identifiers, either remaining to be sent by the sender, or already received by the receiver. Both also have a counter attribute `cnt:` which is used to ensure in-order communication. Initially, the value of the `cnt:` attribute is 0 for both senders and receivers, which gets increased along the communication as sends are received and acknowledged. Furthermore, to establish the target of the communication, sender objects have a receiver attribute `rec:` with the name of the object to which values in the list should be sent. Likewise, receiver objects have a sender attribute `snd:` with the name of the sender from which data is expected. Sender objects like `'Alice` send values in messages such as `to 'Bob from 'Alice val 'a cnt 0`. This message means that it is the first value in the list being transmitted and its contents is `'a`. In the above configuration this first message was already received by `'Bob` who now stores it in its buffer and is awaiting the second value, whose counter will be 1. However, due to the asynchronous nature of the communication, sender `'Alice` is not yet aware that the first value has already been received and is still holding it in its send buffer in case it was lost and has to be re-sent. In the meantime, receiver `'Bob` did send a message `to 'Alice from 'Bob ack 0` acknowledging receipt of the first value `'a`. But this acknowledgement has not yet been received by sender `'Alice`.

Of course, the functional module `FT-COMM-CONF` *does not do anything*. It describes, if you will, the *statics*, i.e., just the distributed states of our system. *Actions* themselves, the system's *dynamics*, are defined in the system module `FT-COMM` in Figure 17. The rules in `FT-COMM` are almost self-explanatory. Sender objects send the first value in their current list, plus a counter, to the receiver with the `snd` rule. However, they still keep the sent value in their buffer until an acknowledgement is received. If the expected acknowledgement is received, the sent value can be cleared from the send buffer and the counter is increased for the next value to be sent (rule `rec-ack1`). Note that the case where the sender receives an acknowledgement for counter value `M` with an empty buffer will not happen for starting configurations with only objects. The case of a duplicated acknowledgement message that *was* already received before is handled by rule `rec-ack2`, where the acknowledgement message is just discarded. Receiver objects perform two actions. Rule `rec1` describes the case where the "expected" value arrives, is put into the receive buffer, the counter is increased, and an acknowledgement message is sent to the sender. The case where the sent value *was* already received is handled by the rule `rec2`, where the receiver's local state does not change, but an acknowledgement message is nevertheless sent, since a previous acknowledgement may have been lost.

```
mod FT-COMM is
  including FT-COMM-CONF .

  var Q : Qid .          var L : QidList .
  vars N M : Nat .       vars A B : Oid .

  rl [snd] : < A : Sender | buff: Q L, rec: B, cnt: M >
    => < A : Sender | buff: Q L, rec: B, cnt: M >
       (to B from A val Q cnt M)
    [print "[snd]:␣" A "␣sends␣" Q "␣to␣" B] .
  rl [rec1] :
    (to B from A val Q cnt M)
    < B : Receiver | buff: L, snd: A, cnt: M >
    => < B : Receiver | buff: L Q, snd: A, cnt: s M >
       (to A from B ack M)
    [print "[rec1]:␣" B "␣receives␣new␣" Q "␣from␣" A] .
  crl [rec2] :
    (to B from A val Q cnt N)
    < B : Receiver | buff: L, snd: A, cnt: M >
    => < B : Receiver | buff: L, snd: A, cnt: M >
       (to A from B ack N)
    if N < M
    [print "[rec2]:␣" B "␣receives␣old␣" Q "␣from␣" A] .
  rl [rec-ack1] :
    (to A from B ack M)
    < A : Sender | buff: Q L, rec: B, cnt: M >
    => < A : Sender | buff: L, rec: B, cnt: s M >
    [print "[rec-ack1]:␣" A "␣receives␣1st␣ack␣" M "␣from␣" B] .
  crl [rec-ack2] :
    (to A from B ack N)
    < A : Sender | buff: L, rec: B, cnt: M >
    => < A : Sender | buff: L, rec: B, cnt: M >
    if N < M
    [print "[rec-ack2]:␣" A "␣receives␣old␣ack␣" N "␣from␣" B] .
endm
```

Figure 17: `FT-COMM` module

To show the progress of the computation we have added `print` attributes to each rule. In general, the print attribute allows one to specify information to be printed when a statement (equation, membership axiom, rule, or strategy) is executed, providing a minimized and flexible trace capability. If printing is turned on, when a statement with a print attribute is applied the pattern following `print` is instantiated using the corresponding matching substitution.

The `FT-COMM` module does indeed ensure in-order fault-tolerant communication. Furthermore, if the sender was sending a list of length k, counters in the sender and the receiver were originally 0, and the receiver's buffer was originally empty, there is a terminating rewrite sequence in whose final state both the sender and the receiver counters have the same value k, the sender's buffer is empty, and the original list is now in the receiver's buffer.

We use Maude's `frewrite` command to explore the behavior of `FT-COMM`. As discussed in Section 3.1 the `frewrite` command implements a rule and position fair rewriting strategy. In the special case of object-message configurations, such as the `FT-COMM` configurations, `frewrite` implements an object-message fair strategy. Roughly speaking, in each round, the strategy attempts to apply object-message rules to all existing object-message pairs and then attempts a single non-object-message rewrite of the resulting configuration using the remaining rules.

For object-message fair rewriting, the configuration constructor must have the `configuration` attribute, object constructors must have the `object` attribute, the first argument must be the object identifier, and message constructors must have the `message` (or `msg`) attribute with first argument an object identifier (the intended receiver). All of these requirements are met by the `FT-COMM` module—and also by the `FT-COMM-IN-FAULTY-ENV` below. To be an object-message rule, the lefthand side must have a configuration constructor on top with two arguments: A, with a message constructor on top, and B, with an object constructor on top, such that the first arguments of A and B are identical, i.e. A is a message for B. The rules `rec1`, `rec2`, `rec-ack1`, and `rec-ack2` of `FT-COMM` are object-message rules. However the rule `snd` is not—and the rules added by `FT-COMM-IN-FAULTY-ENV` below are not either. The full specification of `frewrite` and more examples can be found in [28].

We see in the output below that the protocol terminates as expected.

```
Maude> frew
  < 'Alice : Sender | cnt: 0, buff: 'a 'b 'c 'd, rec: 'Bob >
  < 'Bob : Receiver | cnt: 0, buff: nil, snd: 'Alice > .
result Configuration:
  < 'Alice : Sender | cnt: 4, buff: nil, rec: 'Bob >
  < 'Bob : Receiver | cnt: 4, buff: 'a 'b 'c 'd, snd: 'Alice >
```

To see how the protocol progresses, let us rewrite one step at a time using Maude's `continue` (abbreviated `cont`) command. We see that first `'Alice` sends `'a` to `'Bob` with count `0`.

```
Maude> frew [1] < 'Alice : Sender | cnt: 0, buff: 'a 'b 'c 'd, rec: 'Bob >
                < 'Bob : Receiver | cnt: 0, buff: nil, snd: 'Alice > .
result (sort not calculated): (
  < 'Alice : Sender | cnt: 0, buff: 'a 'b 'c 'd, rec: 'Bob >
  to 'Bob from 'Alice val 'a cnt 0)
  < 'Bob : Receiver | cnt: 0, buff: nil, snd: 'Alice >
```

In the next step, `'Bob` receives `'a` and sends an acknowledgement message to `'Alice` with count `0`.

```
Maude> cont 1 .
result Configuration:
  < 'Alice : Sender | cnt: 0, buff: 'a 'b 'c 'd, rec: 'Bob >
  < 'Bob : Receiver | cnt: 1, buff: 'a, snd: 'Alice >
  to 'Alice from 'Bob ack 0
```

In the third step `'Alice` sends `'a` to `'Bob` with count `0` again, since it has not yet received an `ack` message.

```
Maude> cont 1 .
result (sort not calculated): (
  < 'Alice : Sender | cnt: 0, buff: 'a 'b 'c 'd, rec: 'Bob >
  to 'Bob from 'Alice val 'a cnt 0)
  < 'Bob : Receiver | cnt: 1, buff: 'a, snd: 'Alice >
  to 'Alice from 'Bob ack 0
```

In the fourth step, `'Alice` receives the count `0` acknowledgement, increments its counter, and removes `'a` from its list. *Also*, `'Bob` receives the repeated `'a` with count `0` and sends another `ack`. This is two rewrites, although the command was to continue 1 step. This is because the `frewrite` strategy attempts to deliver a message to each object in a given round.

```
Maude> cont 1 .
result Configuration:
  < 'Alice : Sender | cnt: 1, buff: 'b 'c 'd, rec: 'Bob >
  < 'Bob : Receiver | cnt: 1, buff: 'a, snd: 'Alice >
  to 'Alice from 'Bob ack 0
```

Continue again, `'Alice` sends `'b` to `'Bob` with count `1`.

```
Maude> cont 1 .
result (sort not calculated): (
  < 'Alice : Sender | cnt: 1, buff: 'b 'c 'd, rec: 'Bob >
  to 'Bob from 'Alice val 'b cnt 1
  < 'Bob : Receiver | cnt: 1, buff: 'a, snd: 'Alice >
  to 'Alice from 'Bob ack 0
```

To see the difference between the strategies of the `rewrite` and `frewrite` commands, we use a configuration with two instances of the protocol, that is, two sender-receiver pairs. Using `frewrite` to execute the parallel protocol sessions, with the `print` attribute activated we see that activity of the two sessions is interleaved:

```
Maude> set print attribute on .
Maude> frew [24]
  < 'Alice : Sender | cnt: 0, buff: ('a 'b 'c 'd), rec: 'Bob >
  < 'Ada : Sender | cnt: 0, buff: ('a 'b 'c 'd), rec: 'Boris >
  < 'Bob : Receiver | cnt: 0, buff: nil, snd: 'Alice >
  < 'Boris : Receiver | cnt: 0, buff: nil, snd: 'Ada >  .
[snd]: 'Alice sends 'a to 'Bob
[snd]: 'Ada sends 'a to 'Boris
[rec1]: 'Bob receives new 'a from 'Alice
[rec1]: 'Boris receives new 'a from 'Ada
[snd]: 'Alice sends 'a to 'Bob
[snd]: 'Ada sends 'a to 'Boris
[rec-ack1]: 'Alice receives 1st ack 0 from 'Bob
[rec-ack1]: 'Ada receives 1st ack 0 from 'Boris
[rec2]: 'Bob receives old 'a from 'Alice
[rec2]: 'Boris receives old 'a from 'Ada
    ...
[snd]: 'Alice sends 'd to 'Bob
[snd]: 'Ada sends 'd to 'Boris
[rec-ack2]: 'Alice receives old ack 2 from 'Bob
[rec-ack2]: 'Ada receives old ack 2 from 'Boris
[rec1]: 'Bob receives new 'd from 'Alice
[rec1]: 'Boris receives new 'd from 'Ada
[snd]: 'Alice sends 'd to 'Bob
[snd]: 'Ada sends 'd to 'Boris
[rec-ack1]: 'Alice receives 1st ack 3 from 'Bob
[rec-ack1]: 'Ada receives 1st ack 3 from 'Boris
[rec2]: 'Bob receives old 'd from 'Alice
[rec2]: 'Boris receives old 'd from 'Ada
rewrites: 69 in 2ms cpu (3ms real) (23334 rewrites/second)
```

```
result Configuration:
  < 'Alice : Sender | cnt: 4, buff: nil, rec: 'Bob >
  < 'Ada : Sender | cnt: 4, buff: nil, rec: 'Boris >
  < 'Bob : Receiver | cnt: 4, buff: ('a 'b 'c 'd), snd: 'Alice >
  < 'Boris : Receiver | cnt: 4, buff: ('a 'b 'c 'd), snd: 'Ada >
  (to 'Alice from 'Bob ack 3)
  (to 'Ada from 'Boris ack 3)
```

If instead we use `rewrite`, the rules are applied first to objects and messages in one session, and when that terminates, the rules are applied to objects and messages of the other session.

```
Maude> rew [24]
  < 'Alice : Sender | cnt: 0, buff: ('a 'b 'c 'd), rec: 'Bob >
  < 'Ada : Sender | cnt: 0, buff: ('a 'b 'c 'd), rec: 'Boris >
  < 'Bob : Receiver | cnt: 0, buff: nil, snd: 'Alice >
  < 'Boris : Receiver | cnt: 0, buff: nil, snd: 'Ada > .
[snd]: 'Alice sends 'a to 'Bob
[rec1]: 'Bob receives new 'a from 'Alice
[rec-ack1]: 'Alice receives 1st ack 0 from 'Bob
[snd]: 'Alice sends 'b to 'Bob
    ...
[snd]: 'Alice sends 'd to 'Bob
[rec1]: 'Bob receives new 'd from 'Alice
[rec-ack1]: 'Alice receives 1st ack 3 from 'Bob
[snd]: 'Ada sends 'a to 'Boris
[rec1]: 'Boris receives new 'a from 'Ada
[rec-ack1]: 'Ada receives 1st ack 0 from 'Boris
    ...
[snd]: 'Ada sends 'd to 'Boris
[rec1]: 'Boris receives new 'd from 'Ada
[rec-ack1]: 'Ada receives 1st ack 3 from 'Boris
rewrites: 25 in 1ms cpu (1ms real) (18037 rewrites/second)
result Configuration:
  < 'Alice : Sender | cnt: 4, buff: nil, rec: 'Bob >
  < 'Ada : Sender | cnt: 4, buff: nil, rec: 'Boris >
  < 'Bob : Receiver | cnt: 4, buff: ('a 'b 'c 'd), snd: 'Alice >
  < 'Boris : Receiver | cnt: 4, buff: ('a 'b 'c 'd), snd: 'Ada >
```

In the case of finite behaviors, in the end the result is the same, but if the protocol execution did not terminate, then using `rewrite` one of the sessions might never execute, while using `frewrite` both sessions make progress.

Note that `FT-COMM` is a protocol that not only ensures in-order communication, but is also *fault-tolerant*. But can we also *model* a *faulty* environment where messages can be lost? Yes, we can do so in the system module `FAULTY-ENV`, in Figure 18, which adds such a faulty environment to `FT-COMM`. What is remarkable about the communication protocol specified in `FT-COMM` is that it *still works* in this faulty environment under suitable *fairness assumptions*. Of course, if as soon as every message is sent it is immediately destroyed by the `loss1` or the `loss2` rules, no communication will ever happen. But this is clearly an unfair behavior which makes the protocol's rules hopeless. By assuming fair executions and defining an *equational abstraction* [106] that collapses a multiset of messages into a set, the correctness of the `FT-COMM` protocol in such a faulty environment can actually be model checked using Maude's LTLR model checker [10]. For more on Maude's LTLR model checker see Section 9.

If we repeat the above `frew` command in the `FT-COMM-IN-FAULTY-ENV` module we see that the first few steps are the same as when done in the module `FT-COMM`. However, instead of `'Alice` receiving the `ack` from `'Bob` in the fourth

```
mod FT-COMM-IN-FAULTY-ENV is
  including FT-COMM .

  var Q : Qid .   var M : Nat .   vars A B : Oid .

  rl [loss1] : (to B from A val Q cnt M) => none
    [print "[loss1]:␣lost␣val␣" Q "␣to␣" B] .
  rl [loss2] : (to A from B ack M) => none
    [print "[loss2]:␣lost␣ack␣" M "␣to␣" A] .
endm
```

Figure 18: `FT-COMM-IN-FAULTY-ENV` module

step, the message is lost.

```
Maude> frew [1] < 'Alice : Sender | cnt: 0, buff: 'a 'b 'c 'd, rec: 'Bob >
                < 'Bob : Receiver | cnt: 0, buff: nil, snd: 'Alice > .
[snd]: 'Alice sends 'a to 'Bob
[rec1]: 'Bob receives new 'a from 'Alice
[snd]: 'Alice sends 'a to 'Bob
[loss2]: lost ack 0 to 'Alice
[rec2]: 'Bob receives old 'a from 'Alice
...
```

But is this all? Has the `FT-COMM` example illustrated, all there is to say about distributed objects in Maude? Not at all. It has illustrated the *most basic* possibilities, but many more remain. Here are some: (1) The Full Maude extension supports an even more expressive syntax for objects in which object classes can be structured in *multiple inheritance* hierarchies, rewrite rules can be specified more succinctly and are automatically inherited by subclasses [92, 28]. Furthermore, such class inheritance solves the well-known *inheritance anomaly* between subclassing and concurrency [93]. (2) Configurations need not be *flat*: they can have a *nested* structure —what we call a *Russian dolls* structure. Furthermore, such a nested structure can provide very useful mechanisms for *meta-object-based reflection* [110]. (3) Many distributed algorithms use time, and sometimes physical space, in an essential way. Both time and space can be modeled in an object-based manner in Maude using the Real-Time Maude tool [113] (see, e.g., [114, 79] for applications to sensor networks and to mobile ad-hoc networks). (4) Not only time, but also *randomness* in distributed object systems can be modeled by *probabilisitic rewrite rules* [1] (see, e.g., [76, 14] for two applications, respectively to sensor protocols and to cloud storage systems).

5.2. External Objects

Maude objects should be able to interact by message-passing with a variety of *external objects* that represent external entities with state, including the user regarded as another external object. Any configuration of Maude objects that wish to exchange messages with external objects must include a special portal constructor, defined in module `CONFIGURATION`:

```
  sort Portal .
  subsort Portal < Configuration .
  op <> : -> Portal [ctor] .
```

From an implementation point of view, the main purpose of having a portal subterm in a configuration is to avoid the degenerate case of a configuration that consists just of an object waiting for a message from outside of the configuration. This would be problematic because the special behavior for object-message rewriting and exchanging messages with external objects is attached to the configuration constructor:[7]

```
op __ : Configuration Configuration -> Configuration
  [ctor config assoc comm id: none] .
```

Exchanging messages with external objects is enabled by the `erewrite` command, which performs fair rewriting but also checks for messages in a configuration that are addressed to external objects and checks for messages from external objects that are queued, waiting to enter a configuration containing a specific object.

Certain predefined external objects are available and some of them are object managers that can create more ephemeral external objects that represent entities such as files and sockets or, as we will see in Section 8.3, virtual copies of the Maude interpreter itself.

5.2.1. Standard Streams

Each Unix process has three I/O channels, called standard streams: standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`). In Maude, these are represented as three unique external objects, that are defined in a predefined module `STD-STREAM`.

After more than 20 years you can now write a "Hello World!" program in Maude. Module `HELLO` in Figure 19 shows a very simple program implementing an interaction with the user, which is asked to introduce his/her name to be properly greeted. The equation for `run` produces a starting configuration, containing the portal, a user object to receive messages, and a message to `stdin` to read a line of text from the keyboard. When `stdin` has a line of text, it sends the text to the requesting object in a `gotLine` message.

```
Maude> erew run .
What is your name? Joe
Hello Joe
result Configuration: <> wrote(myObj, stdout) < myObj : myClass | none >
```

5.2.2. File I/O

Unlike standard streams, of which there are exactly three, a Unix process may have many different files open at any one time. Thus in order to create new file handle objects as needed, we have a unique external object called `fileManager`. To open a file, the `fileManager` is sent a message `openFile`. On success, an `openedFile` message is returned, with the name of an external

[7] While a single object or message has sort `Configuration` there is no configuration constructor for such a degenerate configuration. Requiring a portal term ensures that there is a configuration constructor for configurations which otherwise have only a single object or message.

```
mod HELLO is
  including STD-STREAM .

  op myClass : -> Cid .
  op myObj : -> Oid .
  op run : -> Configuration .

  var O : Oid .
  var A : AttributeSet .
  var S : String .
  var C : Char .

  eq run
    = <>
      < myObj : myClass | none >
      getLine(stdin, myObj, "What␣is␣your␣name?␣") .
  rl < myObj : myClass | A >
     gotLine(myObj, O, S)
  => < myObj : myClass | A >
     if S =/= ""
     then write(stdout, myObj, "Hello␣" + S)
     else none
     fi .
endm
```

Figure 19: `HELLO` module

object that is a handle on the open file as one of its arguments and to which messages to read and write the file can be directed. On failure, a `fileError` message is returned, with a text explanation of why the file could not be opened as one of its arguments. These messages are defined in the module `FILE`, which is distributed as part of the Maude system.

The `COPY-FILE` module in Figure 20 illustrates the basic use of files. It specifies a simple algorithm to copy files. In this case, the `run` operator takes two arguments: the names of the file to be copied and the name of the new file. As for the previous example, the equation for `run` produces a starting configuration, containing the portal, a user object to receive messages, and an initial message to open the original file. Once it is opened, the new file is created. Notice the `"w"` argument of the `openFile` message. Once both files are opened, a loop in which a line is read from the original file and written in the copy file is initiated. This loop ends when the end of the file is reached. Both files are then closed.

5.2.3. Socket I/O

Maude's support for sockets works in a similar way to that for files. There is a unique object, `socketManager` defined in a module `SOCKET` and messages to this object can be used to create client or server TCP internet sockets. This feature is crucial to *deploy* a Maude concurrent object system as a *distributed system*. Consider, for example, a configuration containing 1,000 Maude objects. They of course can be run on a single Maude interpreter, but then rewrites corresponding to message sends and receives are necessarily sequentialized by the interpreter. That is, concurrency is only *simulated* that way as an interleaving of

```
fmod MAYBE{X :: TRIV} is
  sort Maybe{X} .
  subsort X$Elt < Maybe{X} .
  op null : -> Maybe{X} .
endfm

view Oid from TRIV to CONFIGURATION is
  sort Elt to Oid .
endv

mod COPY-FILE is
  inc FILE .
  pr MAYBE{Oid} .

  op myClass : -> Cid .
  op myObj : -> Oid .
  ops in:_ out:_ : Maybe{Oid} -> Attribute .
  ops inFile:_ outFile:_ : String -> Attribute .

  op run : String String -> Configuration .
  vars Text Original Copy : String .
  vars FHIn FHOut : Oid .
  var  Attrs : AttributeSet .

  eq run(Original, Copy)
   = <>
     < myObj : myClass | in: null,  inFile: Original,
                         out: null, outFile: Copy >
     openFile(fileManager, myObj, Original, "r") .
  rl < myObj : myClass | in: null, outFile: Copy, Attrs >
     openedFile(myObj, fileManager, FHIn)
  => < myObj : myClass | in: FHIn, outFile: Copy, Attrs >
     openFile(fileManager, myObj, Copy, "w") .
  rl < myObj : myClass | in: FHIn, out: null, Attrs >
     openedFile(myObj, fileManager, FHOut)
  => < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     getLine(FHIn, myObj) .
  rl < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     gotLine(myObj, FHIn, Text)
  => < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     if Text == ""
     then closeFile(FHIn, myObj)
          closeFile(FHOut, myObj)
     else write(FHOut, myObj, Text)
     fi .
  rl < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     wrote(myObj, FHOut)
  => < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     getLine(FHIn, myObj) .
  rl < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     closedFile(myObj, FHIn)
     closedFile(myObj, FHOut)
  => none .
endm
```

Figure 20: File copy with external objects

rewrite steps. Using sockets we can easily distribute those 1,000 Maude objects into, say, 10 machines, each running its own Maude interpreter and holding a configuration of, say, 100 objects. The number of objects is immaterial and is just given for concreteness' sake; furthermore, new objects can be created and destroyed, and Maude objects may communicate not just with other Maude objects but also with various external object. The three key conceptual points to keep in mid are: (1) now the configuration or "soup" of 1,000 objects and messages has been distributed into 10 such soups distributed over 10 machines and communicating through sockets; (2) message passing communication between Maude objects belonging to one of those 10 sub-configurations will happen as usual by rewriting performed by the Maude interpreter for that configuration; (3) instead, a message generated by rewriting in sub-configuration, say, number 2 but addressed to another object in sub-configuration number 8 will be: (i) transformed into a string, (ii) sent through a socket linking those two configurations, (iii) transformed back into a message in sub-configuration 8, and (iv) delivered to the addressee object there, that will then consume it by an appropriate rewrite rule.

For additional details on socket external objects in Maude see [24, 28].

Further Reading. The two most complete references for the semantics of object-based systems in Maude are probably [92, 28]. How meta-objects that can control other objects (or entire object sub-configurations) in "Russian dolls" distributed architectures can easily be defined in Maude is explained in [110]. The specification of real-time concurrent object systems in the Real-Time Maude extension is discussed in [113]. An interesting application using sockets to specify and deploy a mobile version of Maude called Mobile Maude is described in [42, 49].

6. B-Unification, Variants, and $E \cup B$-unification

Maude's predecessors envisioned the inclusion of several symbolic features which were never included in Maude until quite recently: (i) Eqlog [65] envisioned an integration of order-sorted equational logic with Horn logic, providing logical variables, constraint solving, and automated reasoning capabilities on top of order-sorted equational logic (see Section 8.2 for an actual Eqlog interpreter); and (ii) MaudeLog [91] envisioned an integration of order-sorted rewriting logic with queries including logical variables. Among the many symbolic reasoning features that can be supported by Maude, in this paper we focus on order-sorted equational unification (this Section) and order-sorted narrowing-based symbolic reachability analysis (Section 7). For a broader discussion of other symbolic reasoning methods and tools in rewriting logic and Maude see Section 9 and [101**?**].

At first sight, adding symbolic reasoning capabilities to Maude might seem like an incremental improvement; but this is not at all the case. Let us focus, for the moment, on what it means to add equational unification. Order-sorted unification modulo axioms first became available as a built-in feature in 2009 as part of the Maude 2.4 release [25], which supported any combination of order-sorted symbols declared to be either free or associative-commutative (AC). Unification was updated in 2011 as part of the Maude 2.6 release [41]. Built-in equational unification was extended to allow any combination of symbols being either free, commutative (C), associative-commutative (AC), or associative-commutative with an identity symbol (ACU). The performance was dramatically improved, allowing further development of other techniques in Maude. As we explain below, built-in order-sorted unification has been further extended later to allow associativity-only (A) and well as identity (U) axioms. This all means that for axioms B including combinations of these axioms, equational unification in an order-sorted theory (Σ, B) is supported by Maude.

The next natural but highly non-trivial step is supporting equational unification in order-sorted theories of the form $(\Sigma, E \cup B)$, where the equations E oriented as rules are convergent modulo such axioms B. This is highly non-trivial because, although it is well-known that narrowing with the equations E modulo axioms B provides a complete unification *semi-algorithm* [74], the prospects of obtaining a practical equational unification algorithm in this general setting looked rather dim for the following reasons: (i) without an efficient E, B-narrowing strategy the compounded combinatorial explosion of B-unification and unrestricted E, B-narrowing would make such a semi-algorithm hopeless; (ii) almost nothing was known about E, B-narrowing strategies for $B \neq \emptyset$; and (iii) almost nothing was known about *termination* results for complete E, B-narrowing strategies for $B \neq \emptyset$ that would make the (in general undecidable) $E \cup B$-unification semi-algorithm into a *decidable unification algorithm.* The key concepts making it possible to break through these daunting obstacles have been those of *variant* [34], and of *folding variant narrowing* and *variant unification* [60]. The introduction of these variant-based concepts in Maude (see Sections 6.2–6.3 below) has led to a drastic improvement in Maude's symbolic reasoning capabilities: variant generation, variant-based $E \cup B$-unification, and symbolic reachability based on variant-based $E \cup B$-unification became all available for the first time. Initially, all the variant-based features were only available in Full Maude, and for a restricted class of theories called *strongly right irreducible.* However, all these variant-based features are now efficiently supported in Core Maude as explained in Sections 6.2–6.3.

Order-sorted unification modulo axioms B was extended again in 2016 as part of the Maude 2.7 release [39]. First, the built-in unification algorithm allows any combination of symbols being free, C, AC, ACU, CU (commutativity and identity), U (identity), Ul (left identity), and Ur (right identity). Second, variant generation and variant-based unification were implemented as built-in features in Maude. This built-in implementation works for any convergent theory modulo the axioms described above, both allowing very general equational theories (beyond the strongly right irreducible ones) and boosting the performance not only of these features but of their applications. B-unification has been recently further extended to the *associative* case as part of the Maude 2.7.1 release [40]. This is a key contribution because associative unification is infinitary in general and the development of an efficient and *effective in practice* associative unification algorithm that furthermore supports order-sorted typing and combination with any other symbols either free or themselves combining some A and/or C and/or U axioms, has been a highly non-trivial challenge. A key concern in meeting this challenge has been the identification of a fairly broad class of unification problems appearing in many practical applications for which our algorithm is guaranteed to terminate with a *finite* and *complete* set of unifiers. To deal with the unavoidable possibility that the given unification problem may have an infinite set of unifiers, when the problem is outside the class supported by the algorithm in a complete way, the algorithm returns a finite set of unifiers with an *explicit warning* that such a set may be incomplete. In a good number of applications where we have used these new associative symbolic features of Maude, unification problems falling outside the class supported by our algorithm in a complete way often do not even arise[8] in practice.

6.1. Order-Sorted Unification Modulo Axioms B

Maude currently provides an order-sorted B-unification algorithm for all order-sorted theories (Σ, B) such that the order-sorted signature Σ is *preregular* modulo B (see [47, Footnote 2]) and the axioms B associated to function symbols can be any combination of: (i) `iter` equational axioms, which can be declared for some unary symbols; (ii) `comm` (C) commutativity axioms; (iii) `assoc` (A) associativity axioms; and (iv) `id:` identity axioms (U) as well as the `left id:` left identity axioms (Ul) and the `right id:` right identity axioms (Ur), *except* for the following combinations not currently supported: `assoc id`, `assoc left id`, and `assoc right id`. However, these three remaining subcases are easily supported by turning the respective identity axioms into *oriented equations* and then using variant unification modulo the remaining axioms B (see Section 6.2). Maude 2.7.1 provides a B-unification command of the form

```
unify [n] in ⟨ModId⟩ : ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
```

where $k \geq 1$, n is an optional argument providing a bound on the number of unifiers requested, and `ModId` is the module where the command takes place. The unification infrastructure now supports the notion of incomplete unification algorithms (e.g. for associative unification).

Let us show some examples of unification with an associative attribute, which is the last feature available in Maude 2.7.1. See [24] for more examples of unification modulo axioms.

Consider a very simple module where the symbol `_._` is associative:

```
fmod UNIFICATION-A is
  protecting NAT .
  sort NList .
  subsort Nat < NList .
  op _._ : NList NList -> NList [assoc] .
  vars X Y Z P Q : NList .
endfm
```

[8] The Maude-NPA protocol analyzer has already been tested with various protocols using associative operators without encountering any incompleteness warnings (see [68]).

Even if associative unification is infinitary (we include concrete examples below) there are many realistic unification problems that are still finitary. The following unification problem returns five unifiers:

```
Maude> unify in UNIFICATION-A : X . Y . Z =? P . Q .

Solution 1
X:NList --> #1:NList . #2:NList
Y:NList --> #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList
Q:NList --> #2:NList . #3:NList . #4:NList

Solution 2
X:NList --> #1:NList
Y:NList --> #2:NList . #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList . #2:NList
Q:NList --> #3:NList . #4:NList

Solution 3
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList . #4:NList
P:NList --> #1:NList . #2:NList . #3:NList
Q:NList --> #4:NList

Solution 4
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList . #2:NList
Q:NList --> #3:NList

Solution 5
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList
Q:NList --> #2:NList . #3:NList
```

The above output illustrates how fresh variables, not occurring in the original unification problem, are introduced by Maude by using the notation `#N:Sort`.

One possible condition for finitary associative unification (see [40] for further details) is having linear (i.e., unrepeated) *list* variables, as in the example above. On the other hand, the unification problem may not be linear, but it may be easy to detect that there is no unifier, e.g. it is impossible to unify a list `X` concatenated with itself with another list `Y` concatenated also with itself but with a natural number, e.g. 1, in between.

```
Maude> unify in UNIFICATION-A : X . X =? Y . 1 . Y .
No unifier.
```

When nonlinear variables occur on both sides of an associative unification problem, Maude always ensures termination, but sometimes raises an incompleteness warning. Several cases are possible (see [40] for further details):

1. One or more cycles are detected, but they do not give rise to unifiers.

```
Maude> unify in UNIFICATION-A : 0 . Q =? Q . 1 .
No unifier.
```

2. There is at least one cycle that produces an infinite family of most general unifiers. In this case a warning will be issued and only the acyclic solutions are returned.

```
Maude> unify in UNIFICATION-A : 0 . X =? X . 0 .
Warning: Unification modulo the theory of operator _._
has encountered an instance for which it may not be complete.

Solution 1
X:NList --> 0
Warning: Some unifiers may have been missed due to incomplete
unification algorithm(s).
```

Note that the unification problem `0 . X` $=^?$ `X . 0` has an infinite family of most general unifiers $\{\texttt{X} \mapsto \texttt{0}^n\}$ for $\texttt{0}^n$ being a list of n consecutive `0` elements.

3. There is at least one nonlinear variable with more than two occurrences and Maude will use a depth bound rather than cycle detection. If the search tree grows beyond the depth bound, the offending branches will be pruned, and a warning will be given.

```
Maude> unify in UNIFICATION-A : X . X . X =? Y . Y . Z . Y .
Warning: Unification modulo the theory of operator _._
has encountered an instance for which it may not be complete.

Solution 1
X:NList --> #1:NList . #1:NList . #1:NList . #1:NList
Y:NList --> #1:NList . #1:NList . #1:NList
Z:NList --> #1:NList . #1:NList . #1:NList

Solution 2
X:NList --> #1:NList . #1:NList . #1:NList
Y:NList --> #1:NList . #1:NList
Z:NList --> #1:NList . #1:NList . #1:NList

Solution 3
X:NList --> #1:NList . #1:NList
Y:NList --> #1:NList
Z:NList --> #1:NList . #1:NList . #1:NList
Warning: Some unifiers may have been missed due to incomplete
    unification algorithm(s).
```

See [24] for details on the meta-level commands for unification, which are extended with a new constant `noUnifierIncomplete`, and additional warnings generated during associative unification.

6.2. Variants

Consider a term t in a convergent order-sorted equational theory $(\Sigma, E \cup B)$ where the equations E are assumed unconditional. Intuitively, a *variant* of t [34] is *the normal form* u of an *instance* $t\theta$ of t by a substitution θ, which is computed by simplification with E modulo B. For example, for the unsorted signature $\Sigma = \{0, s, +\}$ of addition in the Peano natural numbers, with $E = \{x + 0 = x, x + s(y) = s(x + y)\}$ and $B = \emptyset$, the terms x and $s(x + y')$ are variants of the term $x + y$ for the respective substitutions $\theta_1 = \{y \mapsto 0\}$ and $\theta_2 = \{y \mapsto s(y')\}$. Technically, it is useful to tighten the notion of variant in two ways [60]: (i) by viewing a variant of t as a pair (u, θ) instead of just a

```
fmod EXCLUSIVE-OR is
  sorts Nat NatSet .  subsort Nat < NatSet .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op mt : -> NatSet .
  op _*_ : NatSet NatSet -> NatSet [assoc comm] .
  vars X Y Z U V : [NatSet] .
  eq [idem] :     X * X = mt    [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
  eq [id] :       X * mt = X    [variant] .
endfm
```

Figure 21: `EXCLUSIVE-OR` module

normal form u of an instance term $t\theta$, and (ii) by requiring, without any real loss of generality, that the substitution θ is in normal form, i.e., that for each variable x, the term $\theta(x)$ is in normal form. Of course, *some variants are more general than others*. For example, among the variants of $x + y$, $(x, \{y \mapsto 0\})$ is more general than $(s(x'), \{x \mapsto s(x'), y \mapsto 0\})$, and $(s(x + y'), \{y \mapsto s(y')\})$ is more general than $(s(s(x')+y'), \{x \mapsto s(x'), y \mapsto s(y')\})$. The general definition for $(\Sigma, E \cup B)$ is that a variant (u, θ) of t is *more general* than another variant (v, η) of t iff there is a substitution γ such that: (i) $u\gamma =_B v$, and (ii) for each variable z in t, $\gamma(\theta(z)) =_B \eta(z)$.

A convergent order-sorted theory $(\Sigma, E \cup B)$ is said to have the *finite variant property* (FVP) [34, 60] iff each Σ-term t has a *finite* set of most general variants. We can illustrate this property both by its absence and by its presence. For example, $E = \{x + 0 = x, x + s(y) = s(x + y)\}$ is *not* FVP, since $(x + y, id)$, $(s(x + y_1), \{y \mapsto s(y_1)\})$, $(s(s(x + y_2)), \{y \mapsto s(s(y_2))\})$, ..., $(s^n(x + y_n), \{y \mapsto s^n(y_n)\})$, ..., are all *incomparable* variants of $x+y$. Instead, the following theory *is* FVP:

Example 10. *Consider the equational theory for exclusive or in module* `EXCLUSIVE-OR` *in Figure 21. The attribute* `variant` *specifies that these equations will be used for variant generation and variant-based unification. The* `owise` *attribute for equations should* never *be used in variant equations.*

Given the term `X * Y`*, we can construct several of its variants as follows:*

1. *The pair* $(s(0) * s(0), \{X \mapsto s(0), Y \mapsto s(0)\})$ *is normalized into* $(mt, \{X \mapsto s(0), Y \mapsto s(0)\})$ *since the term s(0) * s(0) is simplified into mt;*
2. *The pair* $(s(0) * U * s(0), \{X \mapsto s(0) * U, Y \mapsto s(0)\})$ *is normalized into* $(U, \{X \mapsto s(0) * U, Y \mapsto s(0)\})$*, and;*
3. *The pair* $(s(0) * U * s(0) * V, \{X \mapsto s(0) * U, Y \mapsto s(0) * V\})$ *is normalized into* $(U * V, \{X \mapsto s(0) * U, Y \mapsto s(0) * V\})$*.*

As claimed above, the module `EXCLUSIVE-OR` *is FVP. But how can we* know *this? The answer is very simple. Maude provides a variant generation command of the form:*

```
get variants [ n ] in ⟨ModId⟩ : ⟨Term⟩ .
```

where n is an optional argument providing a bound on the number of variants requested, so that if the cardinality of the set of variants is greater than the specified bound, the variants beyond that bound are omitted; and `ModId` *is the module where the command takes place.*

Now, as proved in [22], a convergent theory $(\Sigma, E \cup B)$ is FVP iff for each of its function symbols f, say, $f : s_1 \dots s_n \to s$, the term $f(x_1, \dots x_n)$ with x_i of sort s_i, $1 \leq i \leq n$, has a finite *number of variants. Therefore, we can check that the* `EXCLUSIVE-OR` *module of Example 10 has the finite variant property by simply generating the variants for the exclusive-or symbol.*

```
Maude> get variants in EXCLUSIVE-OR : X * Y .
Variant 1                                                Variant 7
[NatSet]: #1:[NatSet] * #2:[NatSet]    .........   [NatSet]: %1:[NatSet]
X --> #1:[NatSet]                                        X --> %1:[NatSet]
Y --> #2:[NatSet]                                        Y --> mt
```

Note that all other symbols f in this module, except the exclusive or symbol, are constructors and therefore have the single, trivial variant $(f(x_1, \dots x_n), id)$, where id denotes the identity substitution.

The above output illustrates a difference between unifiers returned by the built-in unification modulo axioms and substitutions (or unifiers) returned by variant generation or variant-based unification: there are two forms of fresh variables, the former `#n:Sort` *and the new* `%n:Sort`*. Note that the two forms have different counters.*

The FVP property is extremely useful. For example, we shall show in Section 6.3 that if $(\Sigma, E \cup B)$ is FVP and B has a finitary B-unification algorithm, then there is also a, variant-based, finitary $E \cup B$-unification algorithm. But how common is it for a convergent theory to be FVP? Certainly not so common, but more common than one might think. Roughly speaking, *recursive equations* such as, for example, the addition equation $x + s(y) = s(x + y)$ push a theory outside the FVP fold. This seems quite restrictive; but one can easily overlook the power of equational simplification *modulo* axioms such as associativity commutativity (AC). Specifying a function with equations modulo AC can quite often make, what would typically require a recursive function definition without using AC, into a non-recursive one. For example, we can extend the above example of Peano addition with a new sort *Bool* and a strict order predicate $>$ defined by equations: $\{0 > x = \mathit{false}, s(x) > 0 = \mathit{true}, s(x) > s(y) = x > y\}$, which are *unavoidably recursive* in *this* representation. However, $>$ and various other arithmetic functions are part of an FVP theory when we define them modulo ACU by representing the natural numbers with constants 0 and 1 and ACU binary constructor $+$, so that 3 is represented as $1 + 1 + 1$. Then we can define, in a *non-recursive* way, arithmetic functions such as: p for the predecessor function, *max* (resp. *min*) for the biggest (resp. smallest) of two numbers, $\backslash$ for the "monus" function, d for the symmetric difference function, `>` for the strict order predicate, and `~` for the equality predicate, yielding the FVP theory in Figure 22. We can *check* that it is FVP by computing the variants of its function symbols. For example, the generation of variants for the following terms all stop with a finite number of variants:

```
fmod NAT-FVP is
  protecting TRUTH-VALUE .
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero [ctor] .
  op 1 : -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0 prec 33] .
  op _+_ : NzNat NzNat -> NzNat [ctor assoc comm id: 0 prec 33] .
  op p : NzNat -> Nat .               *** predecessor
  op max : Nat Nat -> Nat [comm] .
  op max : NzNat NzNat -> NzNat [comm] .
  op min : Nat Nat -> Nat [comm] .
  op min : NzNat NzNat -> NzNat [comm] .
  op d : Nat Nat -> Nat [comm] .      *** symmetric difference
  op _\_ : Nat Nat -> Nat .           *** monus
  op _~_ : Nat Nat -> Bool [comm] .   *** equality predicate
  op _>_ : Nat Nat -> Bool .

  vars N M : Nat .
  vars N' M' K' : NzNat .

  eq p(N + 1) = N [variant] .
  eq max(N + M, N) = N + M [variant] .
  eq min(N + M, N) = N [variant] .
  eq d(N + M, N) = M [variant] .
  eq (N + M) \ N = M [variant] .
  eq N \ (N + M) = 0 [variant] .
  eq N ~ N = true [variant] .
  eq (N + M') ~ N = false [variant] .
  eq M + N + 1 > N = true [variant] .
  eq N > N + M = false [variant] .
endfm
```

Figure 22: `NAT-FVP` module

```
Maude> get variants in NAT-FVP : N \ M . --- 3 variants
Maude> get variants in NAT-FVP : N ~ M . --- 4 variants
Maude> get variants in NAT-FVP : N > M . --- 3 variants
```

As shown in [51], this FVP example (borrowed from [51]) can be easily extended to an even richer FVP example `INT-FVP` where all the above functions (except monus, which is superseded by actual integer difference using unary minus and +) are extended to the integers, and an absolute value function on integers is added.

Another interesting feature is that variant generation is *incremental*. In this way we are able to support general convergent equational theories $(\Sigma, E \cup B)$ that need be FVP, so that a term t *may* have an infinte number of variants. Let us consider the Maude specification `NAT-VARIANT`, given in Figure 23, of our previous functional module for natural number addition in Peano notation that we already know does *not* have the finite variant property.

On the one hand, it is possible to have a term with a finite number of most general variants although the theory is not FVP. For instance, the term `X + s(0)` has the single variant `s(X)`.

```
Maude> get variants in NAT-VARIANT : X + s(0) .
Variant 1
Nat: s(#1:Nat)
X --> #1:Nat
```

```
fmod NAT-VARIANT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  eq [base] : X + 0 = X [variant] .
  eq [ind] :  X + s(Y) = s(X + Y) [variant] .
endfm
```

Figure 23: `NAT-VARIANT` module

On the other hand, we can incrementally generate the variants of a term that we suspect does not have a finite number of most general variants. For instance, the term `s(0) + X` has an infinite number of most general variants. In such a case, Maude can either output all the variants to the screen (and the user can stop the process whenever she wants), or generate the first n variants by including a bound n in the command.

```
Maude> get variants [10] in NAT-VARIANT : s(0) + X .
Variant 1                                                Variant 10
Nat: s(0) + #1:Nat      ........................      Nat: s(s(s(s(s(0)))))
X --> #1:Nat                                             X --> s(s(s(s(0))))
```

A third approach, particularly when we are *not* sure whether a term has a finite number of variants, is to incrementally increase the bound and, if we obtain a number of variants smaller than the bound, then we know for sure that it had a finite number of most general variants.

6.3. Equational Narrowing, Folding Variant Narrowing, and $E \cup B$-unification

Variant generation relies on *folding variant narrowing* [60], which we informally describe in this section. Since folding variant narrowing is a narrowing *strategy* (more on this below), we begin by explaining *unrestricted equational narrowing* in a convergent order-sorted equational theory $(\Sigma, E \cup B)$. In ordinary rewriting modulo axioms B, one must choose which subterm $t|_p$ of the subject term t and which rule $l \to r$ are going to be considered for rewriting. Instead, in narrowing, one must choose which subterm $t|_p$ of the subject term t, which rule $l \to r$, and which variables of $t|_p$ may need some instantiation in order to match l. That is, a rewriting step computes a *B-matching substitution* σ such that $t|_p =_B \sigma(l)$, whereas a narrowing step computes a *B-unifier* σ such that $\sigma(t|_p) =_B \sigma(l)$.

Consider the functional module `NAT-VARIANT` of Figure 23, the term `s(0) + X` and the two equations `base` and `ind`. Narrowing will instantiate[9] variable `X` with `0` and `s(X')`, respectively. The following two narrowing steps are generated:

$$\mathtt{s(0)} + \mathtt{X} \leadsto_{\{\mathtt{X} \mapsto \mathtt{0}\},\mathtt{base}} \mathtt{s(0)}$$

$$\mathtt{s(0)} + \mathtt{X} \leadsto_{\{\mathtt{X} \mapsto \mathtt{s(\#1:Nat)}\},\mathtt{ind}} \mathtt{s(s(0) + \#1:Nat)}$$

[9]New variables in Maude are introduced as `#1:Nat` or `%1:Nat` instead of `X'`.

Note that, for simplicity, we show only the bindings of the unifier that affect the input term. There are infinitely many narrowing derivations starting at the input expression `s(0) + X` (at each step the reduced subterm is underlined):

1. $\underline{\mathtt{s(0)+X}} \leadsto_{\{\mathtt{X}\mapsto \mathtt{0}\},\mathtt{base}} \mathtt{s(0)}$
2. $\underline{\mathtt{s(0)+X}} \leadsto_{\{\mathtt{X}\mapsto \mathtt{s(\#1{:}Nat)}\},\mathtt{ind}} \mathtt{s(}\underline{\mathtt{s(0)+\#1{:}Nat}}\mathtt{)} \leadsto_{\{\mathtt{\#1{:}Nat}\mapsto \mathtt{0}\},\mathtt{base}} \mathtt{s(s(0))}$
3. $\underline{\mathtt{s(0)+X}} \leadsto_{\{\mathtt{X}\mapsto \mathtt{s(\#1{:}Nat)}\},\mathtt{ind}} \mathtt{s(}\underline{\mathtt{s(0)+\#1{:}Nat}}\mathtt{)}$
   $\leadsto_{\{\mathtt{\#1{:}Nat}\mapsto \mathtt{s(\#2{:}Nat)}\},\mathtt{ind}} \mathtt{s(s(}\underline{\mathtt{s(0)+\#2{:}Nat}}\mathtt{))} \leadsto_{\{\mathtt{\#2{:}Nat}\mapsto \mathtt{0}\},\mathtt{base}} \mathtt{s(s(s(0)))}$

And some of those, infinitely many, narrowing derivations are infinite in length, e.g. by applying rule `ind` infinitely many times:

$$\begin{array}{ll}\underline{\mathtt{s(0)+X}} & \leadsto_{\{\mathtt{X}\mapsto \mathtt{s(\#1{:}Nat)}\},\mathtt{ind}} \mathtt{s(}\underline{\mathtt{s(0)+\#1{:}Nat}}\mathtt{)}\\ & \leadsto_{\{\mathtt{\#1{:}Nat}\mapsto \mathtt{s(\#2{:}Nat)}\},\mathtt{ind}} \mathtt{s(s(}\underline{\mathtt{s(0)+\#2{:}Nat}}\mathtt{))}\\ & \leadsto_{\{\mathtt{\#2{:}Nat}\mapsto \mathtt{s(\#3{:}Nat)}\},\mathtt{ind}} \mathtt{s(s(s(}\underline{\mathtt{s(0)+\#3{:}Nat}}\mathtt{)))}\\ & \ldots\end{array}$$

A *narrowing path* $u_0 \leadsto_{\theta_1} u_1 \ldots u_{n-1} \leadsto_{\theta_n} u_n$, $n \geq 0$, is denoted $u_0 \leadsto^*_\theta u_n$, where $\theta = \theta_1 \ldots \theta_n$ is the so-called *accumulated substitution* obtained by composing the substitutions $\theta_1, \ldots, \theta_n$ for each step.

For a convergent order-sorted equational theory $(\Sigma, E \cup B)$ any $E \cup B$-unification problem can be reduced to a narrowing problem as follows:

1. we add a fresh new sort `Truth` to Σ with a constant `tt`;
2. for each top sort of each connected component of sorts we add a binary predicate `eq` of sort `Truth` and add to E the equation `eq(x,x) = tt`, where `x` has such a top sort;
3. we then reduce an $E \cup B$-unification problem $t =^? t'$ to the narrowing reachability problem
$$\mathtt{eq}(t, t') \leadsto^* \mathtt{tt}$$
modulo B in the theory extending $(\Sigma, E \cup B)$ with these new sorts, operators, and equations, where E and the new equations are used as rewrite rules.

That is, we search for all narrowing paths modulo B from $\mathtt{eq}(t, t')$ to `tt`. The accumulated substitution θ associated to each such path then gives us a $E \cup B$-*unifier* of the equation $t =^? t'$. However, as already pointed out, *unrestricted* equational narrowing with an equational theory $(\Sigma, E \cup B)$ can be very wasteful because of the compounded combinatorial explosion of potentially many B-unifiers at each step and the existence of many, often redundant, narrowing paths. Furthermore, only if all such narrowing paths terminate can we be sure to have a *finite*, complete set of $E \cup B$-unifiers for a given unification problem. But, as pointed out before, modulo axioms B such as AC, unrestricted narrowing almost never terminates, so we are in practice condemned to a $E \cup B$-unification *semi-algorithm*. The upshot is that unrestricted narrowing modulo B is actually hopeless in practice: without a suitable narrowing strategy drastically restricting the narrowing paths and able to detect when narrowing paths can be stopped there is no hope for a practical $E \cup B$-unification semi-algorithm, and even less hope for a, narrowing based, $E \cup B$-unification algorithm.

Folding Variant Narrowing. The *folding variant narrowing strategy* proposed in [60] solves all these problems in one blow. It furthermore provides a method to compute a complete set of variants for *any* convergent equational theory $(\Sigma, E \cup B)$ such that B has a B-unification algorithm. We briefly explain this strategy and how it is used by Maude to compute a complete set of variants of a term, and a complete set of $E \cup B$-unifiers for *any* convergent $(\Sigma, E \cup B)$ having a B-unification algorithm.

Roughly speaking, given a convergent theory $(\Sigma, E \cup B)$, the folding variant narrowing strategy defines a *subset* of narrowing paths, so that only those in such subset are computed. To begin with, only narrowing paths of the form $u_0 \leadsto^*_\theta u_n$ with u_n and θ *normalized* are considered. This exactly means that (u_n, θ) is a *variant* of u_0. In fact, it is shown in [60] that such sequences compute a *complete set of most general variants* of u_0. But folding variant narrowing goes further in two ways: (i) it furthermore discards *redundant* narrowing paths $u_0 \leadsto^*_\theta u_n$, where such a path is redundant if there is another path $u_0 \leadsto^*_{\theta'} u'_m$ such that the variant (u'_m, θ') is *more general* than the variant (u_n, θ). We can then "fold," i.e., subsume, the less general path into the more general one; and (ii) the folding variant narrowing strategy can *safely stop* when all new paths thus computed in a breadth first manner can be folded into previously computed paths. The following are the most remarkable properties about folding variant narrowing for a convergent equational theory $(\Sigma, E \cup B)$:

1. It computes a complete set of most general variants for any term t. In general this set may be infinite and is computed incrementally by Maude.
2. This complete set of variants is *finite* (and the strategy *terminates* for any input term t) iff $(\Sigma, E \cup B)$ is FVP.
3. Extending $(\Sigma, E \cup B)$ with equality operators and the constant `tt` as explained above, a *complete* set of most general $E \cup B$-unifiers of $t =^? t'$ is obtained as the set of all substitutions θ such that $(\mathtt{tt}, \theta)$ belongs to the complete set of most general variants of the term $\mathtt{eq}(t, t')$. In general this set is infinite and is computed by Maude incrementally, so we only have a *semi-algorithm*.
4. This variant-based $E \cup B$-unification semi-algorithm becomes a *finitary* $E \cup B$-unification algorithm, therefore terminating for any unification problem $t =^? t'$, iff $(\Sigma, E \cup B)$ is FVP.

Consider for instance a variant unification problem between terms $X * Y$ and $U * V$ in the `EXCLUSIVE-OR` theory above, which is FVP.

```
Maude> variant unify in EXCLUSIVE-OR : X * Y =? U * V  .
--- 57 unifiers are reported
```

Similarly, we can call variant unification between term $X + Y$ and 0, which has only possible solution, in the `NAT-VARIANT` module above, which is not FVP. The variant unify command terminates if we limit the number of solutions to 1.

```
Maude> variant unify [1] in NAT-VARIANT : X + Y =? 0  .

Unifier #1
rewrites: 4 in 0ms cpu (0ms real) (12903 rewrites/second)
X --> 0
Y --> 0
```

However, it does not terminate if we limit the number of solutions to 2, since keeps trying to generate more and more solutions without being to realize that there is only one.

Further Reading. For order-sorted unification modulo axioms B see [104, 71, 53]. For Maude's order-sorted associative unification algorithm and its combination with other axioms B see [40]. The original paper on variants is [34]. The correctness of the method for checking that a theory is FVP as well as several formulations of the variant notion can be found in [22]. Folding variant narrowing and variant unification are studied in [60]. Note that if an equational theory $(\Sigma, E \cup B)$ is FVP, with a B-unification algorithm, $E \cup B$-unifiability of a conjunction of equalities is decidable. Assuming non-empty sorts, satisfiability in the initial algebra $T_{\Sigma/E\cup B}$ of any positive (only negations and disjunctions) quantifier-free (QF) Σ-formula is then *decidable*. But one can go further. Under mild assumptions about a constructor subspecification for $(\Sigma, E \cup B)$ a theory-generic satisfiability algorithm for *all* QF Σ-formulas can be given (see [102, 69], and for detailed algorithms and an implementation in Maude [124]).

7. Narrowing with Rules and Narrowing Search

When formally analyzing the properties of a rewrite theory $(\Sigma, E \cup B, R, \phi)$, an important problem is ascertaining for specific patterns (i.e., terms with variables) t and t' the following *symbolic reachability problem:*

$$\exists X \; t \longrightarrow^* t'$$

with X the set of variables appearing in t and t', which for this discussion we may assume are a disjoint union of those in t and those in t'. That is, t and t' symbolically describe sets of concurrent states $[\![t]\!]$ and $[\![t']\!]$ (namely, all the ground substitution instances of t, resp. t', or, more precisely, the $E \cup B$-equivalence classes associated to such ground instances). And we are asking: is there a state in $[\![t]\!]$ from which we can *reach* a state in $[\![t']\!]$ after a finite number of rewriting steps? Solving this problem means *searching* for a symbolic solution to it in a hopefully *complete* way (so that if a solution exists it will be found). This has the flavor of the `search` command, and therefore of some kind of model checking: for example, t' could be a pattern describing the *violation* of an invariant, and t a pattern describing a set of initial states. The main difference (and the advantage in this case) is that with the `search` command we explore the *concrete* states reachable from some given *concrete* initial state. Instead, here we explore *symbolically* reachability between possibly infinite *sets* of states such as $[\![t]\!]$ and $[\![t']\!]$. The way we do so is by *narrowing* t with the rewrite rules R in the given system module $(\Sigma, E \cup B, R, \phi)$, *modulo* the equations $E \cup B$.

Provided the rewrite theory $(\Sigma, E \cup B, R, \phi)$ is *topmost* (that is, all rewrites take place at the root of a term), or, as in the case of AC rewriting of object-oriented systems, $\mathcal{R}$ is "essentially topmost," and the rules R are coherent with E modulo B, narrowing with the rules R modulo the equations $E \cup B$ gives a constructive, sound, and complete method to solve reachability problems of the form $\exists X\ t \longrightarrow^* t'$, that is, such a problem has an affirmative answer if and only if we can find a finite narrowing sequence modulo $E \cup B$ of the form $t \rightsquigarrow^* u$ such that u and t' are unifiable modulo $E \cup B$ [111]. The method is *constructive*, because instantiating t with the composition of the unifiers for each step in the narrowing sequence, plus a $E \cup B$-unifier for $u = t'$, gives us a concrete rewrite sequence witnessing the existential formula.

Of course, narrowing with R modulo $E \cup B$ requires performing $E \cup B$-unification at each narrowing step. As explained in Section 6.2, $E \cup B$-unification can itself be performed by folding variant narrowing with the equations E modulo B, provided E is convergent modulo B. Therefore, in performing symbolic reachability analysis in a rewrite theory $(\Sigma, E \cup B, R)$ there are *two levels of narrowing* involved: (i) narrowing with R modulo $E \cup B$ for reachability, and (ii) folding variant narrowing with E modulo B to compute the $E \cup B$-unifiers needed for narrowing with R modulo $E \cup B$.

Maude provides a `vu-narrow` command similar to the `search` command for rewriting. Specifically, `vu-narrow` searches in a breadth-first manner for a substitution instance of the given goal pattern that can be reached by rewriting from a substitution instance of the given pattern for initial states. The general form of the command is: `vu-narrow` t `=>`$\diamond$ t' `.` where t is the pattern of initial states and t' is the goal pattern. The $\diamond$ symbol is a place holder for the options: $\diamond$ = `1` (exactly one rewrite step), $\diamond$ = `+` (one or more steps), $\diamond$ = `*` (zero or more steps), and $\diamond$ = `!` (terminating states). Since the narrowing search may either never terminate and/or find an infinite number of solutions, two *bounds* can be added to a `vu-narrow` command: one bounding the number of solutions requested, and another bounding the depth of the rewrite steps from the initial term t (see below).

Let us illustrate the power of performing narrowing-based reachability analysis modulo variant equations and axioms, including associativity. Consider the specification of a generic grammar interpreter in Maude, based on [3], given in Figure 24. We define a symbol `_@_` to represent the interpreter configurations, where the first underscore represents the current string (of terminal and non-terminal symbols), and the second underscore stands for the considered grammar. For simplificity, we provide four non-terminal symbols `S`, `A`, `B`, and `C` for sort `NSymbol` and four terminal symbols 0, 1, 2, and the finalizing mark `eps` (the empty string) for sort `TSymbol`, but a parametric specification would have been more appropriate.
The `narrowing` attribute specifies that the rule will be used for narrowing-based reachability analysis.

Note the important fact that the string concatenation symbol `__` is not just `assoc`, but has also `eps` as its *identity* element. This means that in each narrowing step with the interpreter's rule equational unification must be per-

```
mod GRAMMAR is
 sorts Symbol NSymbol TSymbol String Production Grammar Conf .
 subsorts TSymbol NSymbol < Symbol < String .
 subsort Production < Grammar .
 ops 0 1 2 eps : -> TSymbol .
 ops S A B C : -> NSymbol .
 op _@_ : String Grammar -> Conf .
 op _->_ : String String -> Production .
 op __ : String String -> String [assoc id: eps] .
 op mt : -> Grammar .
 op _;_ : Grammar Grammar -> Grammar [assoc comm id: mt] .
 vars  L1 L2 U V : String .
 var G : Grammar .
 var N : NSymbol .
 var T : TSymbol .
 rl ( L1 U L2 @ (U -> V) ; G) => ( L1 V L2 @ (U -> V) ; G) [narrowing] .
endm
```

Figure 24: `GRAMMAR` module

formed modulo AU and not just modulo A. This is not directly supported by the order-sorted B-unification of Section 6.1, but *is* supported by the variant-based $E \cup B$-unification of Section 6.3. That is, AU-unification is achieved by transforming the identity property into the FVP variant equations:

```
eq eps U = U [variant] .        eq U eps V = U V [variant] .        eq V eps
     = V [variant] .
```

The interpreter can be used in two ways thanks to narrowing: to generate words of the given grammar, but also to parse a given string (see [20] for further references on this topic). Generating the words of a given grammar is defined by rewriting the configuration (`S` @ Γ) into (st @ Γ) where st is a string of terminal symbols using the rules of the grammar Γ. For example, we have the following search query associated to a context-free grammar defining the language $0^n 1^n$:

```
Maude> vu-narrow [4] S @ (S -> eps) ; (S -> 0 S 1)
                 =>! U @ (S -> eps) ; (S -> 0 S 1) .

Solution 1       Solution 2       Solution 3         Solution 4
U --> eps        U --> 0 1        U --> 0 0 1 1      U --> 0 0 0 1 1 1
```

Parsing a string st according to a given grammar Γ is defined by narrowing the configuration (N @ Γ) into (st @ Γ) where N is a logical variable denoting a non-terminal symbol. For example, we have the following search query:

```
Maude> vu-narrow [1] N       @ (S -> eps) ; (S -> 0 S 1)
                 =>* 0 0 1 1 @ (S -> eps) ; (S -> 0 S 1) .
Solution 1
N --> S
```

Moreover, we can use narrowing to answer a more complex question: *What is the missing production so that the string "0 0 1" is parsed into the non-terminal symbol `S`?*

```
Maude> vu-narrow [1] S     @ (N -> T) ; (S -> eps) ; (S -> 0 S 1)
                 =>* 0 0 1 @ (N -> T) ; (S -> eps) ; (S -> 0 S 1) .
Solution 1
N --> S ;
T --> 0
```

And we can use any grammar, e.g. a Type-0 grammar defining the language $0^n 1^n 2^n$.

```
Maude> vu-narrow [1] N @ (S -> eps) ; (S -> 0 S B C) ; (C B -> B C) ;
                         (0 B -> 0 1) ; (1 B -> 1 1) ; (1 C -> 1 2) ;
                         (2 C -> 2 2)
       =>* 0 0 1 1 2 2 @ (S -> eps) ; (S -> 0 S B C) ; (C B -> B C) ;
                         (0 B -> 0 1) ; (1 B -> 1 1) ; (1 C -> 1 2) ;
                         (2 C -> 2 2) .
Solution 1
N --> S
```

Note that we must restrict the search in the previous narrowing-based search commands, because narrowing does not terminate for these reachability problems. However, it is extremely important that no warning about A-unification incompleteness is shown, ensuring that the symbolic analysis is complete modulo AU, despite associative unification being infinite for some uncommon cases. The key reason is that string variables (`L1`, `L2`, and `U`) in the transition rule are *linear* (`L1` and `L2`) or under order-sorted restrictions (`U`).

7.1. Logic Programming as Symbolic Reachability

In this section we show how narrowing-based symbolic reachability analysis can be used to provide a very simple alternative implementation of logic programming. The key idea is that there is a simple theory transformation:

$$R[\_] : \mathit{HornLogicTheories} \to \mathit{RewriteTheories}$$

so that given a logic program H we obtain an associated rewrite theory $R[H]$ such that any query for H can be solved by a corresponding `vu-narrow` search command for $R[H]$. We explain and illustrate below this theory transformation.

All theories $R[H]$ for any logic program H extend the following module `LP-SEMANTICS`, that imports the `LP-SYNTAX` module, by adding to it the rules of $R[H]$. We no longer need any auxiliary unification or renaming machinery, since narrowing performs those automatically.

```
mod LP-SEMANTICS is
  protecting LP-SYNTAX .

  sort PredicateList .
  op nil : -> PredicateList .
  op _,_ : Predicate PredicateList -> PredicateList .

  var PL : PredicateList .
  vars X Y Z : Term .

  sort Configuration .
  op <_> : PredicateList -> Configuration .
```

For each Horn theory H, $R[H]$ just adds to the above signature the rewrite rules into which the Horn clauses of H are transformed. Specifically, each logic clause P `:-` $P_1, \ldots, P_n$ is transformed into the rewrite rule $\langle P, PL\rangle \to \langle P_1, \ldots, P_n, PL\rangle$, where PL is a new variable of sort `PredicateList` and where the leftmost predicate P is replaced by $P_1, \ldots, P_n$.

Let us illustrate how this transformation is used by means of our running logic programming example.

Example 11 (Symbolic Search LP-evaluation). *For H the logic program of Example 9, $R[H]$ adds to* `LP-SEMANTICS` *the following rewrite rules:*

```
  rl < 'mother('jane, 'mike),PL >  => < PL > [narrowing] .
  rl < 'mother('sally, 'john),PL > => < PL > [narrowing] .
  rl < 'father('tom, 'sally),PL >  => < PL > [narrowing] .
  rl < 'father('tom, 'erica),PL >  => < PL > [narrowing] .
  rl < 'father('mike, 'john),PL >  => < PL > [narrowing] .
  rl < 'parent(X,Y),PL > => < 'father(X,Y),PL > [narrowing] .
  rl < 'parent(X,Y),PL > => < 'mother(X,Y),PL > [narrowing] .
  rl < 'sibling(X,Y),PL > => < 'parent(Z,X),'parent(Z,Y),PL > [narrowing
      nonexec] .
  rl < 'relative(X,Y),PL > => < 'parent(X,Z),'parent(Z,Y),PL > [narrowing
      nonexec] .
  rl < 'relative(X,Y),PL > => < 'sibling(X,Z),'relative(Z,Y),PL > [narrowing
      nonexec] .
```

Note that Maude requires that rules with extra variables in the right-hand side must be labeled with the `nonexec` *keyword, even though the narrowing machinery uses them to perform narrowing steps without any problem.*

We can now evaluate different queries for our example logic program H by giving corresponding `vu-narrow` *search command for $R[H]$ with goal* `< nil >`*.*

First, whether Sally and Erica are sisters; the associated reachability graph is finite and no bound is necessary.

```
Maude> vu-narrow < 'sibling('sally,'erica),nil > =>* < nil > .
Solution 1
```

Which are the siblings of Erica? Sally and herself.

```
Maude> vu-narrow < 'sibling(X,'erica),nil > =>* < nil > .
Solution 1
X --> 'sally

Solution 2
X --> 'erica
```

How many possible siblings are there? Sally and Sally, Sally and Erica, Erica and Sally, Erica and Erica, John and John, and Mike and Mike.

```
Maude> vu-narrow < 'sibling(X,Y),nil > =>* < nil > .
Solution 1
X --> 'sally
Y --> 'sally

Solution 2
X --> 'sally
Y --> 'erica

Solution 3
X --> 'erica
Y --> 'sally

Solution 4
X --> 'erica
Y --> 'erica
```

```
Solution 5
X --> 'john
Y --> 'john

Solution 6
X --> 'mike
Y --> 'mike

Solution 7
X --> 'john
Y --> 'john
```

Are Jane and John relatives? Yes

```
Maude> vu-narrow < 'relative('jane,'john),nil > =>* < nil > .
Solution 1
```

Which are the relatives of John? Tom and Jane.

```
Maude> vu-narrow [2] < 'relative(X,'john),nil > =>* < nil > .
Solution 1
X --> 'tom

Solution 2
X --> 'jane
```

As explained in Section 3.1, this last call produces an infinite narrowing search, so we must restrict the search, asking for two solutions only.

In retrospect, the deep connection between logic programming and narrowing-based reachability analysis is not surprising at all: both are based on *unification*, and Horn clauses can easily be transformed into rules so that solving logic programming queries just *becomes* narrowing search for the `< nil >` list of atomic predicates. But this leaves two pending questions: (1) how can we *mechanize* the $H \mapsto R[H]$ transformation; and (2) how can we obtain a *programming environment* for logic programming in Maude based on narrowing? Both questions can be easily answered by a very powerful Maude feature, namely, *reflection*. In fact, Sections 8.1 and 8.2 will respectively answer questions (1) and (2) not just for logic programming, but for the much more general functional-logic programming language Eqlog [64].

Further Reading. Narrowing-based symbolic reachability analysis of concurrent systems was first studied and proved complete in [111]. To ensure that the narrowing tree is finitely branching, and for performance reasons, we have here assumed that in the topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, (i) the equations $E \cup B$ are FVP, and (ii) the rules R are unconditional. This of course restricts substantially the class of concurrent systems that can be symbolically model checked by narrowing. As explained in [**?**], using a semantics-preserving theory transformation and the concept of *constrained narrowing*, restrictions (i)–(ii) can be dropped and a much wider class of systems can be symbolically model checked. Under the same just-mentioned assumptions (i)–(ii) on $\mathcal{R}$ it is possible to symbolically model check not only invariants using `vu-narrow`, but arbitrary LTL formulas using Maude's LTL logical model checker [7], available in the Maude web page.

8. Reflection, `META-LEVEL`, and Meta-Interpreters

Rewriting logic is reflective [30, 31], in the sense that important aspects of its metatheory can be represented at the object level in a consistent way. That is, the object-level representation correctly simulates the relevant metatheoretic aspects, just as a universal Turing machine correctly simulates any other Turing machine, including itself. This fact is systematically used in the design and implementation of the Maude language, making the metatheory of rewriting logic accessible to the user in a clear, principled, and efficient way.

Rewriting logic being reflective means that there is a finitely presented rewrite theory $\mathcal{U}$ in which we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $\langle\overline{\mathcal{R}}, \overline{t}\rangle$, in such a way that we have the following equivalence:

$$\mathcal{R} \vdash t \longrightarrow^* t' \;\;\Leftrightarrow\;\; \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \longrightarrow^* \langle\overline{\mathcal{R}}, \overline{t'}\rangle$$

where $\mathcal{R} \vdash t \longrightarrow^* t'$ denotes that t rewrites into t' using the rewrite theory $\mathcal{R}$. Since $\mathcal{U}$ is representable in itself, we can have an arbitrary number of levels of reflection, giving place to what is known as a "reflective tower":

$$\mathcal{R} \vdash t \rightarrow^* t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \rightarrow^* \langle\overline{\mathcal{R}}, \overline{t'}\rangle \;\Leftrightarrow\; \mathcal{U} \vdash \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t}\rangle}\rangle \rightarrow^* \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t'}\rangle}\rangle \ldots$$

This section explains how this is achieved in Maude through its predefined `META-LEVEL` and `META-INTERPRETER` modules. While the `META-LEVEL` module provides a purely functional access to key functionality of the universal theory $\mathcal{U}$, the `META-INTERPRETER` module can also handle reflective object-oriented computations that interact with the outside world. Indeed, the meta-interpreter manager and the created meta-interpreters are external objects like internet sockets, files, or standard I/O (see Section 5.2).

In a naive implementation of reflection, each step up the above reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible, so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary. In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`. This module includes definitions of sorts and operations for representing every element in a structured specification. For example, terms are metarepresented as elements of a data type `Term` of terms; modules are metarepresented as terms in a data type `Module` of modules; and views are metarepresented as terms in a data type `View` of views. `META-LEVEL` also contains so-called *descent functions* that use the equivalences in the reflective tower from right to left to lower as much as possible the level of reflective computation for boosting performance. In many cases, the performance cost is just a simple, linear change of data representation before and after the given "descended" computation. In fact, virtually all Maude commands plus many meta-level operations such as unification, matching, rule application, rewriting, search, and so on, are supported one level up the hierarchy as descent functions. For example, `metaReduce`, `metaRewrite`, `metaApply`, and `metaMatch` are some of these descent functions in `META-LEVEL`. Furthermore, reflective operations like `upModule`, `upTerm`, `downTerm`, and other similar ones allow moving various kinds of entities one level up or down in the reflective hierarchy.

Giving a *full* account of the `META-LEVEL` module is beyond the scope of this paper. Full details can be found in [24, 28]. However, we give here a taste of how reflection is supported in Maude by: (1) explaining how a term t is a rewrite theory $\mathcal{R}$ is meta-represented as a meta-term $\bar{t}$ of sort `Term`, (2) explaining how a rewrite theory $\mathcal{R}$ (resp. an equational theory $\mathcal{E}$) is meta-represented as a term $\overline{\mathcal{R}}$ (resp. $\overline{\mathcal{E}}$) of sort `Module`, and (3) illustrating in Section 8.1 how easy it is to define *program transformations* by reflection by means of an example transformation that mechanizes within Maude the Eqlog functional-logic language [64].

8.0.1. The `META-TERM` *module*

In the `META-TERM` submodule of `META-LEVEL`, sorts and kinds are metarepresented as terms in subsorts `Sort` and `Kind` of the sort `Qid` of quoted identifiers. Since characters parentheses, brackets and commas break identifiers in Maude, they must be *scaped* with back quotes. For example, `'NzNat`, `'Map`{Int`,String`}`, and `'Map`{Int`,Tuple`{String`,Set`{Rat`}`}`}` are terms of sort `Sort`. Similarly, `'`[Bool`]`, `'`[List`{Int`}`]` and `'`[NzNat`,Zero`,Nat`]` are valid elements of the sort `Kind`.

A term t is meta-represented as a so-called meta-term $\bar{t}$ of the data type `Term` of terms. The base cases in the metarepresentation of terms are given by subsorts `Constant` and `Variable` of the sort `Qid`. Constants are quoted identifiers that contain the constant's name and its type separated by a '`.`', e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a '`:`', e.g., `'N:Nat`. Appropriate selectors then extract their names and types.

A (non-constant) *function symbol* is meta-represented as a quoted identifier of sort `Qid`. A *term* different from a constant or a variable is meta-represented by applying an operator symbol to a nonempty list of meta-terms using the constructor

```
op _[_] : Qid NeTermList -> Term [ctor] .
```

For example, the natural number term `s(N) + M` is meta-represented as the meta-term `'_+_['s['N:Nat], 'M:Nat]`.

8.0.2. The `META-MODULE` *module*

In the submodule `META-MODULE` of `META-LEVEL`, which imports `META-TERM`, functional and system modules, as well as functional and system theories, are metarepresented in a syntax very similar to their original user syntax. Given meta-representations of sorts, operations, equations, membership axioms, and rules, modules and theories are meta-represented as terms of sort `Module` (and corresponding subsorts, like `FModule` for functional modules and `SModule` for system modules). For example, a system module is meta-represented using the following constructor:

```
op mod_is_sorts_._____endm : Header ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
     -> SModule [ctor gather (& & & & & & & &)] .
```

Sort `Header` can take as values just an identifier in the case of non-parameterized modules or an identifier together with a list of parameter declarations in the case of a parameterized module. Let us get a taste for how each of the different elements in modules and theories are meta-represented by looking at how equations are meta-represented.

```
sorts Equation EquationSet .
subsort Equation < EquationSet .
op eq_=_[_]. : Term Term AttrSet -> Equation [ctor] .
op none : -> EquationSet [ctor] .
op __ : EquationSet EquationSet -> EquationSet [ctor assoc comm id: none] .
```

Similar definitions allow us to represent the rest of the components of modules. To get a feeling about the similarity between the object-level and meta-level notations, let us consider the metarepresentation of the module on the left as the term (called a *meta-module*) displayed on the right:

```
fmod NAT is                            fmod 'NAT is
  pr BOOL .                              protecting 'BOOL .
  sorts Zero Nat .                       sorts 'Zero ; 'Nat .
  subsort Zero < Nat .                   subsort 'Zero < 'Nat .
  op 0 : -> Zero [ctor] .                op '0 : nil -> 'Zero [ctor] .
  op s : Nat -> Nat [ctor] .             op 's : 'Nat -> 'Nat [ctor] .
  op _+_ : Nat Nat -> Nat [comm] .       op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  vars N M : Nat .                       --- no variable declarations
  --- no mbs                             none
  eq 0 + N = N .                         eq '_+_['0.Nat, 'N:Nat] = 'N:Nat .
  eq s(N) + M = s(N + M) .               eq '_+_['s['N:Nat], 'M:Nat]
                                           = 's['_+_['N:Nat, 'M:Nat]] .
endfm                                  endfm
```

To prepare the ground for our program transformation example in Section 8.1, just think for a moment about what a program transformation is in its simplest possible form, and why reflection should provide a powerful way of *meta-programming* such transformations. In Maude, a program is a rewrite theory $\mathcal{R}$. Therefore, the simplest kind of program transformation we can think of is some kind of function, say, *Tr*, that maps any rewrite theory $\mathcal{R}$ to its transformed theory $Tr(\mathcal{R})$. But *where* does this function exits? In a stratosphere called the *meta-level* of rewriting logic. What reflection does is to bring such a stratosphere down to earth, namely, down to the `META-LEVEL` module. Of course, for the program transformation *Tr* to be of any use at all, it should be effective, that is, it should be a *computable* function. But we know by the meta-theorem of Bergstra and Tucker [13] that *any* computable function can be defined by a *convergent*, finite set of equations. Since by reflection we already have an algebraic data type of rewrite theories, namely, the data type defined by the `META-MODULE` functional module, this all means that we can *meta-program* any program transformation *Tr* of our choice as an *equationally-defined function*

```
op Tr : SModule -> Smodule .
```

in a functional module extending `META-MODULE`. Let us see all this for Eqlog!

8.1. A Program Transformation for Eqlog

Program transformation is one of the applications of meta-programming. The following example illustrates the power of program transformations in a way that generalizes the program transformation $H \mapsto R[H]$ from Horn clause theories to rewrite theories defined in Section 7.1 and illustrated in Example 11.

The generalization has to do with considering a much more general class of Horn theories, namely, *order-sorted Horn theories with equality*, which are the theories on which the Eqlog [64] functional-logic language is based. Such theories have the form: $((\Sigma, \Pi), E \cup B \cup H)$, where $(\Sigma, E \cup B)$ is a convergent order-sorted equational theory that, to make sure $E \cup B$-unification terminates, we will assume FVP, and H is a collection of *Horn clauses* defined on the signature Π of *predicate symbols*. We could easily define in Maude a data type whose terms are exactly (reflective versions of) such order-sorted Horn theories with equality, and then we could define by reflection the transformation mapping any such theory to a corresponding meta-module term in Maude. But there is a shortcut that we will take to ease the presentation.

The shortcut has to do with the fact that each order-sorted Horn theory with equality $((\Sigma, \Pi), E \cup B \cup H)$ can be transformed into a *semantically equivalent* order-sorted *equational theory* of the form $(\Sigma \cup \Pi, E \cup B \cup E_H)$, where the predicates Π have been transformed into additional *function symbols*, and the Horn clauses H into additional *conditional equations* E_H by: (i) adding a fresh new sort *Pred* of predicates to Σ having a constant $\top$ denoting "truth," (ii) turning each predicate p in Π having argument sorts $s_1 \dots s_n$ into a function symbol $p : s_1 \dots s_n \to \mathit{Pred}$, and (iii) transforming each Horn clause $p(\overline{u}) \Leftarrow p_1(\overline{u_1}) \wedge \dots \wedge p_n(\overline{u_n})$ into the conditional equation $p(\overline{u}) = \top \Leftarrow p_1(\overline{u_1}) = \top \wedge \dots \wedge p_n(\overline{u_n}) = \top$. For simplicity we will assume that each Eqlog theory T has been specified as an equational theory of the form $T = (\Sigma \cup \Pi, E \cup B \cup E_H)$. This has the advantage of allowing us to express T inside Maude as a functional module, so that our desired transformation $T \mapsto R[T]$ turning T into a rewrite theory can be defined as a meta-level function:

```
op R[_] : FModule -> SModule .
```

To illustrate these ideas, let us consider an Eqlog program that extends that of Example 9 by adding age information for the relatives in the example and an order predicate to compare ages. In its functional version such an Eqlog program can be specified as the functional module in Figure 25.

The transformation $T \mapsto R[T]$ is in essence very simple. It has the form $R : (\Sigma \cup \Pi, E \cup B \cup E_H) \mapsto (\Sigma \cup \Pi \cup \Omega, E \cup B, R[H] \cup R_{eq})$, where Ω adds new sorts `PredList` and `Configuration` and operator declarations `<_>` of sort `Configuration` and `nil` and `_,_` of sort `PredList` just as we did in the $H \mapsto R[H]$ transformation of Section 7.1, and the Horn clauses H (here expressed as conditional equations E_H but this is immaterial) are transformed into rewrite rules exactly as in the $H \mapsto R[H]$ transformation. Furthermore, for each connected component, `[s]`, other than that of `Pred`, a binary equality predicate `_==_ : [s] [s] -> Pred` is added to Ω, and a rule defining this predicate for

```
fmod EXAMPLE is
  protecting TRUTH-VALUE .
  sorts Person Nat Pred .

  ops jane tom sally mike john erica : -> Person [ctor] .

  op T : -> Pred  [ctor] .    *** true
  op mother : Person Person -> Pred [ctor] .
  op father : Person Person -> Pred [ctor] .
  op sibling : Person Person -> Pred [ctor] .
  op parent : Person Person -> Pred [ctor] .
  op relative : Person Person -> Pred [ctor] .

  vars X1 X2 X3 : Person .

  *** Horn Clauses as conditional equations:
  eq mother(jane, mike) = T .
  eq mother(sally, john) = T .
  eq father(tom, sally) = T .
  eq father(tom, erica) = T .
  eq father(mike, john) = T .
  ceq sibling(X1, X2) = T
    if parent(X3, X1) = T /\ parent(X3, X2) = T [nonexec] .
  ceq parent(X1, X2) = T if father(X1, X2) = T .
  ceq parent(X1, X2) = T if mother(X1, X2) = T .
  ceq relative(X1, X2) = T
    if parent(X1, X3) = T /\ parent(X3, X2) = T [nonexec] .
  ceq relative(X1, X2) = T
    if sibling(X1, X3) = T /\ relative(X3, X2) = T [nonexec] .

  ops 0 1 : -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  op _>_ : Nat Nat -> Bool .
  op _>=_ : Nat Nat -> Bool .

  vars n m k : Nat .

  eq n + m + 1 > n = true [variant] .
  eq n > n + m = false [variant] .

  eq n + m >= n = true [variant] .
  eq n >= n + m + 1 = false [variant] .

  op age : Person -> Nat .
  eq age(tom)
    = 1 + 1 + ... + 1 [variant] . *** 50
  eq age(sally)
    = 1 + 1 + ... + 1 [variant] . *** 30
  eq age(john)
    = 1 + 1 + ... + 1 [variant] . *** 10
  eq age(jane)
    = 1 + 1 + ... + 1 [variant] . *** 52
  eq age(mike)
    = 1 + 1 + ... + 1 [variant] . *** 32
  eq age(erica)
  = 1 + 1 + 1 ... + 1 [variant] . *** 28
endfm
```

Figure 25: Eqlog program extending the relatives program in Example 9 (notice the ellipses)

equalities of that kind: `rl < X:[s] == X:[s],PL > => < PL >` is added to the set of rules (these are the rules denoted R_{eq}).

Given self-explanatory auxiliary functions `addOps`, `setName`, `setEqs`, `setRls`, `getSorts`, `getEqs`, `getRls`, and `getName`, the following equation implements the $T \mapsto R[T]$ transformation:

```
op eqlogTransform : FModule -> SModule .
eq eqlogTransform(M)
  = addSorts('PredList ; 'Configuration,
      addOps(
        op 'nil : nil -> 'PredList [none] .
        op '_`,_ : 'Pred 'PredList -> 'PredList [none] .
        op '<_> : 'PredList -> 'Configuration [none] .
        mkEqOps(getSorts(M)),
        transformEqs(
          getEqs(M),
          setEqs(
            setRls(setName(M, qid("R[" + string(getName(M)) + "]")),
              mkEqRls(getSorts(M))),
            none),
          M))) .
```

Auxiliary functions `mkEqOps` and `mkEqRls` create, given a set of sorts, the operator declarations and equations for `_==_` as explained above.

```
op mkEqOps : SortSet -> OpDeclSet .
eq mkEqOps(S ; SS)
  = if S == 'Pred
    then none
    else op '_==_ : kind(S) kind(S) -> 'Pred [none] .
    fi
    mkEqOps(SS) .
eq mkEqOps(none) = none .

op mkEqRls : SortSet -> RuleSet .
eq mkEqRls(S ; SS)
  = if S == 'Pred
    then none
    else rl '<_>['_`,_['_==_[qid("X:" + string(kind(S))),
                            qid("X:" + string(kind(S)))], 'PL:PredList]]
           => '<_>['PL:PredList] [narrowing] .
    fi
    mkEqRls(SS) .
eq mkEqRls(none) = none .
```

The operation `transformEqs` transforms equations of sort `Pred` into the corresponding rules:

```
op transformEqs : EquationSet Module Module -> Module .
op transformCd : EqCondition -> Term .
ceq transformEqs(eq T = 'T.Pred [none] . Eqs, M, M')
  = transformEqs(Eqs,
      addRls(rl '<_>['_`,_[T, 'PL:PredList]]
               => '<_>['PL:PredList] [narrowing] ., M),
      M')
  if leastSort(M', T) = 'Pred .
ceq transformEqs(ceq T = 'T.Pred if Cd [Atts] . Eqs, M, M')
  = transformEqs(
         Eqs,
         addRls(rl '<_>['_`,_[T, 'PL:PredList]]
                  => '<_>[transformCd(Cd)] [Atts narrowing] ., M),
         M')
  if leastSort(M', T) = 'Pred .
eq transformEqs(Eqs, M, M') = addEqs(Eqs, M) [owise] .

ceq transformCd(T = 'T.Pred /\ Cd)
```

```
    = '_`,_[T, transformCd(Cd)]
    if Cd =/= nil .
  eq transformCd(T = 'T.Pred) = '_`,_[T, 'PL:PredList]  .
```

Notice that rules are added to the module as they are generated. The second module does not change, it is used just for checking sorts.

Example 12 (Eqlog Example as Narrowing Search). *Consider the functional module* EXAMPLE *in Figure 25, which is the already-discussed Eqlog program extending the relatives logic program of Example 9 with an* age *operation and order predicates to compare natural numbers. Its transformed system meta-module* eqlogTransform($\overline{\texttt{EXAMPLE}}$) *obtained using the* eqlogTransform *meta-level function has (meta-represented)* unconditional *rewrite rules such as the following ones:*

```
   rl < mother(jane, mike), PL:PredList >
     => < PL:PredList > .
   rl < sibling(X1,X2), PL:PredList >
     => < PL:PredList, parent(X3, X1), parent(X3, X2) > [nonexec] .
```

The system meta-module eqlogTransform($\overline{\texttt{EXAMPLE}}$) *can then be used at the metalevel to perform Eqlog-based symbolic computation for this example using narrowing search*[10]*. For example, we can then use the* metaNarrowingSearch *operation to find persons with a father and a mother in the transformed module:*

```
red metaNarrowingSearch(
      eqlogTransform(upModule('EXAMPLE, true)),
      '<_>['_`,_['father['X:Person, 'Y:Person],
                 '_`,_['mother['Z:Person, 'Y:Person], 'nil.PredList]]],
      '<_>['nil.PredList],
      '*,
      unbounded,
      'none,      ---- vu-narrow folding strategy
      0) .
result NarrowingSearchResult: {
  '<_>['nil.PredList],
  'Configuration,
  ('X:Person <- 'mike.Person ;
   'Y:Person <- 'john.Person ;
   'Z:Person <- 'sally.Person),
  '%,
  (none).Substitution,
  '#
}
```

Or to find out that there are no fathers younger than their children:

```
red metaNarrowingSearch(
      eqlogTransform(upModule('EXAMPLE, true)),
      '<_>['_`,_['father['X:Person, 'Y:Person],
                 '_`,_['_==_['_>_['age['Y:Person], 'age['X:Person]], 'true.
                      Bool],
                        'nil.PredList]]],
      '<_>['nil.PredList],
      '*,
      unbounded,
```

[10] An alternative way of representing Eqlog programs can be found in [56] using system modules and narrowing search and also in [57] using functional modules and folding variant narrowing.

```
      'none,      ---- vu-narrow folding strategy
      0) .
result NarrowingSearchResult?: (failure).NarrowingSearchResult?
```

8.2. An Eqlog Execution Environment

Maude provides meta-programming facilities for the generation of execution environments for a wide variety of languages and logics. We explain here how these facilities may be used to develop a user-friendly notation for the introduction of Eqlog programs. We have extended Full Maude with a new module expression to be able to use Eqlog programs as functional modules as in Example 9 and corresponding commands to execute queries on them.

Full Maude [28] is an extension of Maude written in Maude itself using its reflective capabilities. It was developed as a place in which to experiment with new features, and provide facilities not yet available in the core implementation. Indeed, many of the features now available in Core Maude, like strategies, unification, variants, narrowing, parameterized modules, views, and module expressions like summation, renaming and instantiation, were available in Full Maude long before they were available in Maude (see, e.g., [44, 38]). This same setting represents a perfect place to add new features with which to experiment or develop new prototypes.

The interested reader may find at `http://maude.lcc.uma.es/maude28` a module extending Full Maude that provides a new module expression `R[_]` to transform an Eqlog program T already entered into Maude as a functional module into the rewrite theory $R[T]$ defined in Section 8.1, and a command `solve[_]_.` to get the first n solutions to a query for such an Eqlog program. The extension has been performed as in many previous cases. Please, see guidelines in [48] or [28, 24].

Once the module expression is available, it can be used as any other module expression in Maude, in importation declarations in other modules or in commands. For example, given the `EXAMPLE` module one can select the generated module with

```
(select R[EXAMPLE] .)
```

And then, at the object level, one can write commands of the form:

(solve [n] $A1$,...,An .)

to ask for the first n solutions to the query $A1,\ldots,An$ where the Ai are predicate atoms, including equality predicates of the form t `==` t'.

We can now solve the queries in Example 12 in a more user-friendly syntax:

```
(solve father(X:Person, Y:Person), mother(Z:Person, Y:Person), nil .)

Solution 1
state: < nil >
accumulated substitution:
X:Person --> mike ;
Y:Person --> john ;
Z:Person --> sally
variant unifier: empty substitution

No more solutions.
```

```
(solve father(X:Person, Y:Person),
       age(Y:Person) > age(X:Person) == true,
       nil .)

No more solutions.
```

8.3. Meta-interpreters

The `META-LEVEL` module is *purely functional*. This is because all its *descent* functions are deterministic, even though they may manipulate intrinsically non-deterministic entities such as rewrite theories. For example, the `metaSearch` descent funtion with a bound of, say, 3, is entirely deterministic, since given the meta-representations $\overline{\mathcal{R}}$ of the desired system module and $\bar{t}$ of the initial term plus the bound 3, the result yielded by `search` for $\mathcal{R}$, t and 3 at the object level, and therefore by `metaSearch` at the meta-level, is uniquely determined.

Although `META-LEVEL` is very powerful, its purely functional nature means that it has *no notion of state*. Therefore reflective applications where *user interaction* in a state-changing manner is essential require using `META-LEVEL` in the context of additional features supporting such interaction. Until recently, all such reflective interactions were mediated by the built-in `LOOP-MODE` module [28]: a simple read-eval-print loop where a Maude user can interact from the terminal with a Maude module M already stored in Maude through an object (the *state* of `LOOP-MODE`) consisting of a 3-tuple that holds a Maude term t in module M —thought of as the current "internal state" of the loop— together with input and output buffers. The user interactions do change the state of that 3-tuple by consuming user input, producing output and possibly changing the internal state t to a new state t' according to the user-given rewrite rules defining the desired interaction. For example, Full Maude, the Eqlog extension of it presented in Section 8.2, and many other interactive Maude tools use suitable extensions of `META-LEVEL` and `LOOP-MODE` to support user interaction. This is adequate for many purposes, but limits the type of interactions to simple real-eval-print ones. Much more flexible kinds of reflective interactions are possible by means of Maude's new *meta-interpreters* feature, in which Maude interpreters are encapsulated as external objects and can reflectively interact with both other Maude interpreters and with various other external objects, including the user.

Conceptually a meta-interpreter is an external object that is an independent Maude interpreter, complete with module and view databases, which sends and receives messages. The module `META-INTERPRETER` in Maude's standard prelude contains command and reply messages that cover almost the entirety of the Maude interpreter. For example, it can be instructed to insert or show modules and views, or carry out computations in a named module. As response, the meta-interpreter replies with messages acknowledging operations carried out or containing results. Meta-interpreters can be created and destroyed as needed, and because a meta-interpreter is a complete Maude interpreter, it can host meta-interpreters itself and so on in a tower of reflection. Furthermore the original `META-LEVEL` functional module can itself be used from inside a meta-interpreter.

```
mod RUSSIAN-DOLLS is
  extending META-INTERPRETER .

  op me : -> Oid .
  op User : -> Cid .
  op depth:_ : Nat -> Attribute .
  op computation:_ : Term -> Attribute .

  vars X Y Z : Oid .
  var AS : AttributeSet .
  var N : Nat .
  var T : Term .

  op newMetaState : Nat Term -> Term .
  eq newMetaState(0, T) = T .
  eq newMetaState(s N, T)
    = upTerm(
        <>
        < me : User | depth: N, computation: T >
        createInterpreter(interpreterManager, me, none)) .

  rl < X : User | AS >
     createdInterpreter(X, Y, Z)
  => < X : User | AS >
     insertModule(Z, X, upModule('RUSSIAN-DOLLS, true)) .
  rl < X : User | depth: N, computation: T, AS >
     insertedModule(X, Y)
  => < X : User | AS >
     erewriteTerm(Y, X, unbounded, 1, 'RUSSIAN-DOLLS, newMetaState(N, T)) .
endm
```

Figure 26: Nested meta-interpreter example

The meta-representation of terms, modules and views is shared with the `META-LEVEL` functional module. The API to meta-interpreters defined in the `META-INTERPRETER` module includes several sorts and constructors, a built-in object identifier `interpreterManager` and a large collection of command and response messages. The `interpreterManager` object identifier refers to a special external object that is responsible for creating new meta-interpreters in the current execution context. Such meta-interpreters have object identifiers of the form `interpreter(n)` for natural number n.

Example 13. *Let us illustrate the flexibility and generality of meta-intepreters with a short example. The example, which we call* `RUSSIAN-DOLLS` *after the Russian nesting dolls, is shown in Figure 26. It performs a computation in a meta-interpreter that itself exists in a tower of meta-interpreters nested to a user-definable depth and requires only two equations and two rules.*

The visible state of the computation resides in a Maude object of identifier `me` *and class* `User`*. The object holds two values in respective attributes: the depth of the meta-interpreter, which is recorded as a* `Nat`*, with* `0` *as the top level, and the computation to perform, which is recorded as a* `Term`*.*

The operator `newMetaState` *takes a depth and a meta-term to evaluate. If the depth is zero, then it simply returns the meta-term as the new meta-state. Otherwise a new configuration is created, consisting of a portal (needed for rewriting with external objects to locate where messages exchanged with ex-*

ternal objects leave and enter the configuration), the user-visible object holding the decremented depth and computation, and a message directed at the `interpreterManager` *external object, requesting the creation of a new meta-interpreter, and this configuration is lifted to the metalevel using the built-in* `upTerm` *operator imported from the functional metalevel.*

The first rule of the module handles the `createdInterpreter` *message from* `interpreterManager`*, which carries the object identifier of the newly created meta-interpreter. It uses* `upModule` *to lift its own module,* `RUSSIAN-DOLLS`*, to the metalevel and sends a request to insert this meta-module into the new meta-interpreter. The second rule handles the* `insertedModule` *message from the new meta-interpreter. It calls the* `newMetaState` *operator to create a new meta-state and then sends a request to the new meta-interpreter to perform an unbounded number of rewrites, with external object support and one rewrite per location per traversal in the metalevel copy of the* `RUSSIAN-DOLLS` *module that was just inserted.*

We start the computation with an `erewrite` *command on a configuration that consists of a portal, a user object, and a* `createInterpreter` *message:*

```
Maude> erewrite
                <>
                < me : User | depth: 0,
                  computation: ('_+_['s_^2['0.Zero], 's_^2['0.Zero]]) >
                createInterpreter(interpreterManager, me, none) .
result Configuration:
  <>
  < me : User | none >
  erewroteTerm(me, interpreter(0), 1, 's_^4['0.Zero], 'NzNat)
```

With depth 0, this results in the evaluation of the meta-representation of $2 + 2$ *directly in a meta-interpreter, with no nesting. Passing a depth of 1 results in the evaluation instead being done in a nested meta-interpreter.*

```
erewrite
                <>
                < me : User | depth: 1,
                  computation: ('_+_['s_^2['0.Zero],'s_^2['0.Zero]]) >
                createInterpreter(interpreterManager, me, none) .
result Configuration:
  <>
  < me : User | none >
  erewroteTerm(me, interpreter(0), 5,
    '__['<>.Portal,
         '<_:_|_>['me.Oid,'User.Cid,'none.AttributeSet],
         'erewroteTerm['me.Oid,'interpreter['0.Zero],'s_['0.Zero],
           '_`[_`]['''s_^4.Sort,'''0.Zero.Constant],'''NzNat.Sort]], '
               Configuration)
```

Notice here that the top level reply message `erewroteTerm(...)` *contains a result that is a meta-configuration, which contains the reply* `'erewroteTerm[...]` *meta-message from the inner meta-interpreter.*

Further Reading. As already mentioned, full details can be found in [24, 28]. The most complete treatment of reflection in both rewriting logic and membership equational logic, including proofs of correctness of the meta-representations in the reflective tower, can be found in [31]. About program transformations we only scratched the surface. Inside Maude they are generalized to *module operations* that make Maude *user-extensible* with new module composition features [46]. Outside Maude —or transforming programs in other logics to programs in Maude, or conversely— what "program transformations" are is *maps between logics* in the sense of [89] that can be implemented inside Maude when we use it as a *meta-logical framework* (see [98] and references there). Such meta-level mappings are very useful to use Maude as a *formal meta-tool* to build formal tools for many other logics (see again [98]).

9. Tools and Applications

As its title suggests, this paper has a twofold purpose. On the one hand, it tries to give a gentle introduction to Maude's *declarative programming style* without assuming prior familiarity with the language. On the other hand, it provides, for the first time, a unified account of the most recent Maude features supporting *symbolic computation* as well as other important new features. To keep the paper within reasonable size bounds, other important topics already well covered in the Maude book [28] had to be omitted or be mentioned only in passing. In particular, two important topics have not been fully explained: (i) *model checking* has only been treated in the form of search-based (with either the standard `search` command or with narrowing-based symbolic search) reachability analysis; and `Full Maude` has only made a cameo appearance through its extension into an Eqlog interpreter in Section 8.2. For more details on Full Maude, including its advanced features for object-based programming already mentioned in Section 5, we refer the reader to the detailed account in [28], and for how to build a wide range of formal tools as Full Maude extensions (as we did in this paper for Eqlog) to the quite useful methodological paper [48].

For model checking, the first important distinction to be made is between *explicit-state* model checking, where the search space of all concrete states of a system are explored, and *symbolic* model checking, where sets of states, as opposed to individual concrete states, are represented and explored symbolically. Maude supports *both* kinds of model checking by model checkers directly supported by Core Maude and by additional model checkers built on top of Maude. We first discuss Maude's support for explicit state model checking. Discussion of symbolic model checking is postponed until we discuss symbolic computation later in Section 9.1.

Explicit-State Model Checking in Maude. The most basic form of explicit-state model checking has already been illustrated in this paper, since it is supported by the `search` command. Note that, as further explained and illustrated with examples in [28], `search` can be used to both verify *invariants* or to find violations of invariants in the following sense. Suppose that an invariant has been specified as a boolean-valued predicate, say `p`, on states of sort `State`, and we wish to verify that `p` holds in every state reachable from an initial state `init`. Then we can search for a violation of `p` by giving the search command:

```
search init =>* S:State s.t. p(S:State) =/= true .
```

If the invariant `p` fails to hold, it will do so for some finite sequence of transitions from `init`, and this will be uncovered by the above `search` command since all reachable sates are explored in a breadth first manner. If, instead, the invariant `p` does hold, two things can happen: (i) if the set of states reachable from `init` is *finite*, the `search` command will report failure to find a violation of `p` and therefore `p` holds; but (ii) if there is an infinite number of states reachable from `init`, `search` will never terminate. Two options are then available: (ii)-(a) we can instead perform *bounded model checking* of `p` by specifying a depth bound for the `search` command; or (ii)-(b), as explained in [28], it may be possible to define an *equational abstraction* [106] of the given system module by identifying states by additional equations, so that the system becomes finite-state and the invariant `p` can be verified.

Under the assumption that the set of states reachable from an initial state `init` is finite, Core Maude also supports explicit-state model checking verification of any properties in linear time temporal logic (LTL) through its *LTL model checker*. We refer to [28] for a detailed account of LTL model checking in Maude, including the use of equational abstractions [106] to abstract an infinite-state system into a finite-state one that can actually be model checked for LTL properties. But this is not all. Some important system properties go beyond LTL ones. We did mention in passing properties of this kind when discussing the fault-tolerant communication protocol of Section 5, namely, that only under suitable object and message fairness assumptions could successful termination of the protocol be guaranteed. The most satisfactory way to express these advance properties and effectively model check them is by specifying them in the *linear time temporal logic of rewriting* (LTLR) [96] and verifying them using Maude's *LTLR model checker* [10], which is available in the Maude web page.

9.1. Symbolic Reasoning: Tools and Applications

This paper has placed special emphasis on Maude's novel features supporting symbolic computation, including: (i) B-unification and $E \cup B$-unification; (ii) variants and equational narrowing with the folding variant narrowing strategy; and (iii) narrowing-based symbolic reachability analysis for topmost rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup B, R)$, where: (a) $(\Sigma, E \cup B)$ is FVP, and (b) the rules R are unconditional. The best way to understand features (i)–(iii) is to see them as basic building blocks on top of which a wide range of symbolic reasoning tools and applications can be built. What follows is an attempt to provide an overview of the tools and applications that support symbolic reasoning on top of features (i)–(iii). More detailed accounts can be found in [101**?**].

Symbolic Model Checking. In complete analogy with the explicit-state case, the simplest kind of symbolic model checking supported by Maude is the narrowing-based symbolic reachability analysis provided by feature (iii) above. As in the explicit-state case, such symbolic reachability analysis can be used to verify *invariants*. The simplest (but not the only: see below) way to specify invariants is by providing a finite set $\{u_1, \ldots, u_n\}$ of constructor patterns, so that the invariant's *complement* is the set of ground instances of any of those patterns. As in the explicit-state case, *if* an invariant fails to hold, narrowing-based symbolic reachability analysis is guaranteed to detect the invariant's violation at some finite depth. If, instead, the invariant does hold two things can happen: (i) if the narrowing-based search terminates without finding a violation, the invariant holds; otherwise, several possibilities remain open: (ii)-(a) perform bounded symbolic model checking by fixing a depth bound; (ii)-(b) use *state space reduction* techniques to hopefully make the number of reachable symbolic states *finite* (for example, Maude's `fvu-narrow` command, folds less general states into more general ones for this purpose); and (ii)-(c) use an equational abstraction, where the underlying equational theory remains FVP, in conjunction with (ii)-(b) to make the space of symbolic reachable states finite. In cases (ii)-(b) and (ii)-(c) full verification of the given invariant can be achieved. An important domain-specific symbolic model checker also based on narrowing-based symbolic reachability analysis is the *Maude-NPA* tool for symbolic verification of cryptographic protocols [58]. The point is that a cryptographic protocol $\mathcal{P}$ can be specified in Maude as a topmost rewrite theory $\mathcal{P} = (\Sigma, E \cup B, R)$ whose FVP equational part $(\Sigma, E \cup B)$ specifies the algebraic properties of the protocol's cryptographic functions. As before, security violations (invariant failures) can be specified by constructor patterns $\{u_1, \ldots, u_n\}$ here called *attack states*. The strongest points of the Maude-NPA tool are perhaps that: (1) it has very advanced state space reduction techniques [59], so that a finite symbolic state space is actually reached in many cases, thus achieving full verification; and (iii) because of its support for reasoning modulo an FVP theory $(\Sigma, E \cup B)$, Maude-NPA is arguably the most general tool currently available for verifying cryptographic protocols modulo their algebraic properties.

In complete analogy with the explicit-state model checking case, the above narrowing-based symbolic model checking techniques extend to similar *narrowing-based symbolic LTL model checking* techniques [7] supported by Maude's logical LTL model checker available in the Maude web page. This symbolic technique has been further extended to *narrowing-based symbolic LTLR model checking* in [8]. Furthermore, symbolic methods can also used to define *predicate abstractions* that can effectively model check LTL properties [9].

Term Pattern Predicates. If we take to heart the above-mentioned idea of describing a possibly infinite set of states by a finite set $\{u_1, \ldots, u_n\}$ of constructor patterns, what we can arrive at is a series of increasingly more expressive languages for defining *state predicates* based on patterns. In such languages, logical operations can be effectively computed by symbolic techniques in a way completely similar to how operations on finite automata can effectively perform Boolean operations on their associated regular languages. In fact, if we have a constructor subspecification $(\Omega, E_\Omega \cup B_\Omega) \subseteq (\Sigma, E \cup B)$ such that $(\Omega, E_\Omega \cup B_\Omega)$ is FVP, then pattern conjuction can be effectively computed by variable-disjoint $E_\Omega \cup B_\Omega$-variant unification, and disjunction is just union of patterns. The good properties for the free case $E_\Omega \cup B_\Omega = \emptyset$, including also negation for order-sorted linear patterns, have been investigated in [109]. But we can go further by considering more expressive *constrained patterns* of the form $u \mid \varphi$, where u is an

Ω-term and φ is a QF Σ-formula, so that $u \mid \varphi$ specifies the ground instances of u for which φ holds. State predicates having such constrained patterns $u \mid \varphi$ as atomic predicates and closed under conjunction and disjunction in an effectively, symbolically computable manner have been studied in [125**?**]. Such a language of pattern predicates is very useful to specify sets of states both in reachability logic (see below), and in the constrained style of narrowing-based reachability analysis defined in [**?**]. Another technique where pattern predicates are extremely useful is in *rewriting modulo SMT* [115], where sets of states are represented by pattern predicates $u \mid \varphi$ where satisfiability of φ is decidable by an SMT solver. Roughly speaking, rewriting modulo SMT is a symbolic reachability analysis technique closely related to narrowing-based reachability analysis and even more so to the narrowing-based constrained reachability analysis proposed in [**?**]. The main differences with these two other approaches are: (i) instead of narrowing modulo an FVP theory $E \cup B$, we perform B-matching; (ii) the rules R can be conditional, but their conditions are SMT-solvable formulas; and (iii) after rewriting a symbolic state $u \mid \varphi$ we accumulate SMT solvable constraints coming from a rule's condition in the new symbolic state and check for their satisfiability of the new constraint.

Variant Satisfiability. As already pointed out at the end of Section 6.2, under mild conditions on the constructors of an FVP theory $(\Sigma, E \cup B)$, satisfiability of QF formulas in the initial algebra $T_{\Sigma/E \cup B}$ is *decidable* by theory-generic variant satisfiability algorithms [102, 69]. This is important, since the initial algebra $T_{\Sigma/E \cup B}$ is the initial model of the functional module specified by the theory $(\Sigma, E \cup B)$, so that satisfiability of QF formulas in many *user-defined* algebraic data types can be decided this way. For example, satisfiability of QF formulas in the initial algebras of the `NAT-FVP` and `INT-FVP` examples discussed in Section 6.2 is decidable by this method, and many more examples, including parameterized data types, are given in [102, 69]. For details on the algorithms and implementation see [124]. One tool where these satisfiability results and algorithms are routinely used is in the reachability logic theorem prover (more on this below).

Generalization, Homeomorphic Embedding, and Partial Evaluation. Generalization is the dual of unification. When B-unifying terms t and t' we look for a term u and substitution σ such that $t\sigma =_B u =_B t'\sigma$. Instead, when we want to B-*generalize* two term patterns t and t' we look for a term g and substitutions σ, τ of which they are instances up to B-equality, i.e., such that $g\sigma =_B t$ and $g\tau =_B t'$. In unification we look for most general unifiers (mgu's). Instead, in generalization we look for *least general generalizers* (lgg's). The relevance of [6] and its associated ACUOS tool as a symbolic technique is that it supports reasoning about generalization in a setting that is both order-sorted and modulo axioms B, and does so in a modular way. Specifically, the work in [6] and its Maude implementation provide a modular order-sorted equational generalization algorithm modulo B, where B can be any combination of associativity and/or commutativity and/or identity axioms.

The *homeomorphic embedding* relation $u \lhd v$, where, roughly speaking, u can be obtained from v by dropping some of v's function symbols, gives a general method for stopping any sequence of terms $t_0, t_1, \ldots, t_n, \ldots$ as soon as we can find $i < j$ such that $t_i \lhd t_j$. This important relation has been studied for untyped terms; but in the context of Maude we often need to use the homeomorphic embedding relation $u \lhd v$ when u and v are order-sorted terms and, furthermore, we need to reason not syntactically but *modulo* axioms B such as associativity and/or commutativity, that is, with a relation $u \lhd_B v$. The relation $\lhd_B$, and efficient algorithms for computing it implemented in Maude, have been studied in [**?**].

Both order-sorted generalization modulo B and homeomorphic embedding modulo B are crucial components of a *partial evaluator* for Maude functional modules. Partial evaluation of equational specifications had never been considered before in the order-sorted and modulo B level of generality needed for Maude equational programs with convergent theories of the form $(\Sigma, E \cup B)$. Partial evaluation methods that can work in this very general setting (note that the usual "vanilla flavored" case where Σ is unsorted and $B = \emptyset$ is indeed a very special subcase), have been developed in [3] and have been implemented in Maude in the Victoria tool.

Theorem Provers. Using rewriting logic's nice properties as a logical framework (see the survey [98]), the symbolic techniques currently supported by Maude can be applied to a wide range of theorem provers not just for Maude and rewriting logic but for many other logics. We will focus here on theorem proving tools more closely related to Maude. To begin with, let us discuss tools for *reachability logic*. This logic was originally proposed in [118, 117, 131, 132] as a language-generic approach to program verification parametric on the operational semantics of a programming language. Both Hoare logic and separation logic can be naturally mapped into reachability logic [118, 117]. The work in [125], extends reachability logic from a programming-language-generic logic of programs to a rewrite-theory-generic logic to reason about *both* distributed system designs and programs, based on their rewriting logic semantics. This extension is non-trivial and requires a number of new concepts and results (see [125]). In particular, concepts such as: (i) constructor pattern predicates, (ii) narrowing with conditional rules, and (iii) variant satisfiability, go a long way in making the *constructor-based* version of reachability logic proposed in [125] much more easily mechanizable by exploiting the recent symbolic features of Maude and the Maude-based variant satisfiability algorithms in [124]. Indeed, the work in [125] has been implemented in Maude. It was originally inspired by the also Maude-based work in [81], but it adds to that work a substantial number of new results and methods.

The most recent Maude-based work on reachability logic provers closest to the work in [125] is that in [80] and, even more so, in [23]. The approach in [80] adopts a semantic framework for models similar to the already-discussed work in [131, 132], i.e., state properties are specified using matching logic and assume a given first-order logic model. Therefore, the semantic framework is different from the one in [125]. An important contribution of the work in [80] is its coinductive semantics and justification for circular co-inductive reasoning. Perhaps the recent work closest to [125] in the coinductive approach is that of Ciobâcă and Lucanu in [23]. In summary, for verification of reachability properties of rewrite theories —including Hoare logic properties as a special case— the reachability logic theorem provers in [125], [80] and [23] seem to be the most advanced and most promising, and all do make use of the Maude symbolic techniques described in this paper.

Last, but not least, let us mention two other theorem proving tools. The Tamarin theorem proving tool [88] for verification of cryptographic protocols uses Maude's variant-generation algorithm, initially only for the Diffie-Hellman theory, but recently extended to finite variant theories in Maude [37]. Finally, several decision procedures for formula satisfiability modulo equational theories have been provided based on narrowing in the tool [133].

Further Reading for a Broader Perspective. This entire section can be misleading, since we have said nothing at all about many other application areas such as, for example: (i) specification and verification of programming languages based on their rewriting logic definitions; (ii) real-time and cyber-physical systems; (iii) probabilisitic systems; (iv) logical framework applications; and (v) bioinformatics applications, to mention just a few areas. There is no space here for discussing tools and applications on all those and other areas, or just for discussing many other Maude-based tools. Fortunately, the survey paper [98] gives a quite complete account of this broader perspective and, in spite of being a few years old, is still a good starting point to obtain a broad overview of the many applications made possible by Maude.

10. Conclusions and Future Work

In this paper we have both tried to give an introduction to Maude that does not assume prior acquaintance with the language, and to describe important new features that have been added to the language since 2007, when the Maude book [28] appeared. Our intention has been to provide a journal-level entry point to the language as it currently exists, both for readers new to Maude and for readers familiar with Maude who would like to have a comprehensive explanation of these important new features.

In particular, we have described those features enabling Maude's very general support for *symbolic computation*, including *order-sorted unification algorithms*: (i) modulo axioms B like associativity and/or commutativity and/or identity, and (ii) modulo equations $E \cup B$ where the equations E are convergent modulo B. Of particular importance is the existence of an infinite class of theories $E \cup B$ (namely those having the finite variant property) for which Maude's $E \cup B$-unification algorithm always terminates with a complete set of most general solutions for any unification problem. Furthermore, we have also described Maude's support for *narrowing-based symbolic reachability analysis* (that builds on the $E \cup B$-unification capability). This functionality allows the user to leverage the power of symbolic computation to carry out symbolic model-checking analyses of systems that would otherwise be unfeasible due to the need to explore infinite or very large state spaces. As we have explained in Section 9.1, these symbolic features make possible a wide range of formal tools built using them and many formal analysis applications.

We also discussed Maude's *strategy language*, which provides a declarative and modular way to carve out subsets of a system's behavior without in any way changing the rules specifying the system.

Finally, we introduced new external objects that allow Maude specifications to interact with the external world: input/output objects—the three standard IO objects and file objects (plus the prior socket objects); and a powerful new kind of external objects called *meta-interpreters*. A meta-interpreter encapsulates a Maude interpreter as an object, and can interact with other stateful objects both internal and external (including other meta-interpreters).

Regarding future work, perhaps the most important symbolic computation topic missing in the present paper is *SMT solving*. We have explained in Section 9.1 that *variant-based satisfiability* of quantifier-free formulas for algebraic data types specified by functional modules having the finite variant property and satisfying mild additional assumptions is already available in an extension of the `META-LEVEL` module and is used in reachability logic theorem proving. But there is the additional fact that in recent years experimental versions of Maude supporting access to the CVC4 [12] and Yices [52] SMT solvers have been available and have been used in various applications. The main reason for not including SMT solving in this paper is that we are still experimenting with SMT solving features and it seems preferable to leave this topic for a future publication.

Without trying to be exhaustive, three future directions seem both clear and strategically important:

1. *Symbolic Computation.* Besides further advancing Maude's support for SMT solving, important new advances are needed in narrowing-based symbolic model checking and in many theorem-proving applications.
2. *Distributed Programming.* The present, much more flexible support for interaction with external objects opens up as never before the possibility of a seamless and correct-by-construction passage from Maude specifications of concurrent object systems to their deployment as distributed systems. This can have important advantages for developing highly reliable distributed systems and for doing so in a fully declarative way.
3. *Strategies.* Now that strategies are available and efficiently supported at the Core Maude level, many applications seem ripe, including, for example, the following: (i) strategy-based model-checking algorithms; (ii) support for strategies in Maude-based theorem-proving tools; and (iii) further advances of the rewriting logic semantics project [107, 108] made possible by using strategies in the semantic definition of languages.

Acknowledgements. Durán has been partially supported by MINECO/FEDER project TIN2014-52034-R. Escobar has been partially supported by the EU

(FEDER) and the Spanish MCIU under grant TIN 2015-69175-C4-1-R, by the Spanish Generalitat Valenciana under grant PROMETEO/2019/098, and by the US Air Force Office of Scientific Research under award number FA9550-17-1-0286. Martí-Oliet and Rubio have been partially supported by MCIU Spanish project TRACES (TIN2015-67522-C3-3-R). Rubio has also been partially supported by a MCIU grant FPU17/02319. Meseguer and Talcott have been partially supported by NRL Grant N00173-17-1-G002. Talcott has also been partially supported by ONR Grant N00014-15-1-2202.

11. References

[1] Agha, G.A., Meseguer, J., Sen, K., 2006. PMaude: Rewrite-based specification language for probabilistic object systems, in: Cerone, A., Wiklicky, H. (Eds.), Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages, QAPL 2005, Edinburgh, UK, April 2-3, 2005, Elsevier. pp. 213–239.

[2] Alpuente, M., Ballis, D., Cuenca-Ortega, A., Escobar, S., Meseguer, J., 2019a. $\mathsf{ACUOS}^{\mathbf{2}}$: A high-performance system for modular ACU generalization with subtyping and inheritance, in: Calimeri, F., Leone, N., Manna, M. (Eds.), Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings, Springer. pp. 171–181.

[3] Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J., 2017. Partial evaluation of order-sorted equational programs modulo axioms, in: Hermenegildo, M.V., López-García, P. (Eds.), Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers, Springer. pp. 3–20.

[4] Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J., 2019b. Homeomorphic embedding modulo combinations of associativity and commutativity axioms, in: Mesnard, F., Stuckey, P.J. (Eds.), Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers, Springer. pp. 38–55.

[5] Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J., 2019c. Partial evaluation of order-sorted equational programs modulo axioms. Journal of Logical and Algebraic Methods in Programming (submitted for publication).

[6] Alpuente, M., Escobar, S., Espert, J., Meseguer, J., 2014. A modular order-sorted equational generalization algorithm. Information and Computation 235, 98–136.

[7] Bae, K., Escobar, S., Meseguer, J., 2013. Abstract logical model checking of infinite-state systems using narrowing, in: van Raamsdonk, F. (Ed.), 24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands, Schloss Dagstuhl - Leibniz-Zentrum fr Informatik. pp. 81–96.

[8] Bae, K., Meseguer, J., 2014a. Infinite-state model checking of LTLR formulas using narrowing, in: Escobar, S. (Ed.), Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers, Springer. pp. 113–129.

[9] Bae, K., Meseguer, J., 2014b. Predicate abstraction of rewrite theories, in: [36]. pp. 61–76.

[10] Bae, K., Meseguer, J., 2015. Model checking linear temporal logic of rewriting formulas under localized fairness. Science of Computer Programming 99, 193–234.

[11] Bae, K., Meseguer, J., Ölveczky, P.C., 2014. Formal patterns for multirate distributed real-time systems. Science of Computer Programming 91, 3–44.

[12] Barrett, C., Barbosa, H., Brain, M., Ibeling, D., King, T., Meng, P., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., Tinelli, C., 2018. CVC4 at the SMT competition 2018. CoRR abs/1806.08775.

[13] Bergstra, J., Tucker, J., 1980. Characterization of computable data types by means of a finite equational specification method, in: de Bakker, J.W., van Leeuwen, J. (Eds.), Automata, Languages and Programming, Seventh Colloquium. Springer-Verlag. volume 81 of *Lecture Notes in Computer Science*, pp. 76–90.

[14] Bobba, R., Grov, J., Gupta, I., Liu, S., Meseguer, J., Ölveczky, P., Skeirik, S., 2018. Design, formal modeling, and validation of cloud storage systems using Maude, in: Campbell, R.H., Kamhoua, C.A., Kwiat, K.A. (Eds.), Assured Cloud Computing. J. Wiley. chapter 2, pp. 10–48.

[15] Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E., Ringeissen, C., 1998. An overview of ELAN, in: [77]. pp. 55–70.

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

[17] Braga, C., Verdejo, A., 2007. Modular structural operational semantics with strategies, in: van Glabbeek, R., Mosses, P.D. (Eds.), Proceedings of the Third Workshop on Structural Operational Semantics, SOS 2006, Bonn, Germany, August 26, 2006, Elsevier. pp. 3–17.

[18] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E., 2008. Stratego/xt 0.17. A language and toolset for program transformation. Science of Computer Programming 72, 52–70.

[19] Bruni, R., Meseguer, J., 2006. Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360, 386–414.

[20] Caballero, R., López-Fraguas, F.J., 1999. A functional-logic perspective on parsing, in: Middeldorp, A., Sato, T. (Eds.), Functional and Logic Programming, 4th Fuji International Symposium, FLOPS'99, Tsukuba, Japan, November 11-13, 1999, Proceedings, Springer. pp. 85–99.

[21] Chen, S., Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.M., 2007. A systematic approach to uncover security flaws in GUI logic, in: 2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA, IEEE Computer Society. pp. 71–85.

[22] Cholewa, A., Meseguer, J., Escobar, S., 2014. Variants of Variants and the Finite Variant Property. Technical Report. CS Dept. University of Illinois at Urbana-Champaign. URL: `http://hdl.handle.net/2142/47117`.

[23] Ciobâcă, S., Lucanu, D., 2018. A coinductive approach to proving reachability properties in logically constrained term rewriting systems, in: Galmiche, D., Schulz, S., Sebastiani, R. (Eds.), Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Springer. pp. 295–311.

[24] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C., 2019. Maude Manual (Version 3.0). SRI International – University of Illinois at Urbana-Champaign. URL: `http://maude.cs.uiuc.edu`.

[25] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., 2009. Unification and narrowing in Maude 2.4, in: Treinen, R. (Ed.), Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings, Springer. pp. 380–390.

[26] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Talcott, C.L., 2015. Two decades of Maude, in: [86]. pp. 232–254.

[27] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J., 2002. Maude: specification and programming in rewriting logic. Theoretical Computer Science 285, 187–243.

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

[29] Clavel, M., Eker, S., Lincoln, P., Meseguer, J., 1996. Principles of Maude, in: Meseguer, J. (Ed.), Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996, Elsevier. pp. 65–89.

[30] Clavel, M., Meseguer, J., 2002. Reflection in conditional rewriting logic. Theoretical Computer Science 285, 245–288.

[31] Clavel, M., Meseguer, J., Palomino, M., 2007. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. Theoretical Computer Science 373, 70–91.

[32] Clavel, M., Palomino, M., 2005. The ITP Tool's Manual. Universidad Complutense de Madrid. URL: `http://maude.sip.ucm.es/itp/`.

[33] Colmerauer, A., Kanoui, H., van Caneghem, M., 1979. Étude et Réalisation d'un Système Prolog. Technical Report. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II.

[34] Comon-Lundh, H., Delaune, S., 2005. The finite variant property: How to get rid of some algebraic properties, in: Giesl, J. (Ed.), Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings, Springer. pp. 294–307.

[35] Dershowitz, N., Jouannaud, J.P., 1990. Rewrite systems, in: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science, Vol. B. North-Holland, pp. 243–320.

[36] Dowek, G. (Ed.), 2014. Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. volume 8560 of *Lecture Notes in Computer Science*, Springer.

[37] Dreier, J., Duménil, C., Kremer, S., Sasse, R., 2017. Beyond subterm-convergent equational theories in automated verification of stateful protocols, in: Maffei, M., Ryan, M. (Eds.), Principles of Security and Trust - 6th International Conference, POST 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Springer. pp. 117–140.

[38] Durán, F., 2000. The extensibility of Maude's module algebra, in: Rus, T. (Ed.), Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings, Springer. pp. 422–437.

[39] Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.L., 2016. Built-in variant generation and unification, and their applications in Maude 2.7, in: Olivetti, N., Tiwari, A. (Eds.), Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings, Springer. pp. 183–192.

[40] Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.L., 2018. Associative unification and symbolic reasoning modulo associativity in Maude, in: [121]. pp. 98–114.

[41] Durán, F., Eker, S., Escobar, S., Meseguer, J., Talcott, C.L., 2011a. Variants, unification, narrowing, and symbolic reachability in Maude 2.6, in: Schmidt-Schauß, M. (Ed.), Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia, Schloss Dagstuhl - Leibniz-Zentrum fr Informatik. pp. 31–40.

[42] Durán, F., Eker, S., Lincoln, P., Meseguer, J., 2000. Principles of Mobile Maude, in: Kotz, D., Mattern, F. (Eds.), Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zürch, Switzerland, September 13-15, 2000, Proceedings, Springer. pp. 73–85.

[43] Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X., 2008. Proving operational termination of membership equational programs. Higher-Order and Symbolic Computation 21, 59–88.

[44] Durán, F., Meseguer, J., 1998. An extensible module algebra for Maude, in: [77]. pp. 174–195.

[45] Durán, F., Meseguer, J., 2003. Structured theories and institutions. Theoretical Computer Science 309, 357–380.

[46] Durán, F., Meseguer, J., 2007. Maude's module algebra. Science of Computer Programming 66, 125–153.

[47] Durán, F., Meseguer, J., 2012. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. Journal of Algebraic and Logic Programming 81, 816–850.

[48] Durán, F., Ölveczky, P.C., 2009. A guide to extending Full Maude illustrated with the implementation of Real-Time Maude, in: [116]. pp. 83–102.

[49] Durán, F., Riesco, A., Verdejo, A., 2007. A distributed implementation of Mobile Maude, in: Denker, G., Talcott, C. (Eds.), Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006, Elsevier. pp. 113–131.

[50] Durán, F., Rocha, C., Álvarez, J.M., 2011b. Tool interoperability in the Maude Formal Environment, in: Corradini, A., Klin, B., Cîrstea, C. (Eds.), Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings, Springer. pp. 400–406.

[51] Durán, F., Rocha, C., Meseguer, J., 2019. Proving ground confluence of conditional equational specifications modulo axioms. Journal of Logical and Algebraic Methods in Programming (in this issue).

[52] Dutertre, B., 2014. Yices 2.2, in: Biere, A., Bloem, R. (Eds.), Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Springer. pp. 737–744.

[53] Eker, S., 2011. Fast sort computations for order-sorted matching and unification, in: Agha, G., Danvy, O., Meseguer, J. (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday, Springer. pp. 299–314.

[54] Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sonmez, K., 2002. Pathway logic: Symbolic analysis of biological signaling, in: Altman, R.B., Dunker, A.K., Hunter, L., Klein, T.E. (Eds.), Proceedings of the 7th Pacific Symposium on Biocomputing, PSB 2002, Lihue, Hawaii, USA, January 3-7, 2002, pp. 400–412.

[55] Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A., 2007. Deduction, strategies, and rewriting, in: Archer, M., de la Tour, T.B., Muñoz, C. (Eds.), Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006, Elsevier. pp. 3–25.

[56] Escobar, S., 2014. Functional logic programming in Maude, in: Iida, S., Meseguer, J., Ogata, K. (Eds.), Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi, Springer. pp. 315–336.

[57] Escobar, S., 2018. Multi-paradigm programming in Maude, in: [121]. pp. 26–44.

[58] Escobar, S., Meadows, C., Meseguer, J., 2009. Maude-NPA: Cryptographic protocol analysis modulo equational properties, in: Aldini, A., Barthe, G., Gorrieri, R. (Eds.), Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures, Springer. pp. 1–50.

[59] Escobar, S., Meadows, C., Meseguer, J., Santiago, S., 2014. State space reduction in the Maude-NRL protocol analyzer. Information and Computation 238, 157–186.

[60] Escobar, S., Sasse, R., Meseguer, J., 2012. Folding variant narrowing and optimal variant termination. Journal of Algebraic and Logic Programming 81, 898–928.

[61] Fioravanti, F., Gallagher, J.P. (Eds.), 2018. Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers. volume 10855 of *Lecture Notes in Computer Science*, Springer.

[62] Garavel, H., Tabikh, M., Arrada, I., 2018. Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition, in: [121]. pp. 1–25.

[63] Goguen, J., Burstall, R., 1992. Institutions: Abstract model theory for specification and programming. Journal of the ACM 39, 95–146.

[64] Goguen, J., Meseguer, J., 1984. Equality, types, modules and (why not?) generics for logic programming. Journal of Logic Programming 1, 179–210.

[65] Goguen, J., Meseguer, J., 1986. Eqlog: Equality, types, and generic modules for logic programming, in: DeGroot, D., Lindstrom, G. (Eds.), Logic Programming: Functions, Relations and Equations. Prentice-Hall, pp. 295–363.

[66] Goguen, J., Meseguer, J., 1992. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science 105, 217–273.

[67] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P., 2000. Introducing OBJ, in: Software Engineering with OBJ: Algebraic Specification in Action. Kluwer, pp. 3–167.

[68] González-Burgueño, A., Aparicio-Sánchez, D., Escobar, S., Meadows, C.A., Meseguer, J., 2018. Formal verification of the YubiKey and YubiHSM APIs in Maude-NPA, in: Barthe, G., Sutcliffe, G., Veanes, M. (Eds.), LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018, EasyChair. pp. 400–417.

[69] Gutiérrez, R., Meseguer, J., 2018. Variant-based decidable satisfiability in initial algebras with predicates, in: [61]. pp. 306–322.

[70] Gutiérrez, R., Meseguer, J., Rocha, C., 2015. Order-sorted equality enrichments modulo axioms. Science of Computer Programming 99, 235–261.

[71] Hendrix, J., Meseguer, J., 2008. Order-sorted equational unification revisited, in: Kniesel, G., Pinto, J.S. (Eds.), Proceedings of the Ninth International Workshop on Rule-Based Programming, RULE 2008, Hagenberg Castle, Austria, June 18, 2008, Elsevier. pp. 37–50.

[72] Hidalgo-Herrero, M., Verdejo, A., Ortega-Mallén, Y., 2007. Using Maude and its strategies for defining a framework for analyzing Eden semantics, in: Antoy, S. (Ed.), Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, August 11, 2006, Elsevier. pp. 119–137.

[73] Horn, A., 1951. On sentences which are true of direct unions of algebras. Journal of Symbolic Logic 16, 1421.

[74] Jouannaud, J.P., Kirchner, C., Kirchner, H., 1983. Incremental construction of unification algorithms in equational theories, in: Díaz, J. (Ed.), Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain, July 18-22, 1983, Proceedings, Springer. pp. 361–373.

[75] Katelman, M., Keller, S., Meseguer, J., 2012. Rewriting semantics of production rule sets. Journal of Logic and Algebraic Programming 81, 929–956.

[76] Katelman, M., Meseguer, J., Hou, J.C., 2008. Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking, in: Barthe, G., de Boer, F.S. (Eds.), Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings, Springer. pp. 150–169.

[77] Kirchner, C., Kirchner, H. (Eds.), 1998. Proceedings of the Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1-4, 1998. volume 15 of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[78] Kowalski, R.A., 1979. Algorithm = logic + control. Communications of the ACM 22, 424–436.

[79] Liu, S., Ölveczky, P.C., Meseguer, J., 2015. Modeling and analyzing mobile ad hoc networks in Real-Time Maude. Journal of Logical and Algebraic Methods in Programming .

[80] Lucanu, D., Rusu, V., Arusoaie, A., 2017. A generic framework for symbolic execution: A coinductive approach. Journal of Symbolic Computation 80, 125–163.

[81] Lucanu, D., Rusu, V., Arusoaie, A., Nowak, D., 2015. Verifying reachability-logic properties on rewriting-logic specifications, in: [86]. pp. 451–474.

[82] Lucas, S., Meseguer, J., 2016. Normal forms and normal theories in conditional rewriting. Journal of Logical and Algebraic Methods in Programming 85, 67–97.

[83] Martí-Oliet, N., Meseguer, J., 2002. Rewriting logic as a logical and semantic framework, in: Gabbay, D., Guenthner, F. (Eds.), Handbook of Philosophical Logic, 2nd. Edition. Kluwer Academic Publishers, pp. 1–87. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.

[84] Martí-Oliet, N., Meseguer, J., Verdejo, A., 2004. Towards a strategy language for Maude, in: Martí-Oliet, N. (Ed.), Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004, Elsevier. pp. 417–441.

[85] Martí-Oliet, N., Meseguer, J., Verdejo, A., 2009. A rewriting semantics for Maude strategies, in: [116]. pp. 227–247.

[86] Martí-Oliet, N., Ölveczky, P.C., Talcott, C.L. (Eds.), 2015. Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday. volume 9200 of *Lecture Notes in Computer Science*, Springer.

[87] Martí-Oliet, N., Palomino, M., Verdejo, A., 2007. Strategies and simulations in a semantic framework. Journal of Algorithms 62, 95–116.

[88] Meier, S., Schmidt, B., Cremers, C., Basin, D.A., 2013. The TAMARIN prover for the symbolic analysis of security protocols, in: Sharygina, N., Veith, H. (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Springer. pp. 696–701.

[89] Meseguer, J., 1989. General logics, in: Ebbinghaus, H., Fernandez-Prida, J., Garrido, M., Lascar, D., Rodríguez Artalejo, M. (Eds.), Logic Colloquium'87. Proceedings of the Colloquium held in Granada, Spain, July 20-25, 1987, North-Holland. pp. 275–329.

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

[91] Meseguer, J., 1992b. Multiparadigm logic programming, in: Kirchner, H., Levi, G. (Eds.), Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2-4, 1992, Proceedings, Springer. pp. 158–200.

[92] Meseguer, J., 1993a. A logical theory of concurrent objects and its realization in the Maude language, in: Agha, G., Wegner, P., Yonezawa, A. (Eds.), Research Directions in Concurrent Object-Oriented Programming. MIT Press, pp. 314–390.

[93] Meseguer, J., 1993b. Solving the inheritance anomaly in concurrent object-oriented programming, in: Nierstrasz, O. (Ed.), ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings, Springer. pp. 220–246.

[94] Meseguer, J., 1996. Rewriting logic as a semantic framework for concurrency: a progress report, in: Montanari, U., Sassone, V. (Eds.), CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings, Springer. pp. 331–372.

[95] Meseguer, J., 1997. Membership algebra as a logical framework for equational specification, in: Parisi-Presicce, F. (Ed.), Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3-7, 1997, Selected Papers, Springer. pp. 18–61.

[96] Meseguer, J., 2008. The temporal logic of rewriting: A gentle introduction, in: Degano, P., Nicola, R.D., Meseguer, J. (Eds.), Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday, Springer. pp. 354–382.

[97] Meseguer, J., 2009. Order-sorted parameterization and induction, in: Palsberg, J. (Ed.), Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday, Springer. pp. 43–80.

[98] Meseguer, J., 2012. Twenty years of rewriting logic. Journal of Algebraic and Logic Programming 81, 721–781.

[99] Meseguer, J., 2017. Strict coherence of conditional rewriting modulo axioms. Theoretical Computer Science 672, 1–35.

[100] Meseguer, J., 2018a. Generalized rewrite theories and coherence completion, in: [121]. pp. 164–183.

[101] Meseguer, J., 2018b. Symbolic reasoning methods in rewriting logic and Maude, in: Moss, L.S., de Queiroz, R.J.G.B., Martínez, M. (Eds.), Logic, Language, Information, and Computation - 25th International Workshop, WoLLIC 2018, Bogota, Colombia, July 24-27, 2018, Proceedings, Springer. pp. 25–60.

[102] Meseguer, J., 2018c. Variant-based satisfiability in initial algebras. Science of Computer Programming 154, 3–41.

[103] Meseguer, J., 2019. Generalized rewrite theories, coherence completion and symbolic methods. Journal of Logical and Algebraic Methods in Programming (in this issue).

[104] Meseguer, J., Goguen, J.A., 1989. Order-sorted unification. Journal of Symbolic Computation 8, 383–413.

[105] Meseguer, J., Ölveczky, P.C., 2012. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theoretical Compututer Science 451, 1–37.

[106] Meseguer, J., Palomino, M., Martí-Oliet, N., 2008. Equational abstractions. Theoretical Computer Science 403, 239–264.

[107] Meseguer, J., Roşu, G., 2007. The rewriting logic semantics project. Theoretical Computer Science 373, 213–237.

[108] Meseguer, J., Roşu, G., 2013. The rewriting logic semantics project: A progress report. Information and Computation 231, 38–69.

[109] Meseguer, J., Skeirik, S., 2017. Equational formulas and pattern operations in initial order-sorted algebras. Formal Aspects of Computing 29, 423–452.

[110] Meseguer, J., Talcott, C., 2002. Semantic models for distributed object reflection, in: Magnusson, B. (Ed.), ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Málaga, Spain, June 10-14, 2002, Proceedings, Springer. pp. 1–36.

[111] Meseguer, J., Thati, P., 2007. Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. Higher-Order and Symbolic Computation 20, 123–160.

[112] Olarte, C., Pimentel, E., Rocha, C., 2018. Proving structural properties of sequent systems in rewriting logic, in: [121]. pp. 115–135.

[113] Ölveczky, P.C., Meseguer, J., 2007. Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20, 161–196.

[114] Ölveczky, P.C., Thorvaldsen, S., 2009. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410, 254–280.

[115] Rocha, C., Meseguer, J., Muñoz, C.A., 2017. Rewriting modulo SMT and open system analysis. Journal of Logic and Algebraic Methods in Programming 86, 269–297.

[116] Roșu, G. (Ed.), 2009. Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008. volume 238(3) of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[117] Roșu, G., Ștefănescu, A., 2012a. Checking reachability using matching logic, in: Leavens, G.T., Dwyer, M.B. (Eds.), Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012, ACM. pp. 555–574.

[118] Roșu, G., Ștefănescu, A., 2012b. From Hoare logic to matching logic reachability, in: Giannakopoulou, D., Méry, D. (Eds.), FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings, Springer. pp. 387–402.

[119] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2019a. Model checking strategy-controlled rewriting systems, in: Geuvers, H. (Ed.), 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 34:1–34:18.

[120] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2019b. Parameterized strategies specification in Maude, in: Fiadeiro, J., Țuțu, I. (Eds.), Recent Trends in Algebraic Development Techniques, 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 25, 2018, Revised Selected Papers, Springer. pp. 27–44.

[121] Rusu, V. (Ed.), 2018. Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, Proceedings. volume 11152 of *Lecture Notes in Computer Science*, Springer.

[122] Sasse, R., King, S.T., Meseguer, J., Tang, S., 2013. IBOS: A correct-by-construction modular browser, in: Pasareanu, C.S., Salaün, G. (Eds.), Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers, Springer. pp. 224–241.

[123] Serbanuta, T., Roşu, G., Meseguer, J., 2009. A rewriting logic approach to operational semantics. Information and Computation 207, 305–340.

[124] Skeirik, S., Meseguer, J., 2018. Metalevel algorithms for variant satisfiability. Journal of Logical and Algebraic Methods in Programming 96, 81–110.

[125] Skeirik, S., Ştefănescu, A., Meseguer, J., 2018. A constructor-based reachability logic for rewrite theories, in: [61]. pp. 201–217.

[126] Stehr, M., 2000. CINNI - a generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi, in: Futatsugi, K. (Ed.), Proceedings of the Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18-20, 2000, Elsevier. pp. 70–92.

[127] Stehr, M., Meseguer, J., 2004. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework, in: Owe, O., Krogdahl, S., Lyche, T. (Eds.), From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl, Springer. pp. 334–375.

[128] Stehr, M.O., Meseguer, J., Ölveczky, P.C., 2001. Rewriting logic as a unifying framework for Petri nets, in: Ehrig, H., Juhás, G., Padberg, J., Rozenberg, G. (Eds.), Unifying Petri Nets, Advances in Petri Nets, Springer. pp. 250–303.

[129] Strachey, C., 2000. Fundamental concepts in programming languages. Higher-Order and Symbolic Computation 13, 11–49.

[130] Talcott, C., Eker, S., Knapp, M., Lincoln, P., Laderoute, K., 2004. Pathway logic modeling of protein functional domains in signal transduction, in: Altman, R.B., Dunker, A.K., Hunter, L., Jung, T.A., Klein, T.E. (Eds.), Biocomputing 2004, Proceedings of the Pacific Symposium, Hawaii, USA, 6-10 January 2004, World Scientific. pp. 568–580.

[131] Ştefănescu, A., Ciobâcă, S., Mereuta, R., Moore, B.M., Serbanuta, T., Roşu, G., 2014. All-path reachability logic, in: [36]. pp. 425–440.

[132] Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G., 2016. Semantics-based program verifiers for all languages, in: Visser, E., Smaragdakis, Y. (Eds.), Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016, ACM. pp. 74–91.

[133] Tushkanova, E., Giorgetti, A., Ringeissen, C., Kouchnarenko, O., 2015. A rule-based system for automatic decidability and combinability. Science of Computer Programming 99, 3–23.

[134] Verdejo, A., Martí-Oliet, N., 2000. Implementing CCS in Maude, in: Bolognesi, T., Latella, D. (Eds.), Formal Techniques for Distributed System Development, FORTE/PSTV 2000, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX), October 10-13, 2000, Pisa, Italy, Proceedings, Kluwer. pp. 351–366.