Unification and Narrowing in Maude 2.4*

Manuel Clavel[1,2], Francisco Durán[3], Steven Eker[4], Santiago Escobar[5],
Patrick Lincoln[4], Narciso Martí-Oliet[2], José Meseguer[6], and Carolyn Talcott[4]

[1] IMDEA Software, Madrid, Spain
[2] Universidad Complutense de Madrid, Spain
[3] Universidad de Málaga, Spain
[4] SRI International, CA, USA
[5] Universidad Politécnica de Valencia, Spain
[6] University of Illinois at Urbana-Champaign, IL, USA

Abstract. Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications, and has a relatively large worldwide user and open-source developer base. This paper introduces novel features of Maude 2.4 including support for unification and narrowing. Unification is supported in Core Maude, the core rewriting engine of Maude, with commands and metalevel functions for order-sorted unification modulo some frequently occurring equational axioms. Narrowing is currently supported in its Full Maude extension. We also give a brief summary of the most important features of Maude 2.4 that were not part of Maude 2.0 and earlier releases. These features include communication with external objects, a new implementation of its module algebra, and new predefined libraries. We also review some new Maude applications.

1 Introduction

Maude is a language and a system based on rewriting logic [7]. Maude modules are rewrite theories, while computation with such modules corresponds to efficient deduction by rewriting. Because of its logical basis and its initial model semantics, a Maude module defines a precise mathematical model. This means that Maude and its formal tool environment can be used in three, mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification system.

The first version of Maude was publicly released at the beginning of 1999 and presented at RTA'99 [5]; four years later, Maude 2.0 was introduced at RTA'03 [6]. The new and improved features since Maude 2.0 include: built-in AC unification; narrowing;

* M. Clavel has been partially supported by MICINN grants TIN2005-09207-C03-03 and TIN2006-15660-C02-01, and by CAM program S-0505/TIC/0407. F. Durán has been partially supported by MICINN grant TIN2008-03107 and Junta de Andalucía P06-TIC2250 and P07-TIC3184. S. Escobar has been partially supported by MICINN grant TIN2007-68093-C02-02, Integrated Action HA 2006-0007, and Generalitat Valenciana GVPRE/2008/113. P. Lincoln's effort partially supported by NSF grant CNS-0749931. N. Martí-Oliet has been partially supported by MICINN grant TIN2006-15660-C02-01 and CAM program S-0505/TIC/0407.

object-message fairness and communication with external objects; a new implementation at the core level of its module algebra; new predefined libraries of parameterized data types; and a linear Diophantine equation solver.

Unification is built-in in Core Maude 2.4. Currently, narrowing is available in *Full Maude* [12, 7], an extension of Maude written in Maude itself by taking advantage of its reflective capabilities. It has been used as a testbed for prototyping new features: parameterization [12], strategies [20], unification [7], and so on; and as a key component to build various formal tools by reflection (see Section 5).

There are several functional-logic programming languages based on narrowing (see, e.g., `http://www.informatik.uni-kiel.de/~mh/FLP/implementations.html`). However, we are not aware of any other programming language supporting AC-narrowing, or combining narrowing with all the other features that Maude provides.

The releases of Maude since Maude 2.0 have added many other new features and improvements that cannot be described here. We refer the reader to the Maude documentation [8] for more details. The LNCS book on Maude [7] contains many additional examples and explanations, as well as information on applications and tools. However, the book only covers up to Maude 2.3, and therefore does not cover features like AC unification and narrowing. The Maude system, its documentation, and related papers and applications are available from its website at `http://maude.cs.uiuc.edu`.

2 Unification

Unification is a fundamental deductive mechanism used in many automated deduction tasks. It is also very important in combining the paradigms of functional programming and logic programming. Furthermore, in the context of Maude, unification can be very useful to reason not only about equational theories, but also about rewrite theories. In this section, we explain how order-sorted unification modulo frequently occurring equational axioms is currently supported in Maude 2.4.

Although the most general equational theories supported by Maude are membership equational theories, to obtain practical unification algorithms, allowing us to effectively compute the solutions of an equational unification problem, it is important to restrict ourselves to *order-sorted* equational theories. Furthermore, for an arbitrary set of equations no unification algorithm may be known; even if one is known, the number of solutions may be *infinite*. This suggests a hybrid approach, in which we take advantage of Maude's structuring of a module's equations into equational axioms Ax, such as associativity, and/or commutativity, and/or identity, and equations E, which are assumed to be confluent and terminating modulo Ax. We can then consider order-sorted equational theories of the form $(\Sigma, E \cup Ax)$ and decompose the $E \cup Ax$-unification problem into two problems: one of Ax-unification, and another of $E \cup Ax$-unification that uses an Ax-unification algorithm as a subroutine. The point is that *only* Ax-unification needs to be built-in at the level of Core Maude's C++ implementation for efficiency purposes. Instead, $E \cup Ax$-unification can then be implemented in Maude itself. Since the axioms Ax are well-known and unification algorithms exist for them, the task of building in efficient Ax-unification algorithms, although difficult, becomes manageable.

Unlike unsorted syntactic unification, which always either fails or has a single most general unifier, order-sorted syntactic unification is *not* unitary, that is, there is in general no single most general unifier. What exists (if Σ is finite) is a finite minimal complete set of syntactic unifiers. For some commonly occurring theories having a unification algorithm, such as associativity of a binary function symbol, it is well-known that unification is not finitary. However, for other theories, such as commutativity (C) or associativity-commutativity (AC), unification is finitary, both when Σ is unsorted and order-sorted (and finite). Maude 2.4 provides an order-sorted Ax-unification algorithm for all order-sorted theories $(\Sigma, E \cup Ax)$ such that:

- the signature Σ is *preregular* modulo Ax [7];
- the axioms Ax associated to function symbols are as follows:
  - there can be arbitrary function symbols and constants with no attributes;
  - the `iter` equational attribute can be declared for some unary symbols;
  - the `comm` or `assoc comm` attributes can be declared for some binary function symbols, but no other equational attributes can be given for such symbols.

Explicitly excluded are theories with binary function symbols having either: (i) the `id:`, `left id:`, or `right id:` attributes; or (ii) the `assoc` attribute without the `comm` one; or (iii) a combination of (i) and (ii). The reason for excluding the `assoc` attribute without `comm` is the already-mentioned fact that associative unification is not finitary. The reason for excluding for the moment the `id:`, `left id:`, and `right id:` attributes is that they are *collapse equations* (one of the terms in the equation is a variable), requiring a more complex way of combining their unification algorithms. However, Ax-unification, where Ax includes such `id:`, `left id:`, and `right id:` attributes, is currently supported in Full Maude by narrowing (see Section 3).

If we give to Maude a unification problem in a functional module `fmod` $(\Sigma, E \cup Ax)$ `endfm`, then the equations E are ignored and we get a complete set of order-sorted unifiers modulo the theory (Σ, Ax). To deal with $E \cup Ax$-unification, other methods, that use the Ax-unification algorithm as a component, can later be defined (see Section 5).

Maude provides a unification command of the form:

`unify [`n`] in` *ModId* `:` t_1 `=?` t'_1 `/\ ... /\` t_k `=?` t'_k `.`

where $k \geq 1$, n is an optional argument providing a bound on the number of unifiers, and *ModId* is the name of the module or theory in which the unification takes place.

The use of a bound on the number of unifiers, as well as the use of the AC operator _+_ in the predefined `NAT` module, plus the fact that even small AC-unification problems can generate a large number of unifiers are all illustrated by the following command:

```
Maude> unify [10] in NAT : X:Nat + X:Nat + Y:Nat =? A:Nat + B:Nat .
Solution 1
X:Nat --> #1:Nat + #2:Nat + #4:Nat
Y:Nat --> #3:Nat + #5:Nat
A:Nat --> #1:Nat + #1:Nat + #2:Nat + #3:Nat
B:Nat --> #2:Nat + #4:Nat + #4:Nat + #5:Nat
...
Solution 10
X:Nat --> #1:Nat + #2:Nat
Y:Nat --> #3:Nat
```

```
A:Nat --> #1:Nat + #1:Nat
B:Nat --> #2:Nat + #2:Nat + #3:Nat
```

Notice that in each assignment X `-->` t in a unifier, the variables appearing in the term t are always *fresh* variables of the form `#n:Sort`. Assuming that no bound on the number of unifiers is specified by the user, Maude will compute a *complete* set of order-sorted unifiers modulo *Ax*, for *Ax* a set of supported equational axioms. However, there is no guarantee that the computed set of unifiers is *minimal*, that is, some of the unifiers in the computed set may be redundant, since they could be obtained as instances (modulo *Ax*) of other unifiers in the set.

Order-sorted unification is NP-complete in general because Boolean algebra can be encoded as an order-sorted free theory signature and hence satisfiability can be reduced to an order-sorted free theory unification problem. In practice, reasonable performance can be obtained using a Binary Decision Diagram technique to compute sorts for free variables occurring in unsorted unifiers. Furthermore in the AC case, sort information can be pushed into the unsorted unification algorithm and used to prune the Diophantine basis and the choice of subsets drawn from such a basis [13].

The unification theory combination framework and AC unification algorithm are based on [4] while the Diophantine system solver used by the AC algorithm is based on [10]. The unification algorithm has been thoroughly tested (by S. Escobar and R. Sasse) using CiME [11] as an oracle, and has shown better average performance than CiME on the same problems.

Much of Maude's functionality is supported in its metalevel, so that it becomes available by reflection [7]. Unification is reflected in by the following descent functions:

```
op metaUnify : Module UnificationProblem Nat Nat ~> UnificationPair? .
op metaDisjointUnify :
     Module UnificationProblem Nat Nat ~> UnificationTriple? .
```

The key difference between `metaUnify` and `metaDisjointUnify` is that the latter assumes that the variables in the left- and right-hand unificands are to be considered *disjoint* even when they are not so, and it generates each solution to the given unification problem not as a single substitution, but as a *pair* of substitutions, one for left unificands and the other for right unificands. This functionality is very useful for applications, such as critical-pair checking or narrowing (see Section 3), where a disjoint copy of the terms or rules involved must always be computed before unification is performed.

Since it is convenient to reuse variable names from unifiers in new problems, for example in narrowing, this is allowed via the third argument, which is the largest number n appearing in a unificand metavariable of the form `#n:Sort`. Then the fresh metavariables in the computed unifiers will all be numbered from $n+1$ on.

Results are returned using the following constructors:

```
subsort UnificationPair < UnificationPair? .
subsort UnificationTriple < UnificationTriple? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Nat -> UnificationTriple [ctor] .
```

The `Nat` component is the largest n occurring in a fresh `#`n`:Sort` metavariable. In this way, the next invocation of the function can use this parameter to make sure that the new variables generated are always fresh.

Examples illustrating the use of these metalevel functions can be found in [8].

3 Narrowing

Narrowing generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification instead of matching in order to (non–deterministically) reduce a term.

At each narrowing step, one must choose which subterm of the subject term, which rule of the specification, and which instantiation on the variables of the subject term and the rule's lefthand side is going to be considered. Given an order-sorted rewrite theory (Σ, Ax, R) where R is a set of unconditional rewrite rules such that the lefthand sides are non-variable terms and the rules are explicitly Ax-coherent [22], and Ax is a set of axioms such that a finitary Ax-unification procedure is available in Maude, the *R,Ax-narrowing* relation is defined as $t \rightsquigarrow_{\sigma,p,R,Ax} t'$ iff there is a non-variable position p of t, a (possibly renamed) rule $l \to r$ in R, and a unifier $\sigma \in \mathit{Unif}_{Ax}(t|_p, l)$ such that $t' = \sigma(t[r]_p)$. Full Maude supports a version of narrowing *with simplification*. That is, given an order-sorted rewrite theory $(\Sigma, Ax \cup E, R)$ where R and Ax are defined as above and E are the remaining equations, the combined relation $(\rightsquigarrow_{\sigma,p,R,Ax};\rightarrow^{!}_{E,Ax})$ is defined as $t \rightsquigarrow_{\sigma,p,R,Ax};\rightarrow^{!}_{E,Ax} t''$ iff $t \rightsquigarrow_{\sigma,p,R,Ax} t'$, $t' \rightarrow^{*}_{E,Ax} t''$, and t'' is E,Ax-irreducible. Note that this combined relation may be incomplete, i.e., given a reachability problem of the form $t \rightarrow^{*} t'$ and a solution σ (i.e., $\sigma(t) \rightarrow^{*}_{R,E\cup Ax} \sigma(t')$), the relation $\rightsquigarrow_{\sigma,p,R,Ax};\rightarrow^{!}_{E,Ax}$ may not be able to find a more general solution. The reason is that the equations E should also be executed by narrowing instead of rewriting to ensure completeness under appropriate conditions (see [22] and Section 5).

The user can enter in Full Maude a search command of the form:

```
search [n,m] in ModId : t1 SearchArrow t2 .
```

where: n and m are optional arguments providing, respectively, a bound on the number of desired solutions and the maximum depth of the search; *ModId* is the module where the search takes place; t_1 is the starting *non-variable term*, which may contain variables; t_2 is the term specifying the pattern that has to be reached, with variables possibly shared with t_1; *SearchArrow* is an arrow indicating the form of the narrowing proof from t_1 until t_2 (`~>1` for a narrowing proof consisting of exactly one step; `~>+` for a proof of one or more steps; `~>*` for a proof of none, one, or more steps; and `~>!` to indicate that only *strongly irreducible* final states are allowed, i.e., states that cannot be further narrowed).

Consider, for example, the following Petri-net-like specification of a vending machine to buy apples (`a`) or cakes (`c`) with dollars (`$`) and/or quaters (`q`):

```
(mod VENDING-MACHINE is
   sorts Coin Item Marking Money State .
   subsort Coin < Money .
   op __ : Money Money -> Money [assoc comm] .
   subsort Money Item < Marking .
   op __ : Marking Marking -> Marking [assoc comm] .
   op <_> : Marking -> State .
   ops $ q : -> Coin [format (r! o)] .
   ops a c : -> Item [format (b! o)] .
```

```
    var M : Marking .
    rl [buy-c] : < $ > => < c > .
    rl [buy-c] : < M $ > => < M c > .
    rl [buy-a] : < $ > => < a q > .
    rl [buy-a] : < M $ > => < M a q > .
    rl [change]: < q q q q > => < $ > .
    rl [change]: < M q q q q > => < M $ > .
  endm)
```

We can use the narrowing search command to answer the question: *Is there any combination of one or more coins that returns exactly an apple and a cake?* This is done by searching for states that have a variable of sort `Money` instead of sort `Marking` at the starting term and match a corresponding pattern at the end.

```
Maude> (search [,4] in VENDING-MACHINE : < M:Money > ~>* < a c > .)
Solution 1
M:Money --> $ q q q
Solution 2
M:Money --> q q q q q q q
```

Note that we must restrict the depth, because narrowing does not terminate for this reachability problem even though the above two solutions are indeed the *only* solutions.

Narrowing-based reachability analysis is also available at the metalevel by using the following `metaNarrowSearch` function.

```
op metaNarrowSearch :
     Module Term Term Substitution Qid Bound Bound -> ResultTripleSet .
```

If a non-identity substitution is provided in the fourth argument, then any computed substitution must be an instance of the provided one, i.e., we can restrict the computed narrowing sequences to some concrete shape. The `Qid` metarepresents the appropriate search arrow, similar to the `metaSearch` command (see [8, Section 11.4.6]). For the bounds, the first one is the number of computed solutions, and the second one is the maximum length of the narrowing sequences, i.e., the depth of the narrowing tree.

Unification modulo identity: The `id-unify` command. As described in Section 2, Maude 2.4 provides an order-sorted *Ax*-unification algorithm for all order-sorted theories $(\Sigma, E \cup Ax)$ such that Σ is preregular and *Ax* can include any combination of equational axioms for a function symbol *except* the `id:`, `left id:`, `right id:`, and `assoc` without `comm`. If a theory (Σ, Ax) contains the `id:`, `left id:`, or `right id:` attributes in *Ax* (but *not* `assoc` without `comm`), we can perform unification modulo *Ax* as follows:

1. we decompose *Ax* into a disjoint union $Ax = \widetilde{Ax} \cup Ids$, where $\widetilde{Ax}$ does not contain any `id:`, `left id:`, or `right id:` attribute, and *Ids* is the set of such extra attributes;
2. we define the rewrite theory $(\Sigma, \widetilde{Ax}, \overrightarrow{Ids})$ where $\overrightarrow{Ids}$ contains the obvious rules for each of the equational identity attributes, that is:
   - if f has an `id:` attribute in *Ax* and $[s]$ is the top sort of f (which we can identify with the kind), we add rules $f(x,e) \to x$ and $f(e,x) \to x$ into $\overrightarrow{Ids}$, where x is a variable of sort $[s]$ and e is the identity symbol (if f has also the `comm` attribute, only one such rule is needed);

- Likewise, for f with the `left id:` (resp. `right id:`) attribute, we add the rule $f(e,x) \to x$ (resp. $f(x,e) \to x$) into $\overrightarrow{Ids}$.

3. based on the idea of “variants” in [9], for the Ax-unification problem $t \overset{?}{=} t'$, we compute the variants of t and the variants of t' using narrowing modulo $\widetilde{Ax}$ within the theory $(\Sigma, \widetilde{Ax}, \overrightarrow{Ids})$ and perform $\widetilde{Ax}$-unification pairwise among all the variants of t and t' (see [17] for details).

The Full Maude `id-unify` command implements the above Ax-unification procedure (with $Ax = \widetilde{Ax} \cup Ids$) using variants.[7] Given a module or theory *ModId* having a set Ax of equational axioms for the signature Σ such that Σ does not include symbols with `assoc` without `comm` attributes in Ax, Full Maude provides a unification command for Ax-unification of the form:

```
id-unify in ModId : t =? t' .
```

where only one unification problem is admitted, in contrast to the `unify` command, and such that no limit to the number of unifiers can be specified.

The procedure for equational Ax-unification, where Ax can contain any Maude equational attribute except `assoc` without `comm`, is also available at the metalevel thanks to the `metaACUUnify` function.

```
op metaACUUnify : Module Term Term -> SubstitutionSet .
```

4 Other Available Features

In this section we briefly mention some of the other features introduced in Maude since Maude 2.0. More details can be found in the Maude documentation [8, 7].

Object-message fairness and external objects. Distributed systems can be modeled as multisets of entities, coupled by some suitable communication mechanism. In object-based distributed systems, the entities are objects, each with a unique identity, and the communication mechanism is message passing. Maude 2.4 supports the modeling of such systems by providing a predefined `CONFIGURATION` module and an *object-message fair* rewriting strategy that is well suited for executing object system configurations. It also supports *external objects*, so that objects inside a Maude configuration can interact with different kinds of objects outside it. The external objects directly supported are internet *sockets*, but through them it is possible to interact with other external objects.

Module algebra. As in other languages in the Clear/OBJ tradition, the abstract syntax of Maude specifications can be seen as given by *module expressions*, defining a new module out of previously defined modules by combining or modifying them according to a specific set of operations. Maude 2.4 supports module operations for summation, renaming, and instantiation of parameterized modules. Theories, parameterized modules, and views are the basic building blocks of *parameterized* programming.

[7] Of course, this is less efficient than built-in Ax-unification. However, a number of useful applications can be supported in practice even with `id-unify` (see Section 5).

New predefined libraries. Maude has a standard library of predefined modules. In addition to predefined modules providing commonly used data types, such as Booleans, numbers, strings, and quoted identifiers, that were already available in Maude 2.0, the following modules are predefined. The `RANDOM` module provides a pseudo-random number generator, and the system module `COUNTER` a "counter" that can be used to generate new names. These modules can be used together, e.g., to specify *probabilistic models* in Maude [7]. For certain applications, it is convenient to have a predefined specification for machine integers instead of the arbitrary size integers provided by the `INT` module. The parameterized module `MACHINE-INT` takes a bit-width parameter $n \geq 2$, which must be a power of 2, and defines machine integer operations. For parameterized programming, several functional theories like `TRIV`, `DEFAULT`, `STRICT-WEAK-ORDER`, `TOTAL-PREORDER`, and `TOTAL-ORDER` are predefined. Also predefined are the modules `LIST`, `SET`, `LIST*`, and `SET*`. The `WEAKLY-SORTABLE-LIST` module, parameterized by `STRICT-WEAK-ORDER`, specifies a *stable* version of the *mergesort* algorithm, and the `SORTABLE-LIST` module sorts lists with respect to the `TOTAL-ORDER` theory.

5 Some Applications

In this section we review briefly some Maude applications that have been or can be developed, particularly with the new unification and narrowing infrastructure.

Narrowing-based unification. If we have a dedicated algorithm to solve unification problems in an order-sorted theory (Σ, Ax), then we can use it as a component to obtain a unification algorithm for theories of the form $(\Sigma, E \cup Ax)$, provided the equations E are coherent, confluent and terminating modulo Ax [18]. We just need to add to $(\Sigma, E \cup Ax)$ a new constant tt, a binary symbol eq, and equations of the form $eq(x,x) = tt$ (one for each top sort in Σ, with x of that top sort). Then we can reduce an $E \cup Ax$-unification problem $t \overset{?}{=} t'$ to the narrowing reachability problem $eq(t,t') \rightsquigarrow^{*} tt$ modulo Ax in the theory extending $(\Sigma, E \cup Ax)$ with these new operators, and equations.

The computation of $E \cup Ax$-unifiers by narrowing modulo Ax yields a complete but in general infinite set of $E \cup Ax$-unifiers. When $Ax = \emptyset$, sufficient conditions are known ensuring termination of the basic narrowing strategy (see, e.g., [19, 1]), and therefore yielding a finite complete set of $E \cup Ax$-unifiers. However, for axioms Ax such as AC, it is well-known that narrowing modulo AC "almost never terminates" and, furthermore, that basic narrowing is incomplete [25, 9]. Based on the idea of "variants" in [9], a complete narrowing strategy modulo Ax called *variant narrowing* has been proposed in [17]. Furthermore, in [16] sufficient checkable conditions on $(\Sigma, E \cup Ax)$ have been given ensuring that the $E \cup Ax$-unification algorithm provided by variant narrowing modulo Ax is finitary, even though variant narrowing modulo Ax may still not terminate in spite of such conditions. A Maude-based narrowing library that uses the current built-in unification algorithm as a component has been developed by S. Escobar.

Symbolic reachability analysis in rewrite theories. A rewrite theory, say $\mathcal{R} = (\Sigma, E \cup Ax, R)$, specified in Maude as a system module, describes a concurrent system whose states are $E \cup Ax$-equivalence classes of ground terms, and whose local concurrent transitions are specified by the rules R. When formally analyzing the properties of $\mathcal{R}$, an important problem is ascertaining for specific patterns t and t' the *symbolic reachability problem* $(\exists X)\; t \longrightarrow^{*} t'$ with X the set of variables appearing in t and t'. As shown in [22], provided the rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$ 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 Ax, narrowing with the rules R modulo the equations $E \cup Ax$ gives a constructive, *sound, and complete* method to solve reachability problems of the form $(\exists X)\; t \longrightarrow^{*} t'$.

Of course, narrowing with R modulo $E \cup Ax$ requires performing $E \cup Ax$-unification at each narrowing step, which as explained above can itself be performed by narrowing with the equations E modulo Ax, provided E is coherent, confluent, and terminating modulo Ax. Therefore, in performing symbolic reachability analysis in a rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$ there are usually *two* levels of narrowing and *two* levels of unification: narrowing with R modulo $E \cup Ax$ for reachability, and narrowing with E modulo Ax for unification modulo $E \cup Ax$. This is exactly the approach taken in the Maude-NPA protocol analyzer [14], where cryptographic protocols are formally specified as rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$, and the formal reachability analysis is performed in a *backwards* way, from an attack state to an initial state. This just means that we perform standard (forwards) reachability analysis with the rewrite theory $\mathcal{R}^{-1} = (\Sigma, E \cup Ax, R^{-1})$, where $R^{-1} = \{r \longrightarrow l \mid (l \longrightarrow r) \in R\}$. The equational theory $E \cup Ax$ typically specifies the algebraic properties of the cryptographic functions used in the given protocol, for example, public key encryption and decryption, exclusive or, modular exponentiation, and so on, which often have the finite variant property [9].

Solving a symbolic reachability problem $(\exists X)\; t \longrightarrow^{*} t'$ corresponds to *falsifying* an *invariant*, namely, that all states reachable from t are in the complement of the instances of t'. The paper [15] shows how narrowing can be used to perform a more general *symbolic model checking*, not just for invariants, but for temporal logic formulas.

Building Formal Tools Reflectively in Maude. Another important application area is the development of formal tools by reflection. This can be done directly in Core Maude using the `META-LEVEL` module, or in Full Maude as a language extension.The Maude book [7] describes many such tools: the Maude inductive theorem prover, Church-Rosser checker, coherence checker, sufficient completeness checker, termination tool, real-time Maude tool, and several others. Some recent new tools are the Maudeling and MOMENT-2 tools, for formal specification and analysis in model-based software engineering [23, 3], the already mentioned Maude-NPA [14], and a model checker for the linear temporal logic of rewriting [2]. With metalevel support for unification and narrowing, new formal tools can be built in the near future.

The Rewriting Logic Semantics Project. An important and very active area of Maude applications is based on the idea of giving formal semantics to a programming language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, where $\Sigma_{\mathcal{L}}$ defines the syntax and the semantic types of $\mathcal{L}$, $E_{\mathcal{L}}$ the deterministic semantics, and $R_{\mathcal{L}}$ the concurrent semantics (see [21, 24]). Specifying $\mathcal{R}_{\mathcal{L}}$ in Maude as a system module yields not only an $\mathcal{L}$-interpreter, but also an $\mathcal{L}$-model checker that can perform sophisticated program analysis.

References

1. M. Alpuente, S. Escobar, and J. Iborra. Modular termination of basic narrowing. In *Procs. of RTA'08*, LNCS 5117:1–16, 2008.
2. K. Bae and J. Meseguer. A rewriting-based model checker for the linear temporal logic of rewriting. In *Procs. of RULE'08*, ENTCS, to appear.
3. A. Boronat and J. Meseguer. An Algebraic semantics for MOF. In *Procs. of FASE'08*, LNCS 4961:377-391, 2008.
4. A. Boudet, E. Contejean, and H. Devie. A new AC unification algorithm with an algorithm for solving systems of diophantine equations. In *Procs. of LICS'90*, pp. 289–299.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. The Maude system. In *Procs. of RTA'99*, LNCS 1631:240–243, 1999.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Talcott. The Maude 2.0 system. In *Procs. of RTA'03*, LNCS 2706:14–29, 2003.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All about Maude, A high-performance logical framework*, LNCS 4350, 2007.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *Maude Manual (v. 2.4)*, SRI Intl. & U. of Illinois at Urbana-Champaign, Oct. 2008. Available at `http://maude.cs.uiuc.edu`.
9. H. Comon-Lundh and S. Delaune. The finite variant property: how to get rid of some algebraic properties. In *Procs. of RTA'05*, LNCS 3467:294–307, 2005.
10. E. Contejean and H. Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.
11. E. Contejean, C. Marché, and X. Urbain. *CiME 3*. Available `http://cime.lri.fr/`. 2004.
12. F. Durán and J. Meseguer. Maude's module algebra. *Sci. Comp. Progr.* 66(2):125–153, 2007.
13. S. Eker. Unification in Maude. Talk at the "Protocol eXchange Seminar", Naval Postgraduate School, January 2007. Available at `http://maude.cs.uiuc.edu/papers/`.
14. S. Escobar, C. Meadows, J. Meseguer. A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. *Theor. Comput. Sci.* 367(1-2):162–202, 2006.
15. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Procs. of RTA'07*, LNCS 4533:153–168, 2007.
16. S. Escobar, J. Meseguer, and R. Sasse. Effectively checking the finite variant property. In *Procs. of RTA'08*, LNCS 5117:79–93, 2008.
17. S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. In *Procs. of WRLA'08*, ENTCS:88–102, 2008.
18. J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Procs. of ICALP'83*, LNCS 154:361–373, 1983.
19. J.-M. Hullot. Canonical forms and unification. *Procs. CADE'80*, LNCS 87:318–334, 1980.
20. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In *Procs. of WRLA'04*, ENTCS 117:417–441, 2005.
21. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theor. Comput. Sci.* 373(3): 213-237, 2007.
22. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *High.-Ord. Symb. Comp.* 20(1-2):123–160, 2007.
23. J. E. Rivera and A. Vallecillo. Adding behavioral semantics to models. In *Procs. of EDOC'07*, pp. 169-180, 2007.
24. T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*. In Press. Available online 6 December 2008.
25. E. Viola. E-unifiability via narrowing. In *Procs. of ICTCS'01*, LNCS 2202:426–438, 2001.