# What's New in Maude 2.4

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

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

**Abstract.** This paper describes the main features introduced in Maude since Maude 2.0, which include: communication with external objects; a new implementation of its module algebra with operations for summation and renaming of modules, as well as support for parameterized programming by means of theories and views; new predefined libraries of parameterized data types, supporting efficient versions of lists, sets, maps, and arrays; a linear Diophantine equation solver; a strategy language; and support for unification.

## 1 Introduction

Maude is a language and a system based on rewriting logic [4]. 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 Maude system, its documentation [7], and related papers and applications are available from the Maude website `http://maude.cs.uiuc.edu`.

The first version of Maude was publicly released at the beginning of 1999 and presented at RTA'99 [3]; four years later, Maude 2.0 was introduced at RTA'03 [5]. The new and improved features of Maude 2.0 included: its generalized logical and operational semantics, the access to kinds, the use of frozen arguments, conditional rewrite rules with rewrites in their conditions, the `search` command, new predefined modules of integers, natural, rational and floating-point numbers, quoted identifiers, and strings, an improved metalevel redesign, an LTL model checker, an implementation redesign, and some formal tools.

*Full Maude* is an extension of Maude, written in Maude itself by taking advantage of the reflective capabilities made available in Maude's `META-LEVEL` module [7]. Full Maude has been used as a test bed for prototyping new features: parameterization [4], strategies [16], unification [6], etc. This allows to experiment with such features before implementing them at the core level of the C++ Maude implementation.

In this paper we briefly describe the main features introduced in Maude since Maude 2.0, including communication with external objects; a new implementation at the core level of its module algebra with operations for summation and renaming of modules, as well as support for parameterized programming by means of theories and views; new predefined libraries of parameterized data types, supporting efficient versions of lists, sets, maps, and arrays; a linear Diophantine equation solver; a strategy language to be used at the object level; and support for unification. Each section of this paper is devoted to one of these features. As already mentioned, some of these features have previously been experimented with using Full Maude.

The releases of Maude since Maude 2.0 have indeed many other new features and improvements that cannot be described here. We refer the reader to the Maude documentation [7] for more details. The forthcoming book on Maude [6] contains many additional examples and explanations, as well as information on applications and tools.

## 2 Object-message fairness and external objects

Distributed systems can be naturally modeled in Maude as multisets of entities, loosely coupled by some suitable communication mechanism. An important example is object-based distributed systems in which the entities are objects, each with a unique identity, and the communication mechanism is message passing. Maude supports the modeling of object-based systems by providing a predefined module `CONFIGURATION` that declares sorts representing the essential concepts of object, message, and configuration, along with a notation for object syntax that serves as a common language for specifying object-based systems. To specify an object-based system, the user can import `CONFIGURATION` and then define the particular objects, messages, and rules for interaction that are of interest. In addition to simple asynchronous message passing, Maude also supports complex patterns of synchronous interaction that can be used to model higher-level communication abstractions. A new feature in Maude 2.4 is the implementation of an *object-message fair* rewriting strategy that is well suited for executing object system configurations. This strategy ensures that any message sent to any object will eventually be received and processed by the object.

Maude 2.4 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. Configurations that want to communicate with external objects must contain at least one *portal*, which is the constant <> of sort `Configuration` in the predefined module `CONFIGURATION`. Rewriting with external objects that do not reside in the configuration is started by the external rewrite command `erewrite`, which follows the object-message fair rewriting strategy.

Sockets are declared in the `SOCKET` module. The external object named by the constant `socketManager` is a factory for socket objects. To create a client socket, you send it a `createClientTcpSocket` message with the name of the object the reply should be sent to, the name of the server you want to connect to, and the port you want to connect to. The reply will contain the name of the newly created socket. You can then send data to the server with a `send` message, and indicate readiness to accept data by sending a `receive` message. For example, to have communication between two Maude interpreter instances, one of them must take the server role and offer a service on a given port. To create a server socket, you send `socketManager` a `createServerTcpSocket` message. The only thing you can do with a server socket is to accept clients. The socket you use to communicate with that client behaves just like a client socket for sending and receiving. Communications with socket objects have the form `msgName(to, from, ...)`, i.e., the first argument is the intended receiver of the message and the second is the sender.

The following rules illustrate a very naive two-way communication between two Maude interpreter instances. First, the server waits for clients. If one client is accepted, the server waits for messages from it. When the message arrives, the server converts the received data to a natural number, computes its factorial, converts it into a string, and finally sends this string to the client. Once the message is sent, the server closes the socket with the client.

```
rl [created] : < O : Server | A > createdSocket(O, socketManager, LISTENER)
            => < O : Server | A > acceptClient(LISTENER, O) .
rl [accCli] : < O : Server | A > acceptedClient(O, LISTENER, IP, CLIENT)
           => < O : Server | A > receive(CLIENT, O) acceptClient(LISTENER, O) .
rl [rec] : < O : Server | A > received(O, CLIENT, DATA)
        => < O : Server | A > send(CLIENT, O, string(rat(DATA, 10)!, 10)) .
rl [sent] : < O : Server | A > sent(O, CLIENT)
         => < O : Server | A > closeSocket(CLIENT, O) .
```

The Maude command that initializes the server is as follows, where the configuration includes the portal <>.

```
Maude> erewrite <> < aServer : Server | none >
       createServerTcpSocket(socketManager, aServer, 8811, 5) .
```

Using the following rules, the client connects to the server (clients must be created after the server), sends a message representing a number, and then waits for the response. When the response arrives, rewriting ends, because there are no messages that could block the computation.

```
rl [created] : < O : Client | A > createdSocket(O, socketManager, CLIENT)
            => < O : Client | A > send(CLIENT, O, "6") .
rl [sent] : < O : Client | A > sent(O, CLIENT)
         => < O : Client | A > receive(CLIENT, O) .
```

The initial configuration for the client will be as follows, again with portal <>.

```
Maude> erewrite <> < aClient : Client | none >
       createClientTcpSocket(socketManager, aClient, "localhost", 8811) .
```

## 3 Module algebra

As in Clear [2], OBJ [14], CafeOBJ [13] and other specification languages in that tradition, the abstract syntax for writing specifications in Maude can be seen as given by *module expressions*, defining a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations. In fact, structuring is essential in all specification languages, not only to facilitate the construction of specifications from already existing ones—with more or less flexible reusability mechanisms—but also for managing the complexity of understanding and analyzing large specifications. Maude 2.4 now supports module operations for summation, renaming, and instantiation of parameterized modules.

The summation module operation creates a new module that includes all the information in its summands. For example, FLOAT+STRING is the union of the predefined FLOAT and STRING modules.

Renaming a module allows renamings of sorts and operators, and also changing the attributes of the operator being renamed. For example, the renaming

```
fmod RENAMED-INT is
  protecting INT * (sort Zero to MachineZero, ....
                    sort Int to MachineInt,
                    op s_ : Nat -> NzNat to $succ, ....
                    op _divides_ : NzInt Int -> Bool to $divides) .
endfm
```

is part of the construction of machine integers out of arbitrary size integers described later in Section 4.

Theories, parameterized modules, and views are the basic building blocks of *parameterized* programming. A *theory* defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter. Theories have a *loose* semantics, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model, as opposed to the initial model semantics of modules. A *parameterized module* is a module with one or more *parameters*, each of which is expressed by means of one theory. For lists and sets we do not need any requirement on the data elements, and therefore we may use the trivial theory TRIV, with just a sort Elt, as parameter of such modules; but in other cases, say search trees or sorted lists, we may require a particular operator or an order relation by means of the appropriate (either predefined or user-defined) theories describing the specific requirements (see Section 4).

The instantiation of the formal parameters of a parameterized module with actual parameter modules or theories requires a *view* mapping entities from the formal interface theory to the corresponding entities in the actual parameter module. In general, there may be several ways in which the source theory requirements might be satisfied, if at all, by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target. For example, the following view RingToRat is a view from a theory RING of rings to the predefined functional module RAT of rational numbers, where the mapping omits the operators _+_ and _*_ for addition and product in the ring, because they have the same syntax in both the source and the target of the view.

```
view RingToRat from RING to RAT is
  sort Ring to Rat .
  op e to term 1 .
  op z to 0 .
endv
```

Each view declaration has an associated set of *proof obligations*, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques of the style supported for Maude's logic in [8].

For example, the following declaration is the beginning of a parameterized module `POLYNOMIAL` specifying polynomials on a ring and a set of variables (see [7, 6] for full details). Notice that there are two theories: `RING`, for the coefficients, and `TRIV`, for the variables.

```
fmod POLYNOMIAL{R :: RING, X :: TRIV} is .... endfm
```

We can then define the polynomials with rational coefficients and quoted identifiers as variables by instantiating the module above with the previous view `RingToRat` and another view `Qid` from `TRIV` to `QID`, as follows:

```
fmod QID-RAT-POLY is protecting POLYNOMIAL{RingToRat, Qid} . endfm
```

Parameterized modules cannot be summed, even if all the parameters are bound. Parameterized modules may be renamed, but the renaming must not affect any sorts or operators coming from a parameter theory.

## 4 New predefined libraries

Maude has a standard library of predefined modules that, by default, are entered into the system at the beginning of each session, so that any of these predefined modules can be imported by any other module defined by the user. We describe here the new modules that are part of the Maude prelude, 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.

**Random numbers and counters.** The functional module `RANDOM` adds to `NAT` a pseudo-random number generator. The function `random` is the mapping from `Nat` into the range of natural numbers $[0, 2^{32} - 1]$ computed by successive calls to the Mersenne Twister Random Number Generator. The system module `COUNTER` adds a "counter" that can be used to generate new names and new random numbers in Maude programs that do not want to explicitly maintain this state. These modules can be used together to sample various probability distributions, which can then be used to specify *probabilistic models* in Maude [6].

**Machine integers.** Versions of Maude prior to 2.0 supported machine integers in place of the arbitrary size integers of Maude 2.0. For certain applications, such as specifying programming languages that support machine integers as a built-in data type, it is convenient to have a predefined specification for machine integers. Machine integers are efficiently emulated in terms of arbitrary size integers. The parameterized module `MACHINE-INT` takes a bit-width parameter $n \geq 2$, which must be a power of 2, and defines those operations that have a new semantics when applied to machine integers. In many cases this means applying the operation `$wrap` to the results to correctly simulate the wrap-around effect over an overflow on signed fixed bit width integers by, in effect, extending the sign bit infinitely to the left.

**Predefined theories.** The simplest non-empty theory is called `TRIV` and consists of a single sort. A model of this theory is just a set. The intuition behind this simple theory is that the minimum requirement possible on a parameterized data type construction is having a data type as a set of basic elements to build more data on top of it. For example, in the parameterized module `LIST{X :: TRIV}`, we require a set of elements satisfying `TRIV` to then build lists of such elements. The theory `DEFAULT` is slightly more complex than `TRIV`, in that in addition to a sort it also requires that there be a distinguished "default" element in such a sort.

The `STRICT-WEAK-ORDER` theory specifies the notion of *strict weak order*, that is, a strict partial order with the additional requirement that the incomparability relation is transitive. The `TOTAL-PREORDER` theory specifies the concept of *total preorder*, that is, a total binary relation which is reflexive and transitive. The notions of strict weak order and of total preorder relax in different ways the requirements of a total order—specified in `TOTAL-ORDER`—in such a way that they are complementary: the set-theoretic complement of a strict weak order is a total preorder and vice versa. Both kinds of relations capture the notion that the set of elements is split into partitions which are linearly ordered. This situation naturally arises when records are compared on a given field. These theories are used as requirements for the specification of sorting algorithms.

**Lists and sets.** The Maude prelude includes modules `LIST` and `SET`, both parameterized by `TRIV`, defining *lists* and *sets*, respectively. Lists over a given sort of elements are constructed from the constant `nil` (representing the empty list) and singleton lists (identified with the corresponding elements using a subsort declaration) by means of an *associative* concatenation operator, with identity `nil`, written as juxtaposition with empty syntax `__`. Sets over a given sort of elements are built from the constant `empty` and singleton sets (identified with the corresponding elements using a subsort declaration) with an *associative*, *commutative*, and *idempotent* union operator, with identity `empty`, written `_,_`. Furthermore, the `LIST*` and `SET*` modules specify, respectively, *generalized* or *nestable* lists and sets.

The parameterized module `WEAKLY-SORTABLE-LIST` specifies a *stable* version of the *mergesort* algorithm on lists, so that incomparable elements remain in the same relative order as in the list provided as argument. For this notion to be well defined, the corresponding parameter theory is `STRICT-WEAK-ORDER`. The parameterized module `WEAKLY-SORTABLE-LIST'` specifies the same sorting algorithm, but parameterized instead with respect to the `TOTAL-PREORDER` theory. Finally, there is also a parameterized module `SORTABLE-LIST` for sorting lists with respect to the theory `TOTAL-ORDER`.

**Maps and arrays.** Both *maps* and *arrays* represent a function $f$ between two sets as a set of pairs of the form $\{(a_1, f(a_1)), (a_2, f(a_2)), \ldots, (a_n, f(a_n))\}$ in the graph of the function; each pair $(a_i, f(a_i))$ is called an *entry*. The difference between maps and arrays is that the former leave undefined the result of $f$ over those values not present in the set of entries, while the latter assign a "default" value result in that case. This is made explicit in the respective parameter theories: while maps are parameterized with respect to the theory TRIV, arrays are parameterized with respect to DEFAULT that, in addition to a set of data, provides a default value (see above). More specifically, the domain and codomain values of a map come from the parameters of the parameterized data type specified in the module MAP, both of them satisfying the theory TRIV and thus providing sets of elements. The operator insert adds a new entry to a map, but when the first argument already appears in the domain of definition of the map, the second argument is used to *update* the map. There is also a look-up operator. As explained above, arrays work like maps, with the difference that an attempt to look up an unmapped value always returns the default value, i.e., arrays have a *sparse array* behavior. In the same spirit, mappings to the default value are never inserted. The parameterized ARRAY module also contains definitions of appropriate operators for insertion and lookup.

## 5 Linear Diophantine equation solver

The Maude system includes now a built-in linear Diophantine equation solver. The interface to the solver is defined in the file linear.maude which contains the functional module DIOPHANTINE. The current solver finds non-negative solutions of a system $S$ of $n$ simultaneous linear equations in $m$ variables having the form $Mv = c$, where $M$ is an $n \times m$ integer coefficient matrix, $v$ is a column vector of $m$ variables, and $c$ is a column vector of $n$ integer constants. Both matrices and vectors are represented as sparse arrays with their dimensions implicit and their indices starting from 0; for this we make heavy use of the parameterized module ARRAY, described above.

The solver is invoked with the built-in operator natSystemSolve, which takes as arguments the coefficient matrix, the constant vector, and a string naming the algorithm to be used (see below), and returns the complete set of solutions encoded as a pair of sets of vectors [ $A$ | $B$ ]. The non-negative solutions of the linear Diophantine system correspond exactly to those vectors that can be formed as the sum of a vector from $A$ and a non-negative linear combination of vectors from $B$. In particular, if the system $S$ is homogeneous (i.e., $c =$ zeroVector) then $A$ contains just the constant zeroVector and $B$ is the Diophantine basis of $S$, which will be empty if $S$ only admits the trivial solution. If $S$ is inhomogeneous (i.e., $c \neq$ zeroVector) then, if $S$ has no solution, both $A$ and $B$ will be empty; otherwise, $B$ will consist of the Diophantine basis of $S'$, the system formed by setting $c =$ zeroVector, while $A$ contains all solutions of $S$ that are not strictly larger than any element of $B$.

Deciding whether a linear Diophantine system admits a non-negative, nontrivial solution is NP-complete [19]. Furthermore the size of the Diophantine basis of a homogeneous system can be very large. For example the equation: $x+y-kz=0$, for constant $k>0$, has a Diophantine basis (i.e., set of minimal, nontrivial solutions) of size $k+1$.

There are currently two algorithms implemented in Maude. The string "cd" specifies a version of the classical Contejean-Devie algorithm [11] with various improvements. The algorithm is based on incrementing a vector of counters, one for each variable, and so it can only solve systems where the answers involve fairly small numbers. The string "gcd" specifies an original algorithm based on integer Gaussian elimination followed by a sequence of extended greatest common divisor (gcd) computations. Termination depends on the bound on the sum of minimal solutions established in [17], which can cause a huge amount of fruitless search after the last minimal solution has been found. It performs well on the “sailors and monkey” problem from [11].

## 6 Strategy language

An interesting feature of rewriting logic is that, thanks to its reflective capabilities [10], strategies themselves can also be specified in a declarative way by means of rewrite rules *at the metalevel*, that is, by rewrite rules that guide one level up how the rules of the system we are interested in are applied at the object level. Thus, in Maude strategies can be defined by rules at the metalevel in an extension of its `META-LEVEL` module [7]. There is great freedom to define in this way different *strategy languages* [9], which can then be used to specify and execute strategies for any object theory of interest. However, pragmatic considerations are important to guide strategy language designs that can deal well with relevant applications. Therefore, to make the language easier to use, we have made a basic strategy language for Maude available *at the object level* [16, 12], rather than at the metalevel. In general, the same rewrite theory describing a system may be executed with different strategies, which may each have specific advantages depending on the purpose at hand. Our strategy language allows the definition of strategy expressions that control the way a term is rewritten. We have benefitted from our own previous experience designing strategy languages in Maude, and also from the experience of other languages like ELAN [1] and Stratego [21]. Nevertheless, our design is based on a strict separation between the rewrite rules in *system modules* and the strategy expressions, that are specified in separate *strategy modules*. Thus, in our strategy language design it is not possible to use strategy expressions in the rewrite rules of a system module: they can only be specified in a separate strategy module. In fact, this separation makes possible defining different strategy modules to control in different ways the rewrites of a single system module.

A strategy is described as an expression that, when applied to a given term, produces a *set* of terms as a result, given that the process is nondeterministic in general. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and allowing variables in a rule to be instantiated before its application by means of a substitution. For conditional rules—which may contain rewrite conditions—such rewrite conditions can also be controlled by means of strategies. Basic strategies are combined by means of several combinators, including: regular expression constructions (concatenation, union, and iteration), if-then-else, combinators to control the way subterms of a given term are rewritten, and recursion [16]. We have also developed the notion of *generic strategies* (e.g., backtracking, map, etc.), which are applicable not to a single rewrite theory, but to a wide range of rewrite theories satisfying some parametric requirements.

In order to validate our strategy language design, we have mainly focused on automated deduction and programming language semantics applications. Besides the short examples presented in [16], the language has been successfully used in the implementation of the operational semantics of the ambient calculus [18], the two-level operational semantics of the parallel functional programming language Eden [15], basic completion algorithms [20], and congruence closure algorithms [12]. Our first prototype implementation defined the language at the metalevel in the usual reflective way, while keeping the user interface at the object level for ease and convenience. After validating our language design experimentally and reaching a mature language design, the strategy language has been implemented at the C++ level, at which the Maude system itself is implemented, to make the language a stable new feature of Maude 2.4.

## 7 Unification

Maude now includes support for order-sorted unification in a subset of the theories supported for matching. At the object level the unify command takes a pair of terms and computes a complete set of unifiers. If only free theory operators are involved, this set is guaranteed to be minimal. This is achieved using a novel BDD-based algorithm for computing the possible sorts of free variables that avoids introducing redundant solutions. In the metalevel this capability is reflected by the descent function `metaUnify`.

Because, even in the free theory, order-sorted unification may require the introduction of fresh variables, all unifiers are presented as substitutions from the variables occurring in the problem to fresh variables; in the following example `#i:(Nz)Nat` are fresh variables of sort `(Nz)Nat`.

```
fmod NAT is
  sorts NzNat Nat .                subsort NzNat < Nat .
  op _+_ : Nat Nat -> Nat .        op _+_ : NzNat Nat -> NzNat .
  op _+_ : Nat NzNat -> NzNat .    op 0 : -> Nat .
  op s : Nat -> NzNat .            op f : Nat Nat -> Nat .
endfm

Maude> unify f(A:NzNat, B:NzNat) =? f(X:Nat + Y:Nat, Y:Nat + Z:Nat) .
Decision time: 1ms cpu (0ms real)
Solution 1                         Solution 2
A:NzNat --> #1:NzNat + #2:Nat      A:NzNat --> #1:Nat + #2:NzNat
B:NzNat --> #2:Nat + #3:NzNat      B:NzNat --> #2:NzNat + #3:Nat
X:Nat --> #1:NzNat                 X:Nat --> #1:Nat
Y:Nat --> #2:Nat                   Y:Nat --> #2:NzNat
Z:Nat --> #3:NzNat                 Z:Nat --> #3:Nat
```

**Note:** We expect to release Maude 2.4 at RTA. However, many of the features described above are already available in the current version of Maude 2.3. At the time of writing, the features that are partly available are the strategy language and unification, which are being implemented for Maude 2.4.

## References

1. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12:69–95, 2001.

2. R. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In D. Bjørner, ed., *Abstract Software Specifications, 1979 Copenhagen Winter School*, LNCS 86, pages 292–332. Springer, 1980.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. The Maude system. In N. Narendran and M. Rusinovitch, eds., *Rewriting Techniques and Applications, RTA'99, Trento, Italy*, LNCS 1631, pages 240–243. Springer, 1999.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, ed., *Rewriting Techniques and Applications, RTA 2003, Valencia, Spain*, LNCS 2706, pages 14–29. Springer, 2003.
6. 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. Springer, 2007.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Talcott. *Maude Manual (Version 2.3)*, SRI International & University of Illinois at Urbana-Champaign, January 2007. Available at `http://maude.cs.uiuc.edu`.
8. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.
9. M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In *CADE-14 Workshop on Strategies in Automated Deduction, Townsville, Australia*, pages 1–12, 1997.
10. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
11. E. Contejean and H. Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.
12. S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. Boy de la Tour y C. A. Muñoz, eds., *Strategies in Automated Deduction, STRATEGIES 2006, Seattle, Washington*, ENTCS. Elsevier, 2007.
13. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
14. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, eds., *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
15. M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Looking for Eden through Maude and its strategies. In F. López-Fraguas, ed., *V Jornadas sobre Programación y Lenguajes, PROLE 2005*, pages 13–23. Thomson, 2005.
16. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, ed., *Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain*, ENTCS 117, pages 417–441. Elsevier, 2005.
17. L. Pottier. Minimal solutions of linear diophantine systems: bounds and algorithms. In R. V. Book, ed., *Rewriting Techniques and Applications, RTA-91, Como, Italy*, LNCS 488, pages 162–173. Springer, 1991.
18. F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, eds., *Rule-Based Programming, RULE 2005, Nara, Japan*, ENTCS 147, pages 135–161. Elsevier, 2006.
19. A. P. Tomás. *On Solving Linear Diophantine Constraints*. PhD thesis, Univ. do Porto, 1997.
20. A. Verdejo and N. Martí-Oliet. Basic completion by means of Maude strategies. Paper in preparation, 2007.
21. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer, ed., *Domain-Specific Program Generation*, LNCS 3016, pages 216–238. Springer, 2004.