Two Decades of Maude

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

[1] IMDEA Software, Madrid, Spain
[2] Universidad de Málaga, Spain
[3] CSL, SRI International, CA, USA
[4] DSIC-ELP, Universitat Politècnica de València, Spain
[5] Facultad de Informática, Universidad Complutense de Madrid, Spain

Dedicado a José Meseguer, con ocasión de su 65 cumpleaños, con cariño, amistad y agradecimiento por todo el trabajo realizado conjuntamente en estas dos décadas.

Abstract. This paper is a tribute to José Meseguer, from the rest of us in the Maude team, reviewing the past, the present, and the future of the language and system with which we have been working for around two decades under his leadership. After reviewing the origins and the language's main features, we present the latest additions to the language and some features currently under development. This paper is not an introduction to Maude, and some familiarity with it and with rewriting logic are indeed assumed.

1 The Origins

The story of Maude does not begin on a dark and stormy night as many stories do, but on a sunny Californian day. Maude was conceived at the Logic and Specification Group, part of the Computer Science Laboratory at SRI International, in Menlo Park, California, in the Spring of 1990. José Meseguer was leading that group, after working for several years with J. A. Goguen and other colleagues on order-sorted equational logic [47] and its implementation in the OBJ3 language [48], among many other topics. At that time he was proposing a new computational logic which could provide on the one hand a unified model of concurrency [63, 64], and on the other hand declarative support for (concurrent) object-oriented programming [62, 66]. This new logic was thought of as an extension of (order-sorted) equational logic with rules (understood either as logical inference rules or as transitions in a concurrent system) which, as the equations, would also be executed by rewriting, and for this reason was called *rewriting logic*. The good properties of the logic for unifying several models of computation, including concurrent ones, were soon generalized to representing other models of computation and also other logics, so that rewriting logic was proposed as a logical and semantic framework [58, 59].

In the same way that order-sorted equational logic was implemented as a specification and programming language in OBJ3, behind rewriting logic there was a language waiting to be implemented. But the language, soon to be called *Maude*, was there from the beginning. In addition to the papers [62, 66] devoted to the rewriting logic support of object-oriented programming, in those years several other papers were produced from the programming point of view [65, 75, 83], emphasizing the multiparadigm and parallel programming characteristics of the proposal. Moreover, even the more theoretical papers, such as the ones about the logical framework [58, 59], used the Maude notation for presenting examples in such a way that, with slight changes, most of them could be executed in the current implementation of the language. Those examples already used primitive versions of constructs for parameterized specification and some of them were associated with a name, *Maudelog*, for a version of Maude with logical variables and unification, a feature which has taken much longer to be fully implemented (see Section 3). The way "from OBJ to Maude and beyond" has been well explained by José Meseguer himself in [70].

Although the implementation of the Maude language and system were not yet underway, techniques for compilation of rewriting onto parallel architectures were studied as part of the Rewrite Rule Machine (RRM) project [54, 55] in the early nineties. A sublanguage of Maude, called Simple Maude, which included term rewriting, graph rewriting, and object-oriented rewriting, was proposed as part of this project.

The prospects for an implementation of Maude greatly improved in the mid nineties, with the arrival at SRI International of Steven Eker, as a postdoc expert on term rewriting implementation, and Manuel Clavel, as a PhD student to work on rewriting logic reflection [12]. This is indeed the reason for the title of this paper: it is around this date when the Maude team was born. The work developed in that period was shown in the first public presentation of the Maude system [25], which took place at the first Workshop on Rewriting Logic and its Applications (WRLA) in Asilomar, California, in 1996 [67]. Another presentation at the same event showed the first realization of the reflection ideas in rewriting logic and Maude [26].

Coincidentally, Francisco Durán also joined the group during that event, as a PhD student, to work on the Maude module algebra [29], which led to the development of *Full Maude*. Although the advances of all the work being done by the Maude team in all these areas were shown at the second WRLA in Pont-à-Mousson, France, in 1998 [53] (implementation [17], reflection [15, 13], module algebra [34]), the first public release of Maude had to wait yet another year until the end of 1999 [16]. Maude 1 was presented in RTA 1999 [18], in FASE 2000 [21], in an ETAPS 2000 tutorial [19], and in a journal paper published in 2002 [22].

However, that first public release of Maude was a proof-of-concept. Although it already had many interesting features, there were so many other missing features that it was not the end, but the beginning of much more work, as discussed by the "Towards Maude 2.0" paper [20] presented at the third WRLA in Kanazawa, Japan, in 2000 [45]. It required a lot of effort to complete the implementation and also to write a good manual for Maude 2.0, publicly released in the Summer of 2003, with a presentation in RTA 2003 [23]. Among other new features, this new version provided support for membership equational logic, support for rewrite expressions in rule conditions, new predefined modules, a new version of its metalevel, and an LTL model checker.

The Maude 2 features kept increasing and improving along the following years, when we managed to have yearly releases, helped by intense meetings of the Maude team, at that time distributed in different locations both in the US and in Europe, after each edition of the WRLA, in Pisa (2002), Barcelona (2004), and Vienna (2006). We reached an important milestone with the publication in 2007 of the book "All About Maude" [24]. The book coincided with the release of Maude 2.3, where the main features of the language and its implementation stabilized, including parameterized modules, interaction with external objects, and a greater catalogue of predefined—some of them parameterized—modules and views, among others.

Since then, the Maude team has produced several additional releases until the recent Maude 2.7. Most of the new features in Maude 2.4, presented at RTA 2009 [14], and subsequent versions after it, have been related to order-sorted unification and narrowing, which is the subject of Section 3. In this brief summary of the work related to the origins of Maude and rewriting logic we cannot do justice to all the work done by many people around the world in this area; instead, we direct the reader to the survey written by José Meseguer himself on twenty years of rewriting logic, published in 2012 [72], together with an annotated bibliography [60] compiling all the papers on rewriting logic and its applications written in the period 1990–2012.

2 The Language

The close contact with many specification and programming applications has served as a good stimulus for a substantial increase in expressive power of the rewriting logic formalism in general, and of its Maude realization in particular. Maude is a high-performance language and system supporting both equational and rewriting logic computation for a wide range of applications, including development of theorem-proving tools, language prototyping, executable specification and analysis of concurrent and distributed systems, and logical framework applications in which other logics are represented, translated, and executed.

2.1 Generalized Rewrite Theories in Maude

Maude's functional modules are theories in *membership equational logic* [69, 9], a Horn logic whose atomic sentences are either equalities $t = t'$ or membership assertions of the form $t : s$, stating that a term t has a certain sort s. Such a logic extends OBJ3's [48] order-sorted equational logic and supports sorts, subsorts, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains.

A Maude (system) module is a generalized rewrite theory, defined as a 4-tuple $\mathcal{R} = (\Sigma, E \cup Ax, \phi, R)$, where $(\Sigma, E \cup Ax)$ is a membership equational theory, Ax is a set of equational axioms for which rewriting modulo is available,

R is a set of labeled conditional rewrite rules, and ϕ is a function assigning 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. Rewriting in $(\Sigma, E \cup Ax, \phi, R)$ happens modulo the equational axioms Ax. Maude supports rewriting modulo different combinations of associativity (A), commutativity (C), identity (U), left identity (Ul), right identity (Ur), and idempotence axioms. Computationally, rules are interpreted as local transition rules in a possibly concurrent system. Logically, they are interpreted as inference rules in a logical system. This makes rewriting logic both a general semantic framework to specify concurrent systems and languages [68], and a general logical framework to represent and execute different logics [59]. The combination of evaluation strategies and frozen arguments allows Maude to perform context-sensitive rewriting [57] with both equations E and rules R modulo Ax.

Maude accepts module hierarchies of functional and system modules with user-definable mixfix syntax. The Maude system is implemented in C++ and is highly modular. Maude's core is its rewrite engine, which is extensible, and indeed has been extended, in many different ways since its inception. For instance, new equational theories can be "plugged in" and new built-in symbols with special rewriting (equation or rule) semantics may be easily added. To date, rewriting modulo all combinations of associativity, commutativity, left and right identity, and idempotence have been implemented apart from those that contain both associativity and idempotence.

Over the years, the development of Maude has been guided by the goal of providing a better support for both rewriting logic and its underlying membership equational logic. For instance, the duality between its logical and operational views was completed with the addition of the `nonexec` attribute in Maude 2.0. The point is that efficient and complete computation by rewriting is only possible for equational theories that satisfy properties such as confluence, sort-decreasingness, and termination. Similarly, to be efficiently executable, a generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, \phi, R)$ should first of all have $(\Sigma, E \cup Ax)$ satisfying the above executability requirements, and should furthermore be coherent [36].

Executability is of course what we want for programming; but it is too restrictive for specification, transformation, and reasoning purposes. For this reason, there is a linguistic distinction between modules, that are typically used for programming as executable theories, and theories, which need not be executable and are used for specification purposes, for example, to specify the semantic requirements of actual parameters of parameterized modules, or for theorem-proving purposes. Maude supports specification of arbitrary membership equational logic theories and of arbitrary rewrite theories, while at the same time keeping a sharp distinction between executable and non-executable statements (i.e., equations, memberships, or rules) by means of the `nonexec` attribute. Fully executable equational and rewrite theories are called admissible, and satisfy the above-mentioned executability requirements. This support for a disciplined coexistence of executable and non-executable statements allows not only a seamless integration of specification and code, but also a seamless integration of Maude with its formal tools.

Maude includes some built-in functional modules providing convenient high-performance functionality within the Maude system. In particular, the built-in modules of integers, natural, rational, and floating-point numbers, quoted identifiers, and strings provide a minimal set of efficient operations for Maude programmers.

2.2 Reflection in Maude

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant meta-theoretic aspects.

Rewriting logic is reflective [12] in the precise sense of having a universal theory U that can represent any finitely presented rewrite theory T (including U itself) and any terms t, t' in T as terms $\overline{T}$ and $\overline{t}, \overline{t'}$ in U, so that we have the following equivalence

$$T \vdash t \to t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \to \langle \overline{T}, \overline{t'} \rangle.$$

Since U is representable in itself, we can then achieve a "reflective tower" with an arbitrary number of levels of reflection.

Maude efficiently supports this reflective tower through its `META-LEVEL` module, where Maude terms and modules are reified as elements of a data types `Term` and `Module`, respectively. The processes of reducing a term to normal form in a functional module and of rewriting a term in a system module using Maude's default interpreter are respectively reified by *descent* functions `metaReduce` and `metaRewrite`. Similarly, the process of applying a rule of a system module to a subject term is reified by a function `metaApply`. Furthermore, parsing and pretty printing of a term in a signature, as well as key sort operations are also reified by corresponding metalevel functions, and up and down functions to move terms, modules, and views between levels.

The reflective capabilities of Maude provide a great range of possibilities, many of which have been exploited with different purposes. It has been used, for example, to define alternative rewriting strategies, to define strategy languages, to define module operations, and in general to extend Maude in different ways. This extensibility by reflection is exploited in Maude's design and implementation. Full Maude is an extension of Maude written in Maude itself which has been used since the beginnings of Maude as a place in which to design and experiment with new features. For example, a module algebra of parameterized modules, views, and module expressions in the OBJ style was available in Maude through Full Maude [34, 30, 35] long before it was implemented in C++ for (Core) Maude 2.4. Object-oriented modules, with convenient syntax for object-oriented applications, or parameterized views are currently available in Full Maude but not yet in Core Maude. In summary, we have been 'using our own medicine', using Maude to specify our system before facing the effort of implementing it.

Indeed, Full Maude has been more than a place in which to experiment with new features, it has provided basic infrastructure on which to define extensions of Maude such as Real-Time Maude [76].

The reflective capabilities of Maude have also been key for the development of executable implementations of very different formal models and programming languages. See, e.g., the coordination models for distributed objects in [82] and [73], the definition of Mobile Maude [32] and its socket-based distributed implementation [37], or the Maude Action Tool [28], which provides an executable environment for action semantics.

2.3 Maude's Formal Tools

In addition to its core functionality for rewriting, Maude comes with a number of tools. Some of these tools are directly integrated in the core system, which provides specific commands for them, as for the `search` command, for searching for terms satisfying given pattern and condition reachable from a given initial term, or through operators in predefined modules, as in the case of its LTL model checker. Other tools are provided as extensions by different authors, some using the infrastructure provided by Full Maude, such as the Church-Rosser checker, the coherence checker [36], the termination tool [33], the explicit-state model checker for linear temporal logic of rewriting (LTLR) [4, 71], or the LTL logical model checker [2]; and some independently, such as the Maude inductive theorem prover [12], or the sufficient completeness checker [50]. An attempt to bring these tools under a common environment so that they can be used together to keep track of pending proof obligations and help in their interaction to discharge these proof obligations is currently under development in what is being called the Maude Formal Environment [38].

3 The Present: Unification and Narrowing

As mentioned before, Maude inherited many features from its predecessors, such as order-sorted equational logic and the use of commonly occurring attributes like associativity and commutativity, but other features of its predecessors were left behind, e.g., Eqlog [46] 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; and MaudeLog [65] envisioned an integration of order-sorted rewriting logic with queries including logical variables. The paper [40] revisited this topic and showed how many modern programming features can be implemented using Maude.

Unification is a fundamental deductive mechanism used in many automated deduction tasks and it is essential for programming languages with logical variables. Many functional and logic programming languages use an evaluation mechanism called *narrowing* [1], which is a generalization of term rewriting allowing free variables in terms (as in logic programming) and replacing pattern matching by unification in order to (non-deterministically) reduce these terms.

Unification and narrowing were introduced in Maude in 2009 as part of the Maude 2.4 release [14]. In that version of Maude, unification worked for any combination of symbols being either free or associative-commutative (AC), and it was developed by Eker as a built-in feature in Core Maude. Narrowing worked for system modules without equations and relied on the built-in unification algorithm. It supported the concept of symbolic reachability analysis of terms with logical variables, computing suitable substitutions for the variables in both the origin and the destination terms. Narrowing was first implemented in Full Maude, which allowed us to carry on research on its reasoning capabilities. The latest developments in Maude 2.6 were presented at RTA 2011 [31]. First, Eker improved the built-in unification algorithms 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. Second, the concepts of *variant* [27] and *variant-based unification* [43] led to a significant improvement in the reasoning capabilities. Given an equational theory $(\Sigma, E \cup Ax)$, the E, Ax-variants of a term t are the set of all pairs consisting of a substitution σ and the E, Ax-canonical form of $t\sigma$. Variant generation, variant-based unification, and symbolic reachability based on variant-based unification were all implemented in Full Maude.

In the most recent Maude 2.7 version, Eker has extended the available capabilities. First, the built-in unification algorithm allows any combination of symbols being free, C, AC, ACU, CU, U, Ul, Ur. Second, variant generation and variant-based unification are implemented in C++ at the Core Maude level with excellent performance. Note that the former version of variant generation and variant-based unification in Maude 2.6 was implemented for very simple equational theories called *strongly right irreducible*, but the new implementation in Maude 2.7 got rid of this restriction, allowing really complex equational theories and their combinations.

The classical application of narrowing modulo an equational theory is to perform *$E \cup Ax$-unification* by narrowing with oriented equations E modulo axioms Ax. Indeed, the variant-based equational order-sorted unification algorithm implemented in Maude 2.7 is based on a narrowing strategy, called *folding variant narrowing* [43], that terminates when $E \cup Ax$ has the finite variant property [27], even though unrestricted narrowing typically does not terminate when Ax contains AC axioms [27, 43].

An interesting example of the flexibility of folding variant narrowing, even beyond equational unification, is given for the classic *missionaries and cannibals* problem. In this problem, three missionaries and three cannibals must cross a river using a boat. The boat cannot cross the river with no people on board, and cannot carry more than two people. In any of the banks, the missionaries cannot be outnumbered by cannibals, otherwise the cannibals would eat the missionaries.

A solution for this problem was presented by Goguen and Meseguer in [49] as an equational logic program, requiring constraint-solving features, logical variables, order-sorted types, and axioms. Features of the original solution have been adapted to the current equational variant-based programming features available in Maude using the ideas of [40], and the resulting module is shown in Figure 1.

The imported module `TRIPLIST` defines trip lists (`TripList`), with concatenation operator `_*_` as constructor[6] and a length operation `#_`. Module `PSET` defines sorts `Elem` of people and `PSet` of multisets of people. Multisets are constructed with union operator `_+_`, which is declared as associative, commutative and with an identity symbol, and come with operators `_-_` for removal and `_/\_` for intersection.

Our aim is to find a list of trips, where each trip is a term rooted by a predicate `boat` with a set of missionaries and cannibals. Odd positions in the list represent trips from the left bank to the right bank, and even positions trips from right to left. The `MAC` module defines constants for the missionaries (`taylor`, `helen`, and `william`) and the cannibals (`umugu`, `nzwave`, and `amoc`). `lb(L)` (resp., `rb(L)`) represents the people set in the left (resp., right) bank after the sequence of trips `L`. `mset(PS)` (resp., `cset(PS)`) gives the subset of missionaries (resp., cannibals) in `PS`. The function `boatok` checks whether a trip is ok — one or two people in the boat, where these are from the set of defined cannibals and missionaries — and `solve` is the general predicate for checking/generating the trip list solution — a trip list is a solution if it is a 'good' list, and the sequence of trips leaves the left bank empty. A trip list `L * T` is good if `T` is a valid trip (`boatok`), the sublist `L` is good, and the number of cannibals in each bank is smaller than the number of missionaries in the same bank for each trip in the sequence.

The key is therefore in the definition of the `Success` sort in the `SUCCESS` module. The `success` sort has a constant `success`, an operator `_>>_` that defines the conditional evaluation of constraints, such that the left side is evaluated before the right side, and `_=:=_`, which represents unification between two terms. For each sort, `_=:=_` is defined only in the positive cases, returning `success`; see below for its definition for sort `Bool`.

```
  op _>>_ : [Success] [Success] -> [Success] [frozen(2)] .
  rl success >> X:[Success] => X:[Success] .

  op _=:=_ : Bool Bool -> [Success] [comm] .
  rl X:Bool =:= X:Bool => success .
```

Folding variant narrowing is performed using those equations labeled with the `variant` flag, while the remaining equations are used as usual in Maude. Here we can ask for solutions to the very general problem of the names of missionaries and cannibals carried from one side to the other of the river.

[6] The original solution assumes that lists are created using an associative symbol, but unification modulo associativity is infinitary and it is not available in Maude. The `_*_` operator is therefore not declared associative.

```
fmod MAC is
  pr SUCCESS + TRIPLIST + PSET .
  ops taylor helen william : -> Elem [ctor] .
  ops umugu nzwawe amoc : -> Elem [ctor] .
  var L : TripList .    var T : Trip .    var PS : PSet .

  op gen : Elem -> [Success] .
  eq gen(taylor) = success [variant] .    eq gen(taylor) = success .
  eq gen(helen) = success [variant] .     eq gen(helen) = success .
  eq gen(william) = success [variant] .   eq gen(william) = success .
  eq gen(umugu) = success [variant] .     eq gen(umugu) = success .
  eq gen(nzwawe) = success [variant] .    eq gen(nzwawe) = success .
  eq gen(amoc) = success [variant] .      eq gen(amoc) = success .

  op m0 : -> [PSet] .                     op c0 : -> [PSet] .
  eq m0 = taylor helen william  .         eq c0 = umugu nzwawe amoc  .
  op mset : PSet -> [PSet] .              op cset : PSet -> [PSet] .
  eq mset(PS) = PS /\ m0 .                eq cset(PS) = PS /\ c0 .

  op boatok : Trip -> [Success] .         op boat : PSet -> Trip [ctor] .
  eq boatok(boat(X:Elem)) = gen(X:Elem) .
  eq boatok(boat(X1:Elem X2:Elem))
   = gen(X1:Elem) >> gen(X2:Elem) >> ((X1:Elem =/= X2:Elem) =:= true) .

  ops lb rb : TripList -> [PSet] .
  eq lb(nil) = m0 c0 .
  eq lb(L * boat(PS))
    = if (even # L) then (lb(L) - PS) else (lb(L) /\PS) fi .
  eq rb(nil) = empty .
  eq rb(L * boat(PS))
    = if (even # L) then (rb(L) /\ PS) else (rb(L) - PS) fi .

  op good : TripList -> [Success] .
  eq good(nil) = success .
  eq good(L * T)
   =  boatok(T)
      >> good(L)
      >> ( (# cset(lb(L * T)) =< # mset(lb(L * T))
            or (# mset(lb(L * T)) == 0))
           and
           (# cset(rb(L * T)) =< # mset(rb(L * T))
            or (# mset(rb(L * T)) == 0)) ) =:= true .

  op solve : TripList -> [Success] .
  eq solve(L) = good(L) >> (lb(L) == empty) =:= true .
endfm
```

Fig. 1. Missionaries and cannibals example

```
Maude> variant unify
         solve(nil * boat(E1:Elem E2:Elem) *
               boat(E1:Elem) * boat(E3:Elem E4:Elem) *
               boat(E2:Elem) * boat(E5:Elem E6:Elem) *
               boat(E6:Elem E3:Elem) * boat(E1:Elem E6:Elem) *
               boat(E4:Elem) * boat(E2:Elem E4:Elem) *
               boat(E6:Elem) * boat(E6:Elem E3:Elem))   =?   success  .

Unifier #1
E1:Elem --> helen    E2:Elem --> amoc     E3:Elem --> umugu
E4:Elem --> nzwawe   E5:Elem --> william  E6:Elem --> taylor
Unifier #2
E1:Elem --> william  E2:Elem --> amoc     E3:Elem --> umugu
E4:Elem --> nzwawe   E5:Elem --> helen    E6:Elem --> taylor
...
Unifier #36
E1:Elem --> helen    E2:Elem --> umugu    E3:Elem --> amoc
E4:Elem --> nzwawe   E5:Elem --> taylor   E6:Elem --> william
```

Enumerating all thirty-six solutions takes only a few minutes thanks to the efficient implementation of folding variant narrowing in Core Maude. The most general question is `variant unify solve(L) =? success`, which enumerates all the necessary boat movements from one bank to the other and the missionaries and cannibals moved each time. However, we would have to add more variant equations, for recursive instantiation of a variable of sort `Triplist`, and for recursive instantiation of a variable of sort `PSet`, apart of the current variant equations for instantation of variables of sort `Elem`. The current implementation in Maude is not able to handle the folding variant narrowing search space associated to a unification problem like that, though it will enumerate the solutions given enough resources.

The modern application of narrowing with rules R modulo $E \cup Ax$ is that of *symbolic reachability analysis* [74]. In this case, the rules R are understood as transition rules instead of equations. Narrowing is a complete deductive method [74] for symbolic reachability analysis, that is, for solving existential queries of the form $\exists \overline{x}\ t(\overline{x}) \to^* t'(\overline{x})$ in the sense that the formula holds for R iff there is a narrowing sequence $t \leadsto^*_{R,E\cup Ax} u$ such that u and t' have an $E \cup Ax$-unifier. Furthermore, in symbolic reachability analysis, we may be interested in verifying properties more general than existential properties of the form $\exists X\ t \to^* t'$, since one can generalize the above reachability property to properties of the form $\mathcal{R}, t \models \varphi$, for φ a temporal logic formula. The papers [42, 2] show how narrowing can be used (again, both at the level of transitions with rules R and at the level of equations E) to perform *logical model checking*. Two distinctive features are: (i) the term t does not describe a single initial state, but a possibly infinite set of instances of t (i.e., a possibly infinite set of initial states); and (ii) the set of reachable states does not have to be finite. Therefore, standard model-checking techniques may not be usable, because of a possible double infinity: in the number of initial states, and in the number of states reachable for each of those initial states.

So far, the most successful story about rewriting logic with narrowing is the Maude-NPA protocol analyzer [41], where cryptographic protocols are formally specified as order-sorted rewrite theories and the security analysis is performed in a backwards way, from an attack state to an initial state.

4 The Near Future: Rewriting Modulo SMT

The rapid progress of *satisfiability modulo theories* (SMT) solvers [7] has been one of the most important developments in automated verification and reasoning. A feature recently added to Maude (in an internal version, not publicly released at the time of the publication of this paper) is support for *rewriting modulo SMT* [78], so that functional and system modules can have conditions dealing with SMT data types, which are then solved by the usually more effective SMT solvers.

SMT solvers are decision procedures for an existential fragment of first-order logic with equality, where variables range over SMT data types, such as Booleans, integers, and reals. After presenting the way in which rewriting modulo SMT is being implemented in Maude in Section 4.1, we describe a sample application in Section 4.2.

4.1 Maude SMT

When performing rewriting modulo SMT, the object being rewritten is a symbolic representation of a (possibly infinite) family of terms. In its current Maude implementation, the representation of such family of terms is an ordered pair, where the first component is a term which may include variables ranging over data types supported by an SMT solver and SMT operators on those variables, and the second component is a constraint on those SMT variables. Rewriting proceeds as a search where each rewrite rule may have a condition, interpreted as an SMT constraint. In order to make a rewrite step, the accumulated constraints must be satisfiable, as checked by an SMT solver; when a conditional rule succeeds, the constraint it enforced on the SMT variables in the new term is 'conjuncted' with the existing constraint. Since Maude has no built-in knowledge of the SMT theories, no simplification of the accumulated constraint is performed.

Maude's interface to SMT data types closely follows the SMT-LIB standard [6]. In particular there are functional modules `BOOLEAN`, `INTEGER`, `REAL`, and `REAL-INTEGER` which provide signatures for the SMT-LIB theories of Booleans, integers, reals, and reals combined with integers, respectively. Although the current implementation has some restrictions, we expect to have a full implementation of rewriting modulo SMT in a near future release of Maude.

An SMT rewriting search is initiated with the `smt-search` command, which has a syntax similar to the syntax of the `search` command. The start term may only include SMT variables, which may also appear in the pattern term, and condition.

To give a flavor of how rewriting modulo SMT works, let us consider a small example. We define the *gcd* function using state transitions on a pair of SMT integers that encode Euclid's algorithm.

```
mod EUCLID is protecting INTEGER .
  sort State .
  op gcd : Integer Integer -> State .
  op return : Integer -> State .

  vars I J K X Y Z : Integer .

  crl gcd(X, Y) => gcd(X - Y, Y) if X > Y = true .
  crl gcd(X, Y) => gcd(X, Y - X) if X < Y = true .
  crl gcd(X, Y) => return(X) if X = Y .
endm
```

We then ask about the existence of a pair of integers X, Y such that $gcd(X, Y) = 3$ and $X + Y = 27$.

```
Maude> smt-search [1] gcd(X, Y) =>* return(Z)
         such that Z = 3 /\ X + Y = 27 .

Solution 1
empty substitution
where Z === 3 and X + Y === 27 and X > Y and X - Y > Y and
  X - Y - Y > Y and X - Y - Y - Y < Y and X - Y - Y - Y ===
  Y - (X - Y - Y - Y) and Z === X - Y - Y - Y
```

Maude searches the (typically infinite) tree of SMT rewrites on (term, constraint) pairs for a state that matches the pattern and satisfies the constraint on the variables given in the command. In the success case, it returns a substitution for non-SMT variables in the pattern and a satisfiable constraint on SMT variables. If the search graph is infinite as in this case, the command will not terminate in the failure case unless a depth bound is given.

Like most other Maude commands, `smt-search` is reflected at the metalevel by a corresponding descent function.

Currently, Maude uses CVC4 [5] as its backend SMT solver, however calls to the SMT solver are implemented via an abstract interface and wrappers for other SMT libraries could easily be added in the future.

4.2 Symbolic Analysis of Distance-Bounding Protocols

Having the possibility of using constrained variables gives us the opportunity of making finite potentially infinite search spaces. As for a more interesting application of rewriting modulo SMT, let us consider the case of *distance-bounding* *protocols* [10], a class of security protocols that infer an upper bound on the distance between two agents from the round trip time of messages. This is used, for example, for controlling some kinds of access and for clock synchronization. In a distance-bounding protocol session, the verifier (V) and the prover (P) exchange messages:

$$\begin{aligned} V &\rightarrow P : m \\ P &\rightarrow V : m' \end{aligned}$$

where m is a challenge and m' is a response message (constructed using the components of m such as nonces in m). In order to infer the distance to the prover, the verifier remembers the time, t_0, when the message m was sent, and the time, t_1, when the message m' returns. From the difference $t_1 - t_0$ and the assumptions on the speed of the transmission medium, v, the verifier can compute an upper bound on the distance to the prover, namely $(t_1 - t_0) \times v$.

In [52], a novel attack on distance bounding called *attack in-between-ticks* is presented. The attack is formalized using a model in which provers, verifiers, and attackers may have different clock rates, processing speeds, or observation granularity. The model is based on a multiset-rewriting formalism called *timed local state transition systems* [51] which supports both discrete and dense time. The key insight for the attack is that an attacker can mask his location by exploiting the fact that a message may be sent at any point between two clock ticks of the verifier's clock, while the verifier measures the time at discrete clock ticks. For example, if the time bound is 3, a message could start at time 1.7, which is 2 on the verifier's clock, and the reply received at 4.9, which is 5 on the verifier's clock. From the verifier's perspective the attacker is within range, since $5 - 2 = 3$, but in fact the round trip time was $4.9 - 1.7 = 3.2$.

The model was formalized in Maude SMT and `smt-search` was used to find a symbolic representation of a family of attacks. Using Maude SMT the potentially infinite search space becomes finite, by treating the distance between the verifier and the prover as a constrained variable.

To illustrate the use of SMT, we show the `Tick` rule which advances system time following the approach of Real-Time Maude [76].

```
var  S : Soup .
vars T T1 T2 : Real .

crl [Tick] : { S (Time @ T) (vTime @ T1) }
  =>  { S (Time @ T2) (vTime @ T1) }
  if (T2 > T and (T2 < T1 + 1/1)) = true
  [nonexec] .
```

Here, a system state is a soup of timed facts (`F @ T`) enclosed in brackets. There is a unique fact, `Time @ T`, representing the physical time. The fact `vTime @ T1` represents time as perceived by the verifier. In Real-Time Maude execution and search use a time sampling strategy. In contrast, using Maude SMT the new time is left symbolic, with constraints on its range. Here the constraint says that the next time should be greater than the current time, but should not advance beyond the next verifier time ($T1 + 1$).

The following command searches for a state in which the verifier, `v`, accepts a reply from the prover, `p`, `(Ok(< p, M:Msg >) @ T:Real)`, where the distance bound is `3` (the allowed round trip time is `2 * 3`) and the distance from the verifier to the prover, `dvp`, (or to an attacker, `dva`) is greater than `3`.

```
Maude> smt-search [1] { gensym(0)
                        dist(dva, dvp)
                        (Time @ 0/1)
                        (vTime @ 0/1)
                        (V0(p) @ 0/1)
                        (P0(v) @ 0/1) }
                  =>+ { (Ok(< p, M:Msg >) @ T:Real)
                        S:Soup }
                  such that (dva > 3/1 and dvp > 3/1 ) = true .
```

One (simplified) solution is:

```
S --> ...
  (Time @ toInteger(dvp + dvp) + 1/1)
  (RStart(< p, n(0) >) @ 0/1)                   *** the real start
  (RStop(< p, n(0) >) @ dvp + dvp)              *** the real stop
  (V1(< p, n(0) >) @ toInteger(dvp + dvp) + 1/1) *** the verifiers view
 where dva > 3/1 and dvp > 3/1 and
 toInteger(dvp + dvp) <= 2/1 * 3/1   *** the discrete time looses here
 and T === toInteger(dvp + dvp) + 1/1
 ...
M --> n(0)
```

The verifier's start time is `1/1` thus the elapsed time is `toInteger(dvp + dvp) <= 2/1 * 3/1`.

5 Pathway Logic

There have been many applications of Maude in many different areas, including some as diverse as security [41], cyber-physical systems [44, 3], and model-driven engineering [8, 77] (see, for example, the survey [72]). We very briefly discuss in this section one of them, Pathway Logic, which innovates by modeling nature rather that the usual digital artifacts, very nicely illustrating the modeling capabilities of Maude.

Pathway Logic (PL) is a system for modeling and reasoning about cellular processes such as signal transduction, metabolism, and cell-cell communication in the immune system. The semantic underpinnings of PL is Maude, and José Meseguer was part of the original PL team that developed the key ideas [39]. The first instance of a PL model was a model of a cancer-related signaling pathway crafted in Maude in 2000. In order to facilitate scaling up and interacting with the PL models, the executable Maude model has been augmented with the Pathway Logic Assistant (PLA), an interactive graphical interface that allows a user to easily create specific models, and browse and query them [81].

In addition, a mechanism for semantically grounding the language with links to standard databases has been put in place, and a substantial collection of formal models has been developed [80, 56].

5.1 About Pathway Logic

Signal transduction is the mechanism by which cells sense their environment, process this information, and make decisions: what proteins to produce, what metabolic pathways to activate, whether to replicate, move, or possibly die. Typically, the signal is a chemical or protein in the cells environment that binds to a receptor protein (in the cell membrane). The receptor becomes *active*, initiating the signaling process. The signal is transmitted by change in state and location of proteins involved.

In PL a cell state is represented as a soup of occurrences, where each occurrence has three components: a protein or other biomolecule (gene, metabolite, etc.), a modifier, and a location. The modifier indicates the state of the protein, including binding of small molecules or phosphates, or ability to act on other proteins (enzyme activity). For example, the term `< [Hras - GTP], CLi >` is the occurrence of the protein `Hras` modified by binding to the small molecule `GTP` (Guanosine TriPhosphate), attached to the inside of the cell membrane (`CLi`).[7] Signal transduction steps are formalized as local rewrite rules operating on the relevant part of the cell state.

As an example, rule `1.EgfR.act` formalizes the initiation of signaling in response to the presence of Egf (Epidermal growth factor) in a cell's exterior.

```
rl [1.EgfR.act]
  < ?ErbB1L:ErbB1L, XOut > < EgfR, EgfRC >
  =>
  < ?ErbB1L:ErbB1L :[ EgfR - Yphos], EgfRC >
```

Here, `?ErbB1L:ErbB1L` is a variable of sort `ErbB1L`, `XOut` is the cells external environment, and the infix operator `_:_` represents complex formation. In the model there are two proteins of sort `ErbB1L`: `Egf` and `Tgfa` (Transforming growth factor alpha).

How does the `Egf` signal propagate? To answer the question, we use the PLA. Figure 2 shows the subnet containing all rules relevant to activation (binding to `GTP`) of `Hras` in response to `Egf`. The subnet is generated by backwards collection from the goal `H = < [Hras - GTP], CLi >` in the `Egf` response network.

A specific execution path can be found by mapping the subnet and goal to the language of the LoLA model checker [79] asserting that the goal cannot be reached. If there is a counter example, LoLA returns a list of transitions that can be fired to reach the goal. Maude then converts this to a network and generates the expression to display the interactive graph.

[7] There are in fact two internal syntactic forms for representing cell state: a soup of locations; and a soup of occurrences. We restrict attention to the latter as occurrences correspond to places in a Petri net.

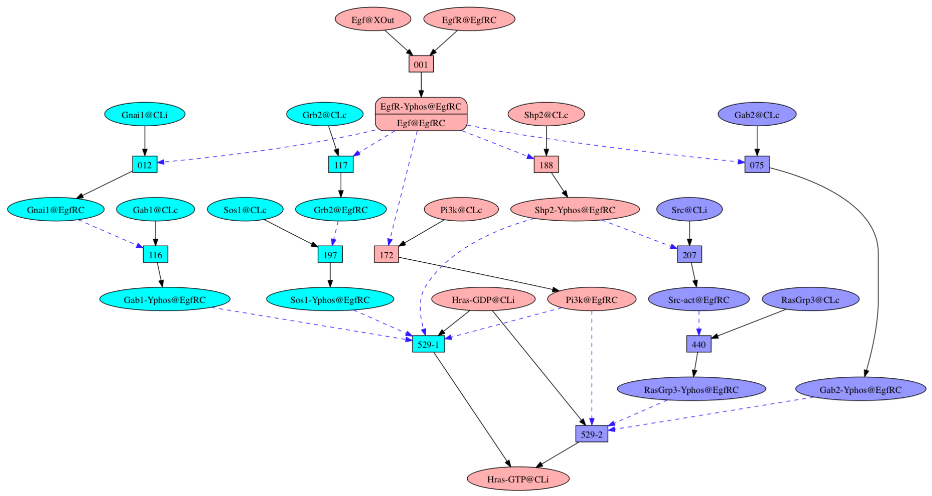

Fig. 2. Activation of Hras in response to Egf. The full subnet with two execution paths compared. Pink belongs to both paths, blue and cyan to different paths. Ovals represent occurrences, rectangles represent rules, with input ovals connected by incoming arrows and output connected by outgoing arrows. Dashed arrows connect occurrences that both input and output (enzymes).

Figure 2 also shows the result of comparing two different execution paths, the second is the result of removing one of the occurrences from the subnet (simulating a knockout) before asking LoLA.

5.2 Maude's Role in Pathway Logic

The integration of the Maude executable model and PLA is achieved using the IOP platform for communication amongst a group of *actors*. Maude's *loop mode and reflection* are key to turning Maude into an actor [61]. The PL Maude actor extends the IMaude actor [61] with rules for handling PL specific requests. IMaude provides data structures for representing state, managing asynchronous interaction, and saving and restoring state. The latter makes essential use of Maude's capabilities for pretty-printing and parsing. PLA is an actor built using an interpreter of a Scheme-like language, JLambda, layered on top of Java. The Maude PL actor listens for requests from PLA and generates expressions in the JLambda language to instruct PLA to construct and render interactive network graphs.

Proteins, modifications, and locations are given different names by different biologists. Thus, to understand what a biological model is talking about it is important to link the names (constants) used to reference databases that provide canonical names and additional information. This is accomplished using the `metadata` attribute of Maude operator declarations. PL operator metadata is a string encoding an S-expression that maps key words to values such as database access identifiers, synonym lists, and biological classifiers. With the aid of a meta-model (componentInfoSpec) the metadata is rendered as a menu of information and active links that are presented when the user clicks on a graph element. The following is the operator declaration for `Egf`. The metadata includes identifiers used by two reference databases (`spnumber` for UniProt and `hugosym` for HGNC), as well as a list of synonyms that can be used when a user unfamiliar with PL naming is searching for information.

```
op Egf : -> ErbB1L [ctor metadata "(\
  (category Ligand)\
  (spnumber P01133)\
  (hugosym EGF)\
  (synonyms \"Pro-epidermal growth factor\"\
            \"EGF_HUMAN\"))"] .
```

Another important requirement for a model is to justify the rules describing signal transduction steps. Where do these rules come from? They are inferred from experimental observations of what is present in a cell (and where) and of response of a cell or a population of cells to different perturbations. In PL each rule is linked (via metadata) to an evidence page that contains a formal representation of the experimental observations used in inferring the rule.[8]

As noted above, the PL rule base consists of symbolic rules that have variables whose sort consists of a finite set of proteins or modifications or locations. Petri net tools (and graphical representations) need concrete instances. One possibility is simply to generate all possible instances. This generates many useless rules, because proteins of a given sort behave similarly in some cases, and differently in other cases. Thus we take advantage of the Maude function to generate all matches of a rule to a given state to generate only concrete rules that are possibly reachable from initial states of interest.

In the spirit of *May I borrow your logic* [11], PL supports multiple representations of a PL knowledge base: Maude signature and rules, Petri nets, JSON, and SBML (Systems Biology Markup Language). The Petri net representation is used for efficient analysis of large networks that takes advantage of the restricted nature of PL rules. SBML is an exchange format used to share models between different systems biology tools including simulators and visualizers. The JSON representation is used to treat a PL knowledge base as a database with efficient query of static relations; for example, finding all rules that involve the protein whose UniProt identifier is P01112 (Hras). Maude could be programmed to answer such queries, but putting the information in a database makes it more widely accessible. In addition, the JSON representation is an easily parsable exchange format that is being used by researchers developing modeling and analysis tools.

Transformations to different representation systems is done by reflecting the PL model to the metalevel and transforming it to a representation in the target system, which can then be written to a file with the help of the PLA actor.

[8] Rules are currently inferred by a human curator.

6 Further Ahead

We have presented our view of the two decades we have been involved in the development of Maude. We have gone from the first years of Maude to the current state of the system and the new features currently under development. What features will Maude users be expecting in the future? Or what features will have a bigger impact for Maude in the future? As we explained before, a current trend in programming languages is to become multi-paradigm, offering flexibility and simplicity for problem specification and solving, and we believe our efforts will go into that direction.

On the one hand, all the logical and symbolic features will be boosted. For instance, other SMT libraries such as veriT could be added to Maude. Also, variant generation and variant-based unification, as well as narrowing in system theories, consider only unconditional rules and equations: conditional narrowing, both at the level of equational logic and rewriting logic, should be added. Moreover, more built-in unification algorithms will be included in Core Maude: we have explored unification algorithms for associativity, for homomorphic encryption, for exclusive-or, etc. Furthermore, we envision conditional narrowing combined with SMT solvers, so that many different reasoning facilities are seamlessly combined.

On the other hand, tool support will be incremented. Maude would not be such a good logical framework without its metalevel capabilities. The Maude Formal Environment will be improved with better tool integration. And we should not forget about tools built on top of Maude, such as the Pathway Logic Assistant, Real-Time Maude, or the Maude-NPA protocol analyzer.

Acknowledgements

Francisco Durán was partially supported by Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech and Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER) under grant POLYCIMS (ref. TIN2014–52034–R). Santiago Escobar was partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN2013–45732–C4–1–P. Narciso Martí-Oliet was partially supported by Spanish MINECO under grant StrongSoft (TIN2012–39391–C04–04) and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).

References

1. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.
2. K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

3. K. Bae, J. Krisiloff, J. Meseguer, and P. C. Ölveczky. Designing and verifying distributed cyber-physical systems using multirate PALS: an airplane turning control system case study. *Sci. Comput. Program.*, 103:13–50, 2015.
4. K. Bae and J. Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming*, 99:193–234, 2015.
5. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. Kind, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, Berlin, 2011.
6. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available from http://smt-lib.org.
7. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
8. A. Boronat and J. Meseguer. An algebraic semantics for MOF. *Formal Asp. Comput.*, 22(3-4):269–296, 2010.
9. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1):35–132, 2000.
10. S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In *EUROCRYPT*, pages 344–359, 1993.
11. M. Cerioli and J. Meseguer. May I borrow your logic? (Transporting logical structure along maps). *Theoretical Computer Science*, 173:311–347, 1997.
12. M. Clavel. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language*. PhD thesis, Universidad de Navarra, Spain, February 1998.
13. M. Clavel. Reflection in general logics, rewriting logic, and Maude. In Kirchner and Kirchner [53], pages 71–82.
14. M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in Maude 2.4. In R. Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.
15. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [53], pages 331–352.
16. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, http://maude.cs.uiuc.edu/maude1/manual/.
17. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude as a metalanguage. In Kirchner and Kirchner [53], pages 147–160.
18. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999, Proceedings*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer, 1999.
19. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. A Maude tutorial. Tutorial distributed as documentation of the Maude

system, Computer Science Laboratory, SRI International. Presented at the *European Joint Conference on Theory and Practice of Software, ETAPS 2000*, Berlin, Germany, March 25, 2000, Mar. 2000.

20. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In Futatsugi [45], pages 294–315.
21. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Using Maude. In T. S. E. Maibaum, editor, *Fundamental Approaches to Software Engineering, Third International Conference, FASE 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25-April 2, 2000, Proceedings*, volume 1783 of *Lecture Notes in Computer Science*, pages 371–374. Springer, 2000.
22. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
23. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2003.
24. 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, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
25. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [67], pages 65–89.
26. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [67], pages 126–148.
27. H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.
28. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In T. Rus, editor, *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2000.
29. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language.* PhD thesis, Universidad de Málaga, Spain, June 1999.
30. F. Durán. The extensibility of Maude's module algebra. In T. Rus, editor, *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2000.
31. F. Durán, S. Eker, S. Escobar, J. Meseguer, and C. L. Talcott. Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In M. Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
32. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications,*

Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zürich, Switzerland, September 13-15, 2000, Proceedings, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2000.

33. F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude termination tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.
34. F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [53], pages 174–195.
35. F. Durán and J. Meseguer. Maude's module algebra. *Science of Computer Programming*, 66(2):125–153, April 2007.
36. F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 81(7–8):816–850, 2012.
37. F. Durán, A. Riesco, and A. Verdejo. A distributed implementation of Mobile Maude. *Electr. Notes Theor. Comput. Sci.*, 176(4):113–131, 2007.
38. F. Durán, C. Rocha, and J. M. Álvarez. Towards a Maude formal environment. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 329–351. Springer, 2011.
39. S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway Logic: Symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, January 2002.
40. S. Escobar. Functional logic programming in Maude. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *Lecture Notes in Computer Science*, pages 315–336. Springer, 2014.
41. S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2007.
42. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In F. Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.
43. S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.
44. M. Fadlisyah, P. C. Ölveczky, and E. Ábrahám. Formal modeling and analysis of interacting hybrid systems in HI-Maude: What happened at the 2010 sauna world championships? *Sci. Comput. Program.*, 99:95–127, 2015.
45. K. Futatsugi, editor. *Proceedings of the Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18-20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.
46. J. Goguen and J. Meseguer. Eqlog: Equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.

47. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
48. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer Academic Publishers, 2000.
49. J. A. Goguen and J. Meseguer. Equality, types, modules, and (why not ?) generics for logic programming. *Journal of Logic Programming*, 1(2):179–210, 1984.
50. J. Hendrix, J. Meseguer, and H. Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.
51. M. Kanovich, T. B. Kirigin, V. Nigam, A. Scedrov, C. Talcott, and R. Perovic. A rewriting framework for activities subject to regulations. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA)*, volume 15 of *LIPIcs*, pages 305–322. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
52. M. I. Kanovich, T. B. Kirigin, V. Nigam, A. Scedrov, and C. L. Talcott. Discrete vs. dense times in the analysis of cyber-physical security protocols. In R. Focardi and A. C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 259–279. Springer, 2015.
53. C. Kirchner and H. Kirchner, editors. *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, 1998.
54. P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms, DIMACS Workshop, May 9-11, 1994*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 309–339. American Mathematical Society, 1994.
55. P. Lincoln, N. Martí-Oliet, J. Meseguer, and L. Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 1994.
56. P. D. Lincoln and C. Talcott. Symbolic systems biology and pathway logic. In S. Iyengar, editor, *Symbolic Systems Biology*, pages 1–29. Jones and Bartlett, 2010.
57. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.
58. N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.

59. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.
60. N. Martí-Oliet, M. Palomino, and A. Verdejo. Rewriting logic bibliography by topic: 1990-2011. *Journal of Logic and Algebraic Programming*, 81(7-8):782–815, 2012.
61. I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An interactive extension of Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27–28, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 315–333. Elsevier, 2005. http://www.sciencedirect.com/science/journal/15710661.
62. J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *Proceedings of the ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 21-25, 1990*, pages 101–115. ACM Press, 1990.
63. J. Meseguer. Rewriting as a unified model of concurrency. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 1990.
64. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
65. J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2-4, 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer, 1992.
66. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.
67. J. Meseguer, editor. *Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.
68. J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, 7th International Conference, Pisa, Italy, August 26–29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.
69. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3-7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.
70. J. Meseguer. From OBJ to Maude and beyond. In K. Futatsugi, J. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 252–280. Springer, 2006.
71. J. Meseguer. The temporal logic of rewriting: A gentle introduction. In P. Degano, R. D. Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 354–382. Springer, 2008.
72. J. Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81(7-8):721–781, 2012.

73. J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36. Springer, 2002.
74. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
75. J. Meseguer and T. C. Winkler. Parallel programmming in Maude. In J.-P. Banâtre and D. L. Métayer, editors, *Research Directions in High-Level Parallel Programming Languages, Mont Saint-Michel, France, June 17-19, 1991, Proceedings*, volume 574 of *Lecture Notes in Computer Science*, pages 253–293. Springer, 1992.
76. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
77. J. E. Rivera, F. Durán, and A. Vallecillo. Formal specification and analysis of domain specific models using Maude. *Simulation*, 85(11-12):778–792, 2009.
78. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. In S. Escobar, editor, *Proceedings of the 10th International on Workshop Rewriting Logic and Its Applications (WRLA 2014)*, volume 8663 of *Lecture Notes in Computer Science*, pages 247–262. Springer, Berlin, 2014.
79. K. Schmidt. LoLA: A Low Level Analyser. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer, 2000.
80. C. Talcott. Symbolic modeling of signal transduction in pathway logic. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, editors, *2006 Winter Simulation Conference*, pages 1656–1665, 2006.
81. C. Talcott and D. L. Dill. Multiple representations of biological processes. *Transactions on Computational Systems Biology*, 2006.
82. C. L. Talcott. Coordination models based on a formal model of distributed object reflection. *Electr. Notes Theor. Comput. Sci.*, 150(1):143–157, 2006.
83. T. C. Winkler. Programming in OBJ and Maude. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 229–277. Springer, 1993.