Building Equational Proving Tools by Reflection in Rewriting Logic*

M. Clavel
Univ. de Navarra, Spain

F. Durán
Univ. de Málaga, Spain

S. Eker
SRI International, USA

J. Meseguer
SRI International, USA

August 7, 2000

Abstract

This paper explains the design and use of two equational proving tools, namely an inductive theorem prover—to prove theorems about equational specifications with an initial algebra semantics—and a Church-Rosser checker—to check whether such specifications satisfy the Church-Rosser property. These tools can be used to prove properties of order-sorted equational specifications in CafeOBJ [17] and of membership equational logic specifications in Maude [9, 12]. The tools have been written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement. The fact that rewriting logic is a reflective logic, and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the design of the tools.

*Supported by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan (IPA), DARPA and NASA through Contract NAS2-98073, and by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114.

1 Introduction

This paper explains the design and use of two proving tools, namely an inductive theorem prover—to prove theorems about equational specifications with an initial algebra semantics—and a Church-Rosser checker—to check whether such specifications satisfy the Church-Rosser property. These tools have been developed as part of the Cafe project [6, 20], and have been integrated, thanks to the efforts of our colleagues in Japan, within the overall Cafe environment. Through that integration, they can be used to prove formal properties of order-sorted equational specifications in CafeOBJ [17]. They can also be used on their own to prove properties of equational specifications in Maude [9]; that is, of its so-called functional modules, that are equational specifications in membership equational logic. An important feature of these tools is that they are written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement.

Both tools treat equational specifications as *data* that is manipulated. For example, the inductive theorem prover may add an induction hypothesis as a new equational axiom; and the Church-Rosser checker computes critical pairs and membership assertions by inspecting the equations in the original specification. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. Indeed, the fact that rewriting logic is a reflective logic [7, 14], and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tools [8, 11].

Rewriting logic reflection has one important additional advantage, namely, the modular and declarative treatment of *proof tactics* or *strategies* [14, 7]. Such tactics can be formally specified by rewrite rules at the metalevel of the theory expressing the tool's inference system, and can be easily changed at will without any modification whatsoever to the inference system itself.

The very high level of abstraction at which the tools have been developed has made it relatively easy for us to build them, makes understanding their implementation much easier, and will make their maintenance and future extensions much simpler than if a conventional implementation, say in C, C++, or Java, had instead been chosen. Thanks to the high performance of the Maude engine [9]—that can reach 1.66 million free-theory rewrites per second, and from 130 thousand to one million associative-commutative rewrites per second for some applications running on a 500 MHz Alpha—these important benefits in ease of development, understandability, maintainability, extensibility, and in flexibility for introducing formally-defined proof tactics, are achieved without sacrificing performance. Even though they have not been optimized for performance, and in spite of using several levels of reflection and sophisticated rewriting modulo associativity and associativity-commutativity, the tools have quite competitive performance. For example, the inductive theorem prover is very responsive in interactive mode, with input/output time typically dominating over computation time. The Church-Rosser checker also offers reasonable performance. For example, generating all the critical pairs and membership assertions for the number hierarchy from the natural to the rational numbers takes 2,091,898 rewrites, which are performed in 12 seconds running on a 450MHz Pentium II.

The rest of the paper is as follows. We first give a brief review of membership equational logic, rewriting logic, and Maude, including reflective features and the related topic of strategies; and we summarize the reflective design of the tools. Then, each of the tools, including its inference system and its corresponding Maude implementation, is explained and illustrated with examples in a separate section. We end the paper with some concluding remarks.

2 Basic Concepts and Tool Design

2.1 Membership Equational Logic

Membership equational logic [37, 3], is conservative extension of both order-sorted equational logic and partial equational logic with existence equations [37]. It supports partiality, subsorts, operator overloading, and error specification.

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, (K, Σ) a many-sorted (although it is better to say "many-kinded") signature, and $S = \{S_k\}_{k \in K}$ a K-kinded set of *sorts*.

An Ω-*algebra* is then a (K, Σ)-algebra A together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or nonerror, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulae, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ \ t = t\ \ \mathit{if}\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *membership axioms* of the form

$$(\forall X)\ \ t : s\ \ \mathit{if}\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic has all the usual good properties: soundness and completeness of appropriate rules of deduction, initial and free algebras, relatively free algebras along theory morphisms, and so on [37].

2.2 Rewriting Logic

Rewriting logic [32] is like a two-faced Janus, in that it expresses an essential equivalence between logic and computation in a particularly simple way. Namely, system *states* are in bijective correspondence with *formulas* (modulo whatever structural axioms are satisfied by such formulas: for example, modulo the associativity and commutativity of a certain connective) and concurrent *computations* in a system are in bijective correspondence with *proofs* (modulo appropriate notions of equivalence between computations and between proofs). Given this equivalence between computation and logic, a rewriting logic axiom of the form $t \longrightarrow t'$ has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can *change* to the corresponding instance of t' concurrently with any other state changes; that is, the computational reading is that of a *local concurrent transition*. Logically, it just means that we can derive the formula t' from the formula t; that is, the logical reading is that of an *inference rule*.

Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user-definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place *modulo* such axioms. Because of this ecumenical neutrality, rewriting logic has, from a logical viewpoint, good properties as a *logical framework* [31], in which many other logics can be naturally represented. And, computationally, it has also good properties as a *semantic framework* [35], in which many different system styles and models of concurrent computation can be naturally expressed without any distorting encodings.

The design of the present tools exploits and illustrates the logical framework capabilities of rewriting logic by formally specifying the *inference system* of each tool as a rewrite theory. The fact that these tools reason about *equational theories* is not a restriction of the general framework capabilities: inference systems for reasoning about specifications in any other logic could be represented just as easily. However, the fact that rewriting logic contains equational logic as a sublanguage and that Maude has good support for its equational sublanguage is an added advantage.

We can represent a *rewrite theory* as a four-tuple $T = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic[1] [37, 3], L is a set of labels, to label the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional [32].

2.3 Maude

Maude [9, 12] is a language whose modules are theories in rewriting logic. The most general Maude modules are called *system modules*. They have the syntax `mod` T `endm` with T the rewrite theory in question, expressed with a syntax quite close to the corresponding mathematical

notation[2]. The equations E in the equational theory (Ω, E) underlying the rewrite theory $T = (\Omega, E, L, R)$ are presented as a union $E = A \cup E'$, with A a set of *equational axioms* introduced as *attributes* of certain operators in the signature Ω—for example, an operator $+$ can be declared associative and commutative by keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating *modulo* the axioms A. Maude supports rewriting modulo different combinations of such equational attributes: operators can be declared associative, commutative, with identity, and idempotent [9]. Maude contains a sublanguage of *functional modules* of the form `fmod` (Ω, E) `endfm`, where, as before, $E = A \cup E'$, with E' Church-Rosser and terminating modulo A. A system module `mod` T `endm` specifies the initial model of the rewrite theory T. Similarly, a functional module `fmod` (Ω, E) `endfm` specifies the initial algebra of the equational

[1]More generally, the choice of the equational logic underlying rewriting logic is a parameter that can be instantiated to different variants.

[2]See [9] for a detailed description of Maude's syntax, which is quite similar to that of OBJ3 [25]. In this paper we adopt an optimistic viewpoint, assuming that the syntax of the modules that we present is for the most part self-explanatory.

theory (Ω, E), which is at the same time the initial model of (Ω, E) when viewed as a (degenerate) rewrite theory.

The syntax of Maude modules follows quite closely the mathematical notation for specifying the corresponding mathematical theories. For example, the following module `NAT` specifies the natural numbers in Peano notation.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> Nat [ctor] .
  ops (_+_) (_*_) : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
  eq 0 * N = 0 .
  eq s N * M = M + (N * M) .
endfm
```

Note that the addition and multiplication operators have been declared commutative with the `comm` equational attribute. The `ctor` attribute declares zero and successor as *constructors*, stating that all other ground terms can be proved equal to terms built using only these two operators.

2.4 Reflection and the `META-LEVEL`

Rewriting logic is reflective [13, 14] in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T (including U itself) we have the following equivalence

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

where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U. Since U is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$T \vdash t \rightarrow t' \Leftrightarrow U \vdash \langle \overline{T}, \overline{t} \rangle \rightarrow \langle \overline{T}, \overline{t'} \rangle \Leftrightarrow U \vdash \langle \overline{U}, \overline{\langle \overline{T}, \overline{t} \rangle} \rangle \rightarrow \langle \overline{U}, \overline{\langle \overline{T}, \overline{t'} \rangle} \rangle \ldots$$

For efficiency reasons, the Maude implementation provides key features of the universal theory U in a built-in module called `META-LEVEL`. In particular, META-LEVEL has sorts Term and Module, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and a module T have sorts Term and Module, respectively.

Rather than giving here a detailed explanation of the representations $\overline{t}$ and $\overline{T}$ of terms and modules—for which we refer the reader to [8]—we instead illustrate them with simple examples. In the representation of terms, constants are represented as pairs using the operator {_}_, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant 0 in the module NAT in Section 2.3 is represented as {'0}'Nat. The operator _[_] corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form and the second argument the list of its subterms separated by commas. For example, the term s s 0 + s 0 in the module NAT is metarepresented as

```
'_+_['s_['s_[{'0}'Nat]], 's_[{'0}'Nat]].
```

The representation of modules follows very closely Maude's syntax, except that terms in equations, memberships, and rules, are now metarepresented as explained above. For example, the representation $\overline{\texttt{NAT}}$ of the module NAT is the following term of sort Module:

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [ctor] .
  op '_*_ : 'Nat 'Nat -> 'Nat [ctor] .
  var 'N : 'Nat . var 'M : 'Nat .
  none
  eq '_+_[{'0}'Nat, 'N] = 'N .
  eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
  eq '_*_[{'0}'Nat, 'N] = {'0}'Nat .
  eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N, 'M]] .
endfm
```

META-LEVEL has also a number of important metalevel functions [9]. For our purposes here we focus on the functions meta-reduce and meta-apply.

The function `meta-reduce` takes as arguments the representation of a module T and the representation of a term t or of a membership assertion $t : s$ in that module. When the second argument is the representation $\overline{t}$ of a term t in T, the function `meta-reduce` returns the representation of the fully reduced form of the term t using the equations in T. When the second argument of `meta-reduce` is the representation of a membership assertion $t : s$, the term t is fully reduced using the equations in T and then the representation of the value of the resulting membership predicate is returned.

The operation `meta-apply` has the form:

`meta-apply(`T`,`t`,`l`,`σ`,`n`)`. Its first four arguments are representations in `META-LEVEL` of a module T, a term t in T, a label l of some rule in T, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n. `meta-apply` is evaluated as follows:

1. the term t is first fully reduced using the equations in T;
2. the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;
3. the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise `{error*, none}` is returned;
4. the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in T;
5. the pair formed using the constructor `{_,_}` whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

2.5 Strategies

System modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. We need then to have flexible ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate *strategies*. In Maude, thanks to its reflective capabilities, strategies can be made *internal* to the logic. That is, they can be defined by rewrite rules in a normal module in Maude, and can be reasoned about as with rules in any other module.

In fact, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. The idea is to use the operations `meta-reduce` and `meta-apply` as basic strategy expressions, and then to extend the module `META-LEVEL` by additional strategy expressions and corresponding semantic rules. That is, `meta-reduce` and `meta-apply` allows us to take *elementary steps* in the deduction process, but we can take *bigger steps* by defining more complex *strategy expressions* in a module extending `META-LEVEL` by *equations* that define in rewriting logic the precise rewriting semantics for each of the constructors building up strategy expressions in our strategy language of choice. A number of strategy languages that have been defined following this methodology can be found in [7, 9].

2.6 Reflective Design of the Tools

Based on these concepts, each of the tools has then a very simple design. Consider, for example, the design of the inductive theorem prover. Its purpose is to prove inductive properties of a functional module T in Cafe or Maude, which has an initial algebra semantics. The theory T about which we want to prove inductive properties is at the object level. The rules of inference for induction can be naturally expressed as a rewrite theory $\mathcal{I}$. But since this rewrite theory uses T as a data structure—that is, it actually uses its representation $\overline{T}$—the theory $\mathcal{I}$ should be defined at the metalevel. Proving an inductive theorem for T corresponds to applying the rules in $\mathcal{I}$ with some *strategy*. But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{I}$ is already at the metalevel, we need *two levels of reflection* above the object level to clearly distinguish levels and make our design entirely *modular*, so that, for example, we can change the strategy without any change whatsoever to the inference rules in $\mathcal{I}$. This is illustrated by the picture in Figure 1.

The inductive Church-Rosser checker tool has a similar reflective design. Again, the module T that we want to check is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses $\overline{T}$ as a data structure and therefore is a rewrite theory at the metalevel. Since the checking process

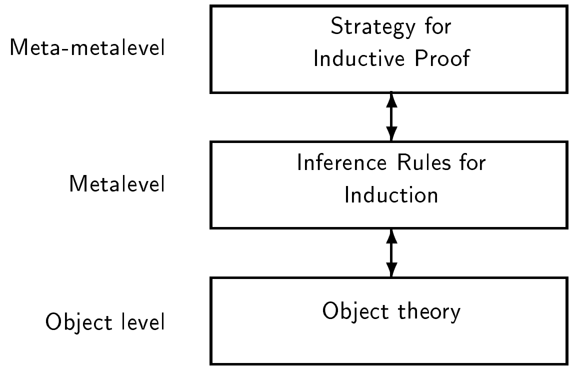

Figure 1: The design of the inductive theorem prover.

can be described in a purely functional way, there is no need in this case for an additional strategy layer at the meta-metalevel: one reflective level suffices.

The reflective design of the tools includes also the design in Maude of a user interface for each tool. For example, the fact that two levels of reflection are used in the inductive theorem prover should be hidden from the user, so that he/she can interact with the prover by entering modules and giving commands at the object level and can get appropriate results and feedback from the tool also at that level. Suitable user interfaces for each tool that accomplish this have been built in Maude using the meta-parsing and meta-pretty-printing functions in `META-LEVEL`, together with a simple `LOOP-MODE` module in Maude [9] that provides input/output. Using these facilities, the interfaces are then built for each tool by specifying in Maude with rewrite rules the desired behavior in response to the different commands that can be entered by the user.

Our approach for building these interfaces has been to integrate the tools within the Full Maude environment [9, 18], a language extension of Maude written in Maude that supports parameterized modules, parameter theories, views, and module expressions, and that already has a user interface built using reflective methods. In this way, we have been able to reuse much of the functionality required for the tools' interfaces. Furthermore, in this way we can integrate Full Maude with the tools and submit to them modules in Full Maude's database. We do not give details about the implementation of the interfaces, nor about the integration of the tools with Full Maude. Complete explanations about how this has been done for the Church-Rosser checker can be found in [18].

3 The Inductive Theorem Prover

In this section we present the specification in Maude of an inductive theorem prover to prove theorems about the initial model of a functional module in Cafe or in Maude[3]. As in the case of the Church-Rosser checker tool that will be discussed in Section 4, it is very natural to adopt a reflective design to specify in Maude an inductive theorem-prover. The idea is that the functional module T about which we want to prove inductive theorems is at the object level; an inference system $\mathcal{I}$ for inductive proofs uses $\overline{T}$, the metarepresentation of module T, as data, and therefore can be specified as a rewrite theory at the metalevel; then, different *proof tactics* to guide the application of the rewrite rules that specify the inference rules in $\mathcal{I}$ are strategies that can be represented at the meta-metalevel.

We first introduce an inference system for inductive proofs, and we explain how it can be specified in Maude as a rewrite theory at the metalevel. Then, we discuss the representation of proof tactics at the meta-metalevel and illustrate their use. We also give an inductive proof method to ensure that a set of constructors generate the initial algebra of the specification—so that only terms involving such constructors need to be included in the *test sets* used in the induction scheme—and illustrate this proof method with examples. We also discuss the generation of the test sets themselves.

3.1 Inductive Inference Rules

The following rules 1–6 provide a sound inference system for proving that a universally quantified atomic formula of the form $(\forall X)t = t'$, or of the form $(\forall X)t : s$ is an inductive consequence of a given membership algebra specification. Note the "backwards reasoning" form of these rules. That is, in each rule the inference above the bar is in fact the conclusion that can be

[3]Cafe functional modules are order-sorted equational theories; Maude functional modules are theories in membership equational logic. Thanks to the conservative embedding of order-sorted equational logic into membership equational logic [37], Cafe modules are easily translated into corresponding Maude modules.

established if we can prove the "subgoals" under the bar. This presentation of the rules agrees with the intended use of the inductive prover, in which a user will submit an atomic sentence to be proved as the initial goal, and then this goal will be successively transformed by rewriting—using the inference rules 1–6 as rewrite rules—into different sets of subgoals, until no subgoals are left if the proof succeeds.

Note also that the atomic sentences that these rules can attempt to prove are precisely those of membership equational logic. Since a very general version of order-sorted algebra can be embedded in membership algebra in a conservative way [37], they remain sound for proving inductive theorems of order-sorted functional specifications in Cafe. In fact, the extra generality of membership equational logic allows us to prove useful properties of order-sorted specifications that cannot be stated as order-sorted sentences. Two good examples are sufficient generation proofs—that are expressed as inductive proofs of membership assertions for the defined symbols (see Section 3.3)—and inductive proofs of membership assertions generated as proof obligations by the Church-Rosser Checker to ensure the ground-Church-Rosser property for a given order-sorted specification (see Section 4.1.2).

In what follows, $vars(t)$ denotes the set of variables that occur in a term t.

1. **Test Set Induction**

$$\frac{M \vdash_{\text{ind}} (\forall X).p}{M \vdash_{\text{ind}} \mathit{step}(p, X \setminus \{x\}, x, s, u_1) \cdots M \vdash_{\text{ind}} \mathit{step}(p, X \setminus \{x\}, x, s, u_n)}$$

where $(\forall X).p$ is an atomic sentence in membership equational logic, $x \in X$ has sort s and $\{u_1, \ldots, u_n\}$ is the test set for sort s and

$$\begin{aligned} &\mathit{step}(p, X, x, s, u) = \\ &\qquad (\forall vars(u)).\left(\left(\bigwedge_{y:s' \in vars(u),\ s' \leq s} (\forall X).p[y/x]\right) \Rightarrow (\forall X).p[u/x]\right) \end{aligned}$$

2. **Proof in Variety**

$$\frac{M \vdash_{\text{ind}} p}{M \vdash p},$$

where p is a sentence in membership equational logic.

3. **Constants Lemma**

$$\frac{M \vdash (\forall\{x_1, \ldots, x_n\}).p}{M \cup \{\texttt{op}\ c_1 \,\texttt{:->}\ s_1 . \cdots \texttt{op}\ c_n \,\texttt{:->}\ s_n .\} \vdash p[c_1/x_1, \ldots, c_n/x_n]}$$

where $\forall\{x_1, \ldots, x_n\}).p$ is a sentence in membership equational logic, x_i has sort s_i, for $1 \leq i \leq n$, and $c_1, \ldots, c_n$ do not occur in M.

4. **Implication Elimination**

$$\frac{M \vdash (\bigwedge_i (\forall X_i).u_i = v_i) \wedge (\bigwedge_j (\forall X_j).w_j : s_j) \Rightarrow (\forall X).t = t'}{M \cup \{\texttt{eq}\ u_i = v_i .\}_i \cup \{\texttt{mb}\ w_j : s_j .\}_j \vdash (\forall X).t = t'}$$

$$\frac{M \vdash (\bigwedge_i (\forall X_i).u_i = v_i) \wedge (\bigwedge_j (\forall X_j).w_j : s_j) \Rightarrow (\forall X).t : s}{M \cup \{\texttt{eq}\ u_i = v_i .\}_i \cup \{\texttt{mb}\ w_j : s_j .\}_j \vdash (\forall X).t : s}$$

where in both rules we assume that for each $i \in \{1, \ldots, n\}$, $(vars(u_i) \cup vars(v_i)) \subseteq X_i$, and for each $j \in \{1, \ldots, m\}$, $vars(w_j) \subseteq X_j$.

5. **Proof by Rewriting for Ground Atoms**

$$\frac{M \vdash t = t'}{t \downarrow_M \equiv t' \downarrow_M} \quad \text{if } t,\ t' \text{ are ground.}$$

$$\frac{M \vdash t : s}{t \downarrow_M : s} \quad \text{if } t \text{ is ground.}$$

6. **Equational and Membership Lemmas**

$$\frac{M \vdash_{\text{ind}} p}{M \vdash_{\text{ind}} (\forall X).t = t' \qquad M \cup \{\texttt{eq}\ t = t' .\} \vdash_{\text{ind}} p}$$

$$\frac{M \vdash_{\text{ind}} p}{M \vdash_{\text{ind}} (\forall X).t : s \qquad M \cup \{\texttt{mb}\ t : s .\} \vdash_{\text{ind}} p}$$

where $(\forall X).t = t'$, $(\forall X).t : s$, and p are atomic sentences in membership equational logic.

The explicit induction style of the above rules of inference is similar to that used in other explicit induction approaches in equational theorem proving such as those of OBJ [22, 23] and LARCH [26].

We give now an overview of the specification in Maude of the inference system just introduced. This specification is given by a module called `ITP`, in which membership equational formulas α are represented as terms $\overline{\alpha}^f$ using the operations `trueFormula`,`equality`, `sortP`, `implication`, `conjunction`, and `VQuantification`. We illustrate this representation using the module `NAT` in Section 2.3. For example, the universally quantified atomic formula $(\forall\{\mathtt{N},\mathtt{M}\}).\mathtt{N + M} = \mathtt{M + N}$ is represented in `ITP` by the term

$$\mathtt{VQuantification((}\overline{\mathtt{N}}\ \mathtt{;}\ \overline{\mathtt{M}}\mathtt{),\ equality(}\overline{\mathtt{N + M}}\mathtt{,}\ \overline{\mathtt{M + N}}\mathtt{)).}$$

Similarly, $(\forall\{\mathtt{N},\mathtt{M}\}).\mathtt{N + M} : \mathtt{Nat}$ is represented by the term

$$\mathtt{VQuantification((}\overline{\mathtt{N}}\ \mathtt{;}\ \overline{\mathtt{M}}\mathtt{),\ sortP(}\overline{\mathtt{N + M}}\mathtt{,}\ \overline{\mathtt{Nat}}\mathtt{).}$$

Sets of terms are built with the constructor `termSet`, with `emptyTermSet` the empty set of terms. The constructor `test` is used to represent the test set for a given sort. A set of test sets is built with the constructor `testSet`, with `emptyTestSet` the empty set of test sets. For example, the term $\mathtt{test(}\overline{\mathtt{Nat}}\mathtt{,\ termSet(}\overline{\mathtt{0}}\mathtt{,}\ \overline{\mathtt{s\ N}}\mathtt{))}$ is the representation of the test set[4] for the sort `Nat` in the module `NAT` in Section 2.3.

The (sub)goals for our inductive theorem prover are represented with the constructors `proveinInitial` and `proveinVariety`, for proofs in the initial model and proofs in the variety, respectively. For example, the goal

$$\mathtt{NAT} \vdash_{\text{ind}} (\forall\{\mathtt{N},\mathtt{M}\}).\mathtt{N + M} = \mathtt{M + N}$$

is represented in `ITP` by the term

$$\mathtt{proveinInitial(}I\mathtt{,}\ \overline{\mathtt{NAT}}\mathtt{,}\ \overline{(\forall\{\mathtt{N},\mathtt{M}\}).\mathtt{N + M} = \mathtt{M + N}}^f\mathtt{,}\ \overline{\mathtt{NATTS}}\mathtt{),}$$

where I is a string of positive numbers, and $\overline{\mathtt{NATTS}}$ is the representation of the set of test sets for the sorts in the functional module `NAT`. The strings of

[4] The correctness of a test set for a given sort depends on having shown that the subsignature of constructors on which the test set is based does indeed generate all terms of that sort, in the sense that any term of that sort is provably equal to a constructor term. As explained in Section 3.3, given a subsignature of constructors for our specification we can use the inductive theorem prover to show that indeed the constructors generate all terms of all sorts, and then we can mechanically check or generate test sets for each sort. In what follows we shall assume that this has already been done for the specification in question.

positive numbers are used to number the (sub)goals in a proof. Finally, sets of (sub)goals are built with the constructor `goalSet`, with `emptyGoalSet` the empty set of goals.

The rewrite rules in the module `ITP` mirror very closely the inference system described above. Here we only explain the specification of the inference rules **Implication Elimination** and **Proof by Rewriting for Ground Atoms**. When explaining the rules in `ITP` we also explain briefly the auxiliary functions used in them. Note that our specification makes use of the reflective capabilities of Maude, in particular: (i) that functional modules can be treated as data that can be manipulated; and (ii) that reducing a term to its normal form is reified by the built-in function `meta-reduce`. Full details about the specification of the `ITP` tool can be found in [10].

We start with the declaration of some variables that will be used in the rules:

```
var IS : IntString .   vars T1 T2 : Term .  var Th : FModule .
vars Alpha Beta : Formula .  var G : GoalSet .
```

The inference rules for **Implication Elimination** are jointly specified by the rule `implicationElimination`.

```
op addToModule : Formula Module -> Module .

rl [implicationElimination]:
  goalSet(proveinVariety(IS, Th, implication(Alpha, Beta)), G)
  => goalSet(proveinVariety(IS, addToModule(Alpha, Th), Beta), G) .
```

Let T be a functional module, and $\alpha = (\forall X_1).p_1 \wedge \ldots \wedge (\forall X_n).p_n$ a sentence over the signature of T such that, for $1 \leq i \leq n$, p_i is either $u_i = v_i$ or $w_i : s_i$. Then, the function `addToModule(`$\overline{\alpha}^f$`, `$\overline{T}$`)` returns $\overline{T'}$, where T' is the module T extended in the following way: for each component $(\forall X_i).u_i = v_i$ in the conjunction α the equation `eq` $u_i = v_i$ is added to the set of equations of T; and for each component $(\forall X_i).w_i : s_i$ in the conjunction α the membership axiom `mb` $w_i : s_i$ is added to the set of membership axioms of T.

The inference rules for **Proof by Rewriting for Ground Atoms** are specified by the rules `byRewriting`. Notice the use of the built-in function `meta-reduce` in our specification of these inference rules.

```
rl [byRewriting]:
```

```
    goalSet(proveinVariety(IS, Th, equality(T1, T2)), G)
    => goalSet(proveinVariety(IS, Th,
                equal(meta-reduce(Th, T1), meta-reduce(Th, T2))), G) .
  rl [byRewriting]:
     goalSet(proveinVariety(IS, Th, sortP(T1, S)), G)
    => goalSet(
          proveinVariety(IS, Th,
             equal(meta-reduce(Th, T1 : S), {'true}'Bool)), G) .

eq proveinVariety(IS, Th, equal(T, T)) = emptyGoalSet .
```

3.2 Proof Strategies

As explained above, different *proof tactics* to guide the applications of the rewrite rules in the module `ITP` are strategies that can be represented at the meta-metalevel.

In [10], two proof tactics are defined in detail: a strategy `simplify` that tries to simplify a particular (sub)goal as much as possible, and a strategy `induction` that, when there is only one goal to be proved, applies the induction rule on a selected variable and then simplifies the resulting subgoal as much as possible. These strategies are computed by a function `rewInWith` that takes three arguments: the representations of a module T, the representation of a term t, and the strategy to be computed. The definition of `rewInWith` is such that, as the computation of a given strategy proceeds, t gets rewritten by controlled application of rules in T, and the strategy S is rewritten into the remaining strategy to be computed; in case of termination, this is the `idle` strategy.

As an example, we can use the strategy `induction` to prove the associativity of the operation `+` (addition) as defined in the module `NAT` in Section 2.3, that is, the formula $(\forall\{\mathtt{X},\mathtt{Y},\mathtt{Z}\}).\mathtt{X + (Y + Z)} = \mathtt{(X + Y) + Z}$. We prove it computing the `induction` strategy on the variable `Z`, that is, the term

$$\texttt{rewInWith(}\overline{\texttt{ITP}}\texttt{,}\ \overline{\texttt{proveinInitial(1, NAT, }\overline{(\forall\{\texttt{X,Y,Z}\}).\texttt{X+(Y+Z)} = \texttt{(X+Y)+Z}}^{f}\texttt{, NATTS), induction(}\overline{\overline{\texttt{Z}}}\texttt{)}}\texttt{) .}$$

which is reduced to

$$\texttt{rewInWith(}\overline{\texttt{ITP}}\texttt{, }\overline{\texttt{emptyGoalSet}}\texttt{, idle) .}$$

3.3 Sufficient Generation

The test set induction rule instantiates an induction variable by the terms in a test set. The most classical case is the instantiation of a variable of sort `Nat` by `0` and `s N`. These expressions suffice, because zero and successor are the *constructors* that build up all natural numbers. In general, however, how can we justify the correctness of a test set? We need to show that all the ground terms of a given sort are provably equal to constructor terms that are instances of the terms in the test set for that sort. This brings us to the topic of sufficient generation, that is, of showing that a given constructor subspecification—that may actually satisfy some equations of its own—generates all the sorts in the initial algebra of our specification, in the sense that all terms of that sort can be proved equal to constructor terms. We give here a general method of using the inductive theorem prover to prove sufficient generation, and illustrate the method with some examples; then we discuss a simple test set generation scheme. Our method is based on a theorem about protecting extensions of membership equational logic specifications by Bouhoula, Jouannaud, and Meseguer [3]. For the sake of a simpler exposition, we deal here with the case of order-sorted specifications. The more general case of membership equational logic specifications has a similar treatment and will be discussed elsewhere.

Sufficient generation is a property somewhat weaker than sufficient completeness. Given an order-sorted specification $T = (S, \leq, \Sigma, \Gamma)$ and a subspecification of constructors $T_0 = (S, \leq, \Omega, \Gamma_0)$, with $T_0 \subseteq T$, we say that the initial algebra of T is *sufficiently complete* relative to T_0 if the unique Ω-homomorphism

$$\mathcal{T}_{T_0} \longrightarrow \mathcal{T}_T|_\Omega,$$

from the initial algebra of T_0 to the Ω-algebra obtained from the initial algebra of $\mathcal{T}_T$ by forgetting the operations in $\Sigma - \Omega$, is an *isomorphism*. Instead, we say that the initial algebra of T is *sufficiently generated* by the subsignature of constructors Ω, if the unique Ω-homomorphism

$$\mathcal{T}_\Omega \longrightarrow \mathcal{T}_T|_\Omega$$

is surjective for each sort $s \in S$. This is equivalent to requiring that for each $s \in S$ and each $t \in \mathcal{T}_{\Sigma,s}$ there is a $t_0 \in \mathcal{T}_{\Omega,s}$ with

$$T \vdash t = t_0.$$

Sufficient completeness is of course a stronger property than sufficient generation. But for being justified in using only constructor terms in the test sets of an inductive proof, only sufficient generation must be checked.

We give below a general method for checking that the initial algebra of T is sufficiently generated by a subsignature of constructors Ω. The advantage of this method is that we can use the inductive theorem prover to check the proof obligations that it generates.

The method is as follows. Given T and a subsignature of constructors Ω, we define a new specification $E_\Omega T = (ES, \leq_E, E\Sigma, \Gamma)$ with:

- sorts ES those of S plus, for each connected component C of $(S, \leq)$ a new "error sort" E_C greater than all the sorts in C,
- order $\leq_E$ the obvious extension of $\leq$ to make each E_C the top of each component,
- operators $E\Sigma$ those of Ω (with same arity and coarity), plus for each operator $f : s_1 \dots s_n \to s$ in $\Sigma - \Omega$ with $c(s_1), \dots, c(s_n), c(s)$ the corresponding connected components, an operator $f : E_{c(s_1)} \dots E_{c(s_n)} \to E_{c(s)}$, and
- axioms Γ exactly the same as those of T.

Then, to prove that the initial algebra of T is sufficiently generated by Ω, it is enough to show that for each[5] $f : s_1 \dots s_n \to s$ in $\Sigma - \Omega$,

$$E_\Omega T \vdash_{\text{ind}} f(x_1, \dots, x_n) : s \qquad (\ddagger)$$

where the variables $x_1, \dots, x_n$ have respective sorts $s_1, \dots, s_n$. Note that this inductive requirement takes place in membership equational logic. Therefore, we implicitly use the conservative embedding of order-sorted equational logic into membership equational logic [37], and the fact that Maude's functional modules are theories in membership equational logic.

We can then use the inductive theorem prover to prove each of the proof obligations ($\ddagger$). Note that, since the specification $E_\Omega T$ is of course sufficiently generated by its own signature $E\Sigma$, and $E\Sigma$ has the property that for each $s \in S$ all operators of coarity less or equal to s are in Ω, when proving the proof obligations ($\ddagger$), we are justified in using only Ω-*terms* in the test sets of sorts $s \in S$ in $E_\Omega T$.

[5]Note that an operator declaration $f : s_1 \dots s_n \to s$ can be in $\Sigma - \Omega$ while at the same time another declaration $f : s'_1 \dots s'_n \to s'$ can be in Ω. For example, `s : Int -> Int` is not a constructor, but `s : Nat -> Nat` is a constructor.

3.3.1 Test Set Generation

We illustrate the use of the general methodology to prove sufficient generation below. However, since for proving inductively sufficient generation we need a test set, we explain now a simple test set generation scheme.

Once we have proved that a subsignature of constructors sufficiently generates the initial algebra of a specification, we are justified in using only those constructors to associate to each sort a test set. Of course, the choice of what terms to use for a test set is not unique. For example, in a specification of the natural numbers with a sort `NzNat` of nonzero natural numbers and constructors `0 : -> Nat`, and `s : Nat -> NzNat`, we could use as test set for the sort `NzNat` the term `s N` with `N` a variable of sort `Nat`, but we could also have used as test set of sort `NzNat` the terms `s 0` and `s s N`, with `N` a variable of sort `Nat` as well.

Our test set generation function chooses the first, simpler alternative. That is, it always yields terms of depth zero or one in the test set of each sort. Such terms are obtained by inspecting what constructors have sort smaller or equal to the sort in question; for each such constructor $\mathtt{f} : \mathtt{s}_1 \ldots \mathtt{s}_n \to \mathtt{s}$ a term $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ with $\mathtt{x}_i$: $\mathtt{s}_i$, for $1 \leq i \leq n$, is then added to the test set. As an optimization that avoids unnecessary subgoals in inductive proofs, if a constructor has two overloadings, with the rank of one of them smaller or equal to that of the other, and both with coarities smaller or equal to the sort in question, then only the most general constructor is used in generating the test set for that sort.

Given a subsignature of constructors for which sufficient generation has been proved, the function `genTestSets` takes a module[6] and generates the test sets for each sort in it.

The function that generates the test sets is `genTestSets`, which takes a module and returns a pair of sort `TestSetSol` consisting of a module and the family of test sets generated.

```
op genTestSets : Module -> TestSetSol .
```

It may be that in the process of generating the test sets new variables have to be added to the original specification. Therefore, the test sets generated have to be used together with this new module.

[6] As said in Section 2.3, in Maude 1.0.5 constructor opertors can be distinguished using the `ctor` attribute.

3.3.2 Proving Sufficient Generation of Natural Numbers

We shall prove that the constructors for the module NAT presented in Section 2.3 are 0 and s_. To do this we create a module ENAT as a copy of NAT in which we introduce a new sort ErrorNat as supersort of Nat, and move up to this new sort all the defined operations, except those considered as candidates for constructors, that is, we declare operators _+_ and _*_ as

```
ops (_+_) (_*_) : ErrorNat ErrorNat -> ErrorNat [comm] .
```

and leave unchanged the declarations for 0 and s_.

The first thing we need in order to be able to use the theorem prover is to obtain a test set for ENAT. What we can do is to consider all the operators of ENAT as constructors, something we are always justified in doing, since the entire operator set obviously generates the initial algebra. For this purpose we call the function genTestSets presented above with a metalevel syntactic variant CNAT of the module ENAT in which all the operators are syntactically marked as constructors

in

Then, calling the function genTestSets with CNAT as argument gives us a pair consisting of a modified module, in which some variables may have been introduced, and the corresponding family of test sets. Transforming back the module returned with the test sets to the original module syntax without constructor declarations we obtain a module which, in order to continue using the same notation introduced above, we shall call ENAT.

Let us call ENATTS the test sets returned:

```
testSet(test(Zero, 0),
        test(Nat, termSet(0, s N)),
        test(ErrorNat, termSet(0, s N, ErrorNat&'0 + ErrorNat&'1,
                               ErrorNat&'0 * ErrorNat&'1)))
```

Now we can prove the proof obligations for sufficient generation by the method already explained. To prove that the operators s_ and 0 are constructors for NAT, we need to prove inductively that in the module ENAT, given variables N and M of sort Nat, N + M and N * M are of sort Nat. A detailed description of the proofs using the inductive theorem prover presented above and some other examples can be found in [10].

4 The Church-Rosser Checker

The goal of *executable* equational specification languages is to make *computable* the abstract data types specified in them by initial algebra semantics. In practice this is accomplished by using specifications that are *ground*-Church-Rosser and terminating, so that the equations can be used from left to right as simplification rules. This approach is fully general; indeed, a well-known result of Bergstra and Tucker [2] shows that *any* computable algebraic data type can be specified by means of a finite set of ground-Church-Rosser and terminating equations, perhaps with the help of some auxiliary functions added to the original signature.

Therefore, for formal reasoning purposes it becomes very important to know whether a given specification is indeed ground-Church-Rosser and terminating. A nontrivial question is how to best support this with adequate tools. For proving termination one can use standard termination proof techniques that are well-supported in tools such as CiME [15]. A thornier issue is what to do for establishing the ground-Church-Rosser property for a terminating specification. The problem is that a specification with an initial algebra semantics can often be ground-Church-Rosser even though some of its critical pairs may not be joinable. That is, the specification can often be ground-Church-Rosser without being Church-Rosser for arbitrary terms with variables. In such a situation, blindly applying a completion procedure that is trying to establish the Church-Rosser property for arbitrary terms may be both quite hopeless—the procedure diverges or gets stuck because of unorientable rules, and even with success may return a specification that is quite different from the original one—and unnecessary: the specification was ground-Church-Rosser!

Our Church-Rosser checker tool is oriented to checking specifications with an initial algebra semantics that have already been proved terminating using a termination tool and now need to be checked to be ground-Church-Rosser. Since, for the reasons mentioned above, user interaction will typically be quite essential, completion is not attempted. Instead, if the specification cannot be shown to be ground-Church-Rosser by the tool, proof obligations are generated and are given back to the user as a useful guide in the attempt to establish the ground-Church-Rosser property. Since this property is in fact inductive, in some cases the inductive theorem prover can be enlisted to prove some of these proof obligations (see Section 4.3.2). In other cases, the user may in fact have to modify the original specification by carefully considering the information conveyed by the proof obligations. We give in Section 4.3 some methodological guidelines for the use of the tool, and illustrate the use with some examples; we also explain there that the issue of finding general inductive proof techniques for proving the ground-Church-Rosser property is at the moment an interesting open problem.

The present tool only accepts order-sorted conditional specifications such that each of the operation symbols has either no equational attributes, or only the commutativity attribute[7]. Furthermore, it is assumed that such specifications do not contain any built-in sort or function, and that they have already been proved terminating using another tool. The tool attempts to establish the Church-Rosser property *modulo* the commutativity of some of the operators by checking a sufficient condition. For order-sorted specifications, being Church-Rosser and terminating means not only confluence—so that a unique normal form will be reached—but also a *descent* property, namely that the normal form will have the least possible sort among those of all other equivalent terms. Therefore, the tool's output consists of a set of critical pairs and a set of membership assertions that must be shown, respectively, ground-joinable, and ground-rewritable to a term with the required sort.

After introducing some basic formal concepts and results underlying the tool's design, and after discussing several auxiliary functions used by the tool, including order-sorted commutative unification, we explain key parts of the tool and then give recommendations for its use and some examples. equations "..."

4.1 Church-Rosser Order-Sorted Specifications

In this section we introduce the notion of Church-Rosser order-sorted specification. We assume specifications of the form $\mathcal{S} = (\Sigma, R \cup A)$ where Σ is a preregular order-sorted signature [24], and $R \cup A$ is a set of equations such that A is a set of *sort-preserving* equational axioms, that is, whenever

[7] As we shall see in Section 4.2.1, in the current version of the tool the unification and matching algorithms are specified equationally, and only for signatures whose operators do not have equational attributes, or have just the commutativity attribute. Maude will have built-in matching and unification functions `meta-match` and `meta-unify` in the near future. Once they are available, the tool will be easily extensible to the case in which the corresponding combinations of equational attributes are supported.

$t =_A t'$ we have[8] $LS(t) = LS(t')$. The equations R will be used as rewrite rules modulo the axioms A. Furthermore, in what follows, and for the purposes of the present tool, the axioms A will only consist of *commutativity* axioms for certain binary operators in Σ. The problem, then, is to check whether our specification $\mathcal{S}$ has the Church-Rosser property.

After giving some auxiliary definitions, we introduce the notion of Church-Rosser order-sorted specifications, and describe the sufficient condition used by our tool to attempt checking the Church-Rosser property.

4.1.1 The Confluence Property

We shall use the standard notation for the positions of a term [1]; we denote *the subterm of* t *at position* p as $t|_p$ and a term t with its subterm $t|_p$ replaced by the term t' as $t[t']_p$. Recall that the set of variables occurring in a term t is denoted $vars(t)$. Then, given a substitution $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, we denote by $D(\sigma)$ the set of variables $\{x_1, \ldots, x_n\}$, and by $I(\sigma)$ the set $\bigcup_{i=1}^{n} vars(t_i)$.

We say that a term t *A-overlaps* another term with distinct variables t' if there is a nonvariable subterm $t'|_p$ of t' for some position p of t' such that the terms t and $t'|_p$ can be A-unified.

Definition 1 *Given an order-sorted equational specification* $\mathcal{S} = (\Sigma, R \cup A)$ *with the above assumptions, and given conditional rewrite rules* $l \rightarrow r$ `if` $\bar{u}\!\downarrow \bar{v}$ *and* $l' \rightarrow r'$ `if` $\bar{u'}\!\downarrow \bar{v'}$ *in* R *such that* $vars(l) \cap vars(l') = \emptyset$ *and* $l|_p\sigma =_A l'\sigma$*, for some nonvariable position* p *of* l *and* A*-unifier* σ*, then the triple*

$$ccp(l\sigma[r'\sigma]_p,\ r\sigma,\ \bar{u}\sigma\!\downarrow \bar{v}\sigma \wedge \bar{u'}\sigma\!\downarrow \bar{v'}\sigma)$$

is called a (conditional) critical pair.

We consider all unifiers, not only the most general ones. Note also that the critical pairs accumulate the substitution instances of the conditions in the two rules, as in [3].

Given a specification $\mathcal{S} = (\Sigma, R \cup A)$, a critical pair $ccp(t, t', c)$ is more general than another critical pair $ccp(u, u', c')$ if there exists a substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$, and $c\sigma =_A c'$.

Then, given a specification $\mathcal{S}$, let $\mathrm{MCP}(\mathcal{S})$ denote the set of most general critical pairs between rules in $\mathcal{S}$ that, after simplifying both sides of the

[8] We denote the least sort [24] of a term t by $LS(t)$.

critical pair using the equations in $\mathcal{S}$, are not identical critical pairs modulo A of the form $ccp(t, t, c)$. To determine the overlaps of the lefthand sides of the rules the variables in them have to be renamed in order to get disjoint sets of variables appearing in each of them. Under the assumption that the order-sorted equational specification $\mathcal{S}$ is terminating, then, if $\mathrm{MCP}(\mathcal{S}) = \emptyset$, we are guaranteed that the specification $\mathcal{S}$ is *confluent*, and therefore, each term t has a unique canonical form $t \downarrow_{\mathcal{S}}$.

4.1.2 The Descent Property

For an order-sorted specification it is not enough to be confluent. The canonical form should also provide the most complete information possible about the sort of a term. This intuition is captured by our notion of Church-Rosser specifications.

Definition 2 *We call a confluent and terminating order-sorted specification* $\mathcal{S}$ Church-Rosser *iff it additionally satisfies the following* descent *property: for each term* t *we have* $LS(t) \geq LS(t \downarrow_{\mathcal{S}})$. *Similarly, we call a ground-confluent and terminating specification* $\mathcal{S}$ ground-Church-Rosser *iff for each ground term* t *we have* $LS(t) \geq LS(t \downarrow_{\mathcal{S}})$.

Note that these notions are more general and flexible than the requirement of confluence and

sort-decreasingness [29, 21]. The issue is how to find checkable conditions for descent that, in addition to the computation of critical pairs, will ensure the Church-Rosser property. This leads us into the topic of specializations.

Given an order-sorted signature $(S, \leq, \Sigma)$, a sorted set of variables X can be viewed as a pair $(\bar{X}, \mu)$ where $\bar{X}$ is a set of variable names and μ is a sort assignment $\mu \colon \bar{X} \to S$. Thus, a *sort assignment* μ for X is a function mapping the names of the variables in $\bar{X}$ to their sorts. The ordering $\leq$ on S is extended to sort assignments by

$$\mu \leq \mu' \Leftrightarrow \forall x \in \bar{X}, \mu(x) \leq \mu'(x).$$

We then say that μ' *specializes* to μ, via the substitution

$$\rho : (x : \mu(x)) \leftarrow (x : \mu'(x))$$

called a *specialization* of $X = (\bar{X}, \mu')$ into $\rho(X) = (\bar{X}, \mu)$. Note that if the set of sorts is finite, or if each sort has only a finite number of sorts below it, then a finite sorted set of variables has a finite number of specializations.

The notion of specialization can be extended to axioms. A specialization of an equation $(\forall X, l = r)$ is another equation $(\forall \rho(X), \rho(l) = \rho(r))$ where ρ is a specialization of X. A specialization of a rule $(\forall X, l \to r$ `if` $c)$ is a rule $(\forall \rho(X), \rho(l) \to \rho(r)$ `if` $\rho(c))$ where ρ is a specialization of X.

The checkable conditions that we have to add to the critical pairs to test for the descent property are called membership assertions.

Definition 3 *Let $\mathcal{S}$ be an order-sorted specification whose signature satisfies the assumptions already mentioned. Then, the set of* (conditional) membership assertions *for a conditional equation $t = t'$* `if` *c is defined as the set of assertions $t'\theta : LS(t\theta)$* `if` *$c\theta$ such that θ is a specialization of $vars(t)$ and $LS(t'\theta \downarrow_{\mathcal{S}}) \not\leq LS(t\theta)$.*

A membership assertion $t : s$ `if` c is more general than another membership assertion $t' : s'$ `if` c' if there exists a substitution σ such that $t\sigma =_A t'$, $s \leq s'$, and $c\sigma =_A c'$.

A fundamental result[9] underlying our tool is that the absence of critical pairs and of membership assertions in such an output is a sufficient condition for a terminating specification $\mathcal{S}$ to be *Church-Rosser*. In fact, for terminating unconditional specifications this check is a necessary and sufficient condition; however, for conditional such specifications, the check is only a sufficient condition because if the specification has conditional equations we can have unsatisfiable conditions in the critical pairs or in the membership assertions; that is, we can have $\langle$ MCP($\mathcal{S}$), MMA($\mathcal{S}$) $\rangle \neq \langle \emptyset, \emptyset \rangle$ with $\mathcal{S}$ still Church-Rosser. Furthermore, even if we assume that the specification is unconditional, since for specifications with an initial algebra semantics we only need to check that $\mathcal{S}$ is ground-Church-Rosser, we may often have specifications that satisfy this property, but for which the tool returns nonempty sets of critical pairs or of membership assertions as proof obligations.

4.2 The Specification of a Church-Rosser Checker

The Church-Rosser checker tool is written entirely in Maude. It is in fact an executable specification in rewriting logic of the inference system to check a sufficient condition for the Church-Rosser property.

[9] A detailed proof of this result is beyond the scope of this work, and will be presented elsewhere. For related results in membership equational logic see [3].

The input to the Church-Rosser checker tool is given by the functional module we want to check. As we mentioned in Section 2.6, the tool has been completely integrated within the Full Maude environment [9, 18], so that the Church-Rosser property can be checked on any specification satisfying the conditions mentioned above that has been previously entered to Full Maude. Furthermore, the specification in question may be *structured*, with other specifications, perhaps defined by module expressions, as submodules.

After introducing order-sorted unification and matching, we explain the specifications of the confluence and descent checks. The specification of the checking of the Church-Rosser property in then presented.

4.2.1 Order-Sorted Commutative Unification and Matching

In this section we sketch the order-sorted unification algorithm we have implemented. It yields a complete set of unifiers for a unification problem in which it is assumed that the order-sorted signature of the specification in question can have some operators declared to be *commutative*—in which case, all the remaining subsort-overloaded versions of any such operator should also have been declared commutative—but no other equational axioms have been declared as attributes for the operators. Therefore, the order-sorted unification is performed *modulo* the commutativity of certain operators.

The basic idea is to turn each of the textbook-style inference rules for such a unification algorithm into corresponding rewrite rules in Maude. Figure 2 below gives the inference system; their Maude counterparts can be found in [10]. Here we only summarize the basic concepts and the top-level functionality.

Following [27], we unify using the sort information as soon as possible in order to quickly discard failures. Then, we complete the simplification process and push the assertions of the sorts on the solutions that have been found. A similar approach has been followed for order-sorted matching.

The order-sorted unification algorithm for solving equations (pairs of the form $t =^?_c t'$) can be described as the result of a simplification step followed by a solving step. The simplification step is a sequence of decomposition, merging, and mutation steps, transforming the initial unification problem into an equivalent disjunction of systems of fully decomposed equations of the form $x =^?_c t$, where x is a variable appearing only once in the system. The solving step consists in finding the finite set of solutions for the simplified system.

We give in Figure 2 the set of inference rules used for the unification algorithm. They operate on 3-tuples of the form $\langle V;\ E;\ \sigma\rangle$ and on 4-tuples of the form $[V;\ C;\ \sigma;\ \theta]$. A 3-tuple $\langle V;\ E;\ \sigma\rangle$ consists of a set of variables V, a set of equations E, and a substitution σ. A 4-tuple $[V;\ C;\ \sigma;\ \theta]$ consists of a set of variables V, a set of membership constraints C, and substitutions σ and θ. The rules operating on the first kind of tuples correspond to the first stage of the process, which is quite similar to syntactic unification. The main differences are in the rules **Check** and **Eliminate**, in which the sort information is used to try to quickly discard failure. In the second stage the constraints on the solutions are checked. We assume a well-founded order $\succ$ on equations, and we denote by $s \cap s'$ the set of maximal lower bounds of sorts s and s'.

The specification for order-sorted matching is quite similar to the specification for unification. The main differences come from the fact that now we have to consider noncommutative equations, since to find a match from a term t to another term t' is to find a substitution σ such that $t' =_c t\sigma$. The check for the correct sorting of an equation in solved form $x =_c^? t$ is reduced to comparing the sort of the variable x with the least sort of the term t being assigned.

4.2.2 Confluence Check

We present in this section the specification for the confluence check, carried out by the `conflCheck` function. We assume that the module passed as argument to `conflCheck` has been extended with renamed copies of the variables in it. Specifically, we assume that for each variable V in the original module there is a variable V`@` which is not used. Since to check the overlappings there cannot be common variables between the terms, we generate a renamed copy of one of the equations just by renaming each variable.

We declare a sort `CritPair` for critical pairs and a sort `CritPairSet` for sets of critical pairs, and constructors for them. The constructors for critical pairs (`cp`) and for conditional critical pairs (`ccp`) have, respectively, two and four arguments. The two arguments of `cp` and the first two of `ccp` are the terms forming the critical pair. The last two arguments in a conditional critical pair correspond to the condition, which is given following the conventions for conditions in membership axioms, equations, and rules in the `META-LEVEL` module.

```
op cp : Term Term -> CritPair .
```

Deletion of Trivial Equations

$$\frac{\langle\ V;\ \{t =_c^? t,\ E\};\ \sigma\ \rangle}{\langle\ V;\ E;\ \sigma\ \rangle}$$

Check

$$\frac{\langle\ \{x:s,\ V\};\ \{x =_c^? t,\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$

if $x \neq t$ and (x occurs in t or $s \cap LS(t) = \emptyset$)

Decomposition

$$\frac{\langle\ V;\ \{f(t_1,\ \ldots,\ t_n) =_c^? f(t'_1,\ \ldots,\ t'_n),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =_c^? t'_1,\ \ldots,\ t_n =_c^? t'_n,\ E\};\ \sigma\ \rangle}$$

if $n \neq 2$ or f noncommutative

$$\frac{\langle\ V;\ \{f(t_1,t_2) =_c^? f(t'_1,t'_2),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =_c^? t'_1,\ t_2 =_c^? t'_2,\ E\};\ \sigma\ \rangle \vee \langle\ V;\ \{t_1 =_c^? t'_2,\ t_2 =_c^? t'_1,\ E\};\ \sigma\ \rangle}$$

if f commutative

Clash of Symbols

$$\frac{\langle\ V;\ \{f(t_1,\ \ldots,\ t_n) =_c^? g(t'_1,\ \ldots,\ t'_m),\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$

if $n \neq m$ or $f \neq g$

Merging

$$\frac{\langle\ \{x:s,\ V\};\ \{x =_c^? t,\ x =_c^? t',\ E\};\ \sigma\ \rangle}{\langle\ \{x:s,\ V\};\ \{x =_c^? t,\ t =_c^? t',\ E\};\ \sigma\ \rangle}$$

if $x = t \succ t = t'$

Eliminate

$$\frac{\langle\ \{x:s,\ V\};\ \{x =_c^? t,\ E\};\ \sigma\ \rangle}{\langle\ \{x:s,\ V\};\ E\theta;\ \{\sigma\theta,\theta\}\ \rangle}$$

with $\theta = \{x \leftarrow t\}$

if x does not occur in t and $s \cap LS(t) \neq \emptyset$

Transition

$$\frac{\langle\ V;\ \emptyset;\ \sigma\ \rangle}{[\ V;\ \emptyset;\ \sigma;\ \sigma\ ]}$$

Solving ($x \leftarrow t$)

$$\frac{[\ \{x:s,\ V\};\ C;\ \{x \leftarrow t,\ \sigma\};\ \theta\ ]}{[\ V;\ \{(t:s),\ C\};\ \sigma;\ \theta\ ]}$$

Solving ($x : s$)

$$\frac{[\ \{x:s,\ V\};\ \{x:s',\ C\};\ \emptyset;\ \theta\ ]}{\bigvee_{s'' \in s \cap s'} [\ \{x:s'',\ V\};\ C;\ \emptyset;\ \theta\ ]}$$

Solving ($f(t_1,\ldots,t_n) : s$)

$$\frac{[\ V;\ \{f(t_1,\ldots,t_n):s, C\};\ \emptyset;\ \theta\ ]}{\bigvee_{\substack{f:s_1\ldots s_n \rightarrow s' \\ s' \leq s \\ s_1\ldots s_n\ \mathit{maximal}}} [V;\ \{t_1:s_1,\ldots,t_n:s_n, C\};\ \emptyset;\ \theta]}$$

Figure 2: Inference rules for commutative order-sorted unification.

```
op ccp : Term Term Term Term -> CritPair .
```

Given a specification $\mathcal{S}$, the `critPairs` function

```
op critPairs : Module -> CritPairSet .
```

finds all the critical pairs between the equations in $\mathcal{S}$ considered as rules, oriented from left to right. One critical pair is generated for each unifier for each of the possible nonvariable overlappings of the lefthand sides of any two equations in the module. These critical pairs are calculated by finding all the possible such pairs for each of the equations in the module with a renamed copy of each one of the other equations in the module including itself. For each pair of equations, their left sides are unified at any nonvariable position of the term of the first equation, and then a critical pair is constructed for each one of the solutions of the unification problem.

Once all the critical pairs have been generated, the trivial ones—those of the forms `cp(`t`, `t`)` or `ccp(`t`, `t`, `c`)`, for some term t and condition c—are removed by the function `delete`, which is applied again to the set of critical pairs resulting from the simplification process. This simplification is achieved by the function `simplify`, which uses the metalevel function `meta-reduce` to reduce both sides of each of the critical pairs to their normal forms in the given specification. Finally, we apply the function `maxCritPairSet` to get the set of maximal critical pairs among those remaining. In order to get a smallest possible set of critical pairs, besides eliminating trivial ones, we need to also take care of those that are repeated or that are less general than some other critical pair. The function `maxCritPairSet` returns the set of most general critical pairs. The `conflCheck` function is then specified as follows.

```
op delete : CritPairSet -> CritPairSet .
op simplify : CritPairSet Module -> CritPairSet .
op maxCritPairSet : CritPairSet Module -> CritPairSet .
op conflCheck : Module -> CritPairSet .
eq conflCheck(M)
  = maxCritPairSet(delete(simplify(delete(critPairs(M)), M)), M) .
```

4.2.3 Nonconfluence of a Natural Numbers Specification

Our first example illustrating the Church-Rosser checker is the very simple specification `NAT` of the natural numbers presented in Section 2.3. This specification is perfectly reasonable. Its initial model is the set of natural numbers $\mathbb{N}$, with the sum and product operators. However, although it is ground-confluent it is not confluent. of the pair. The output given by the tool is as follows.

```
Checking solution :
  var N : Nat .  var N@ : Nat .
  cp s (N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
```

Note that all variables in the critical pairs, and in the membership assertions are collected and included as part of the output. This critical pair was generated from one of the solutions of the unification corresponding to overlapping at the top the lefthand sides of the equation `s N * M = M + (N * M)` and of its renamed copy `s N@ * M@ = M@ + (N@ * M@)`. One critical pair is generated for each of the solutions of the unification of both lefthand sides at the top. After eliminating the trivial one and reducing each of the terms of the other one in the specification we obtain the critical pair returned by the tool.

In this case the set of membership assertions is empty. This means that the equations are descending. We shall study descent checking in below, and will return to this example, to make it confluent, in Section 4.3.1.

4.2.4 Descent Check

Given a specification $\mathcal{S}$, `descCheck` returns $\mathrm{MMA}(\mathcal{S})$, that is, the set of most general membership assertions among those generated for all specializations of the equations not satisfying the descent condition. For each one of the possible specializations of each of the equations in the module the least sort of the term in the lefthand side is compared with the least sort of the term in the righthand side, reduced to its normal form. Thus, for each equation $(\forall X, l = r)$ that, considered as a rewrite rule, fails to satisfy the descent condition, that is, $LS(r\downarrow) \not\leq LS(l)$, a membership assertion of the form `mb` r `:` $LS(l)$ is generated. Similarly, for a conditional equation $(\forall X, l = r$ `if` $c)$ failing to satisfy the descent condition, a *conditional* membership assertion of the form `cmb` r `:` $LS(l)$ `if` c is generated.

The syntax of the `descCheck` function is as follows.

```
op descCheck : Module -> MembAxSet .
```

4.2.5 The Integers Fail Confluence and Descent

Given the module `NAT` in Section 2.3, let us consider now the following specification for integers to illustrate the way in which the descent check is accomplished.

```
fmod INT is
   protecting NAT .
   sorts Int Neg .
   subsorts Zero < Neg < Int .
   subsort Nat < Int .
   ops (s_) (p_) : Int -> Int .  op p_ : Neg -> Neg .
   ops (_+_) (_*_) : Int Int -> Int [comm] .
   op -_ : Int -> Int .          op square : Int -> Nat .
   vars N M : Nat .  vars I J : Int .  vars P Q : Neg .
   eq s p I = I .                eq p s I = I .
   eq I + 0 = I .                eq I + s N = s (I + N) .
   eq I + p P = p (I + P) .      eq I * 0 = 0 .
   eq I * s N = (I * N) + I .    eq I * p P = (I * P) + - I .
   eq - 0 = 0 .                  eq - s N = p - N .
   eq - p P = s - P .            eq square(I) = I * I .
endfm
```

This specification is perfectly reasonable: it is ground-confluent, and its initial model is the ring of the integers with the usual operations, including also the square function. However, it is not confluent and if fails the descent property.

To check descent, all the instances specializing the variables to smaller sorts are generated for each equation. Let us consider, for example, the instances for the equation `square(I) = I * I`. reduced The specializations for which descent is not satisfied are: `square(I@Neg) = I@Neg * I@Neg`, with `I@Neg` a variable of sort `Neg`, and `square(I) = I * I`, and the corresponding proof obligations generated are `I@Neg * I@Neg : Nat` and `I * I : Nat`. It is easy to see that the second one is more general than the first. Therefore, only the second one is included in the output of the tool, together with three critical pairs.

```
Checking solution :
  var I : Int . var P : Neg . var P@ : Neg .
  var N : Nat . var N@ : Nat .
```

```
cp s - P + (- P@ + (P * P@)) = s - P@ + (- P + (P * P@)) .
cp p - N + (P@ + (N * P@)) = p (P@ + (- N + (N * P@))) .
cp s ( N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
mb I * I : Nat .
```

Regarding the critical pairs, the first one comes from the overlapping of the equation `I * s N = (I * N) + I` with a renamed copy of itself at the top. The same critical pair is also generated in the overlapping of the equation `s N * M = M + (N * M)` with a renamed copy of itself at the top. Nevertheless, only one of them is taken; the other one is eliminated in the last step of the process. The second critical pair comes from the overlapping at the top of the equations `I * s N = (I * N) + I` and `I * p Q = (I * Q) + - I`. Finally, the last of these critical pairs comes from the overlapping of the equation `I * p Q = (I * Q) + - I` with a renamed copy of itself at the top.

4.2.6 The Church-Rosser Checker

In the present section we discuss the function `checking`. This function checks the Church-Rosser property of an order-sorted equational specification satisfying the conditions mentioned in the previous sections. The checking for confluence and for descent is respectively carried out by the functions `conflCheck` and `descCheck` presented above.

The output of the `checking` function is given as a 3-tuple of sort `CheckSol` consisting of a set of critical pairs, a set of membership assertions (corresponding to descent proof obligations) and the set of variables appearing in them.

As already mentioned, the modules passed to the functions `conflCheck` and `descCheck` are previously *prepared* in order to get the variables needed in the respective processes. The function `prepMod` adds to the module the variables needed in the checking. We also define functions `varDeclSet` to collect the set of variables in a set of critical pairs and in a set of membership assertions.

```
op checkSol : VarDeclSet CritPairSet MembAxSet -> CheckSol .
op prepMod : Module -> Module .
op varDeclSet : Module CritPairSet -> VarDeclSet .
op varDeclSet : Module MembAxSet -> VarDeclSet .
```

Finally, the definition of the `checking` function is as follows.

```
op checking : Module -> CheckSol .
eq checking(M)
  = checkSol((varDeclSet(prepMod(M), conflCheck(prepMod(M)))
             varDeclSet(prepMod(M), descCheck(prepMod(M)))),
      conflCheck(prepMod(M)), descCheck(prepMod(M))) .
```

4.3 How to Use the Church-Rosser Checker

This section illustrates with examples the use of the Church-Rosser checker tool, and suggests some methods that—using the feedback provided by the tool—can help the user establish that his/her specification is ground-Church-Rosser. For additional examples see [18, 10].

We assume a context of use very different from the usual context for completing an equational theory. The starting point for completing a theory, say the theory of groups, is an equational theory that is *not* Church-Rosser. A Knuth-Bendix-like completion process then attempts to make it so by *automatically adding* new oriented equations. In our case, however, we assume that the user has developed an *executable specification* of his/her intended system with an initial algebra semantics, and that this specification has already been *tested* with examples, so that the user is in fact confident that the specification is *ground-Church-Rosser*, and wants only to check this property with the tool.

Of course, the tool can only guarantee success when the user's specification is unconditional and Church-Rosser, and not just ground-Church-Rosser. That is, not generating any proof obligations is only a *sufficient* condition. But in many cases of interest the specification will typically be *ground* Church-Rosser, but not Church-Rosser, so that a collection of critical pairs and of membership assertions will be returned by the tool as proof obligations.

An important methodological question is what to do, or not do, with these proof obligations. As the examples that we discuss illustrate, what should *not* be done is to let an automatic completion process add new equations to the user's specification in a mindless way. In some cases this is even impossible, because the critical pair in question cannot be oriented. In many cases it will certainly lead to a nonterminating process. In any case, it will modify the user's specification in ways that can make it difficult for the user to recognize the final result, if any, as intuitively equivalent to the original specification.

The feedback of the tool should instead be used as a guide for *careful thought* about one's specification. As several of the examples studied indicate, by analyzing the critical pairs returned, the user can understand why they could not be joined. It may be a mistake that must be corrected. More often, however, it is not a matter of a mistake, but of an axiom that is either *too general*—so that its very generality makes joining the critical pair impossible, because no more equations can apply to it—or *amenable to an equivalent formulation* that is unproblematic—for example, by reordering the parentheses for an operator that is ground-associative—or both. In any case, it is the user himself/herself who must study where the problem comes from, and how to fix it by modifying the specification. Interaction with the tool then provides a way of modifying the original specification and ascertaining whether the new version passes the test or is a good step towards that goal.

Since the user's specification has an *initial* algebra semantics and the property of interest is checking that it is *ground* Church-Rosser, the proof obligations returned by the tool are *inductive* proof obligations. Therefore, after having introduced some modifications that may already eliminate some of the critical pairs and membership assertions generated by the tool, the user may be left with proof obligations for which the best approach is not any further modification of the specification, but, instead, an inductive proof.

Inductive proof of the joinability of critical pairs is a thornier issue, for which we lack at present good methods. The problem is that, if we have a critical pair $\texttt{cp}(t,\ t')$ generated by the tool for a module M, proving inductively $M \vdash_{\text{ind}} t = t'$ is *not* sufficient for ensuring joinability. This is because such a proof can add new equations as lemmas that may destroy the Church-Rosser property, so that joinability with those new equations does not guarantee joinability with the original equations. What must instead be proved inductively is $M \vdash_{\text{ind}} t \downarrow t'$ but this requires new inductive methods that, as far as we know, have not yet been developed.

A related unresolved methodological issue is what to do with *conditional* critical pairs or membership assertions whose conditions are *unsatisfiable*. For example, the condition may be of the form `tt = ff`, where `tt` and `ff` are constants in a user-defined Boolean sort. If we already *knew* that the specification was Church-Rosser, we could conclude from the fact that `tt` and `ff` are irreducible that they are different, so that the condition cannot be satisfied. But this is precisely what we need to prove. It is quite possible that a modular/hierarchical approach could be used, in conjunction with new inductive proof methods, to establish the unsatisfiability of such conditions and then discard the corresponding proof obligations. But such an approach has still to be developed.

4.3.1 Making the Natural Numbers Specification Confluent

In Section 4.2.3 we studied the confluence of a simple specification for the natural numbers. Let us reconsider it and analyze in more detail the checker's output, and how we can make the specification confluent. If we manage to make it confluent, since no membership assertions are generated by the tool, this would show that the specification is Church-Rosser and, a fortiori, ground-Church-Rosser.

The user of the Church-Rosser checker tool can in some cases succeed by orienting a critical pair, adding it to the specification, checking termination, and resubmitting the modified specification to the checker. However, in this case it is clear that this critical pair cannot be oriented, since, if we were to orient it, its addition would turn the specification into a nonterminating one. The way to handle this problem in our approach consists in studying the specification, and, in particular, the equations generating the critical pairs, and trying to find a "smart" solution modifying the specification in a way as intuitive and as clear as possible, since it is the user who decides the modifications to apply. The key is then to give to the user the information needed to allow him/her to proceed as he/she thinks best.

As seen in Section 4.2.3, only one critical pair is generated for the specification of the natural numbers presented there. We saw how this critical pair comes from the overlap of the equation `s N * M = M + (N * M)` with a renamed copy of itself at the top. This indicates that a term of the form `s N * s M` can be rewritten to both `s (N + (M + (N * M)))` and `s (M + (N + (N * M)))`. Note that these terms cannot be further reduced, but their ground instances can be reduced; that is, the tool's output does not contradict the specification's ground confluence.

Looking to the unjoinable terms in the picture above, we can think of writing such equation as, for example, `s N * s M = s ((N + M) + (N * M))`. Replacing the original equation by this one in the input to the Church-Rosser checker, the tool does not return any critical pair. This means that, once termination has been checked, we are guaranteed that the modified specification is confluent.

Since we do not have any proof obligation for descent we can conclude that the specification is Church-Rosser.

4.3.2 Proving the Integers Specification Ground-Church-Rosser

in Let us now apply our technique to the specification for the integers presented in Section 4.2.5 in order to eliminate the critical pairs discussed there, so as to make the specification confluent.

The first thing we should do is to apply the results in a modular way, that is, we should apply to the specifications the changes that have been applied on the specifications being imported. Therefore, we should incorporate the changes making the specification of the natural numbers confluent in the previous section. However, this does not necessarily mean that we are going to get fewer critical pairs or membership assertions. Notice that in the analysis of the critical pairs we saw how the first critical pair was generated from two different overlappings, one of them between one of the equations in `NAT` with itself, which was eliminated, and another one from one equation in `INT` also with a renamed copy of itself. Thus, in this example we get the same result from the tool.

Observing the considerations of Section 4.2.5, we realize that these critical pairs are generated for exactly the same reasons by which the critical pair for `NAT` was generated. Therefore, we can make similar modifications to `INT`. Specifically, let us replace the equations `I * s N = (I * N) + I` and `I * p P = (I * P) + - I` by the following three ones:

```
eq s N * s M = s ((N + M) + (N * M)) .
eq p P * s N = s ((P + - N) + (P * N)) .
eq p P * p Q = s ((- P + - Q) + (P * Q)) .
```

Calling the checker with these equations in the input module instead of the previous ones no critical pair is given in the output.

Regarding descent, we need to prove inductively the membership assertion given. That is, we have to treat it as the proof obligation that has to be satisfied in order to be able to assert that the specification is ground-decreasing. In this case, we have to prove `INT` $\vdash_{\text{ind}}$ `(∀I) I * I: Nat`. This can be done using the theorem prover presented in Section 3, as was shown in [10]. Therefore, we have successfully transformed our original `INT` specification into one that is confluent and ground-descending, and therefore ground-Church-Rosser.

5 Concluding Remarks

Our experience in building and using the two tools that we have described has been very encouraging. There are several natural extensions of these tools that we have already implemented and others that we plan to pursue in the near future (check Clavel's web-page, `http://sophia.unav.es/~clavel`, for the most recent version of the `ITP`). One promising direction for extensions of the `ITP` is supporting a mixture of the current explicit induction methods and the automated induction techniques for membership equational logic proposed in [3].

Similarly, the Church-Rosser checker should be extended in several directions. First, the set of equational attributes used to rewrite modulo should be extended to contain other axioms besides commutativity. Furthermore, using the results in [3], the tool should be generalized so as to deal with membership equational logic specifications and not just order-sorted specifications. Completion should also be added as an additional facility, perhaps integrating CiME as an auxiliary termination tool. Finally, this tool should also be extended from equational logic to rewriting logic, to check *coherence* [40] of rewrite theories, or to complete such theories so that they become coherent.

More broadly, the present experience suggests that the reflective approach that we have taken in building the present tools is a promising general methodology to build many other theorem proving and formal analysis tools based on formal systems for different logics. Besides the two tools described in this paper, there is in fact a substantial body of experimental evidence supporting the feasibility and convenience of using Maude as a *formal metatool* to generate formal tools from their declarative specifications. The paper [11] discusses several other such formal tool generator applications developed by different authors, including: (1) formal translation tools from HOL to Nuprl, from linear logic to rewriting logic [30, 7], from the Wright architectural description language to CSP and from CSP to rewriting logic, and from pure (higher-order) type systems to rewriting logic [38]; (2) a proof assistant for the open calculus of constructions (an equational extension of the calculus of constructions) [38], and (3) tools for formal specification languages such as the Full Maude tool for Maude [18, 19], the definition in Maude by Millen and Denker of their Common Authentification Specification Language (CAPSL) and its translation into the CIL intermediate language [16], and the execution environment for Tile Logic in rewriting logic [4, 5].

An important added benefit of building a formal environment of tools this way is that, since each tool is a theory in rewriting logic, they are much easier to interoperate by just combining their corresponding rewrite theories. We have for example seen the usefulness of using the inductive theorem prover to prove proof obligations generated by the Church-Rosser checker, and have explained how both tools are integrated with the Full Maude environment. In general, using the reflective techniques and the flexible logical framework capabilities of rewriting logic, we hope to make good advances towards the goal of *formal interoperability* [36], that is, the capacity to move in a mathematically rigorous way across different formalizations, and to use in a rigorously integrated way the different tools supporting such formalizations.

Acknowledgments

We have benefited very much from discussions and positive suggestions from Patrick Lincoln and Natarajan Shankar at SRI, and from those of our fellow members in the Cafe Project, particularly from Kokichi Futatsugi, Ataru Nakagawa, Toshimi Sawada, Masaki Ishiguro, and Masayuki Fujita. Thanks to their efforts, earlier versions of our tools were successfully integrated within the Cafe environment. The present paper is based on extensions of both tools based on the current metalevel of Maude [9] and having convenient user interfaces defined within Maude and a number of additional features.

The theoretical foundations of, and automated deduction techniques for, the tools have benefited very much from our ongoing collaboration with Adel Bouhoula and Jean-Pierre Jouannaud on theorem proving techniques for membership equational logic [3], and from very fruitful discussions with them during the Cafe project.

The explicit induction approach adopted for the inductive theorem prover extends in a reflective way a similar explicit induction approach to equational theorem proving in OBJ proposed by Joseph Goguen [22, 23]. We have benefited considerably from Joseph Goguen's advice and from the OBJ experience in the design of the inductive theorem prover.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and* all that. Cambridge University Press, 1999.

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

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1):35–132, 2000.

[4] R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, July 1998.

[5] R. Bruni, J. Meseguer, and U. Montanari. Internal strategies in a rewriting implementation of tile systems. In Kirchner and Kirchner [28].

[6] CafeOBJ-Project. *Procs. of the CafeOBJ Symposium'98*. 1998.

[7] M. Clavel. *Reflection in general logics and in rewriting logic with applications to the Maude language*. PhD thesis, U. of Navarre, 1998.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [28].

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, January 1999. Available at `http://maude.csl.sri.com`.

[10] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Manuscript, SRI International, February 1998.

[11] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, eds., *Formal Methods*, vol. 1709 of *Lecture Notes in Computer Science*, pp. 1684–1704. Springer-Verlag, 1999.

[12] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [34].

[13] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, ed., *Procs. of Reflection*, 1996.

[14] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [34].

[15] E. Contejean and C. Marché. The CiME system: tutorial and user's manual. Manuscript, Université Paris-Sud, Centre d'Orsay.

[16] G. Denker and J. Millen. CAPSL intermediate language. In N. Heintze and E. Clarke, eds., *Workshop on Formal Methods and Security Protocols*, 1999.

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

[18] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language.* PhD thesis, U. de Málaga, 1999.

[19] F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [28].

[20] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Procs. of Kunming International CASE Symposium*, 1994.

[21] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[22] J. Goguen. OBJ as a theorem prover with application to hardware verification. In P. Subramanyam and G. Birtwistle, eds., *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 218–267. Springer-Verlag, 1989.

[23] J. Goguen. Theorem proving and algebra. Manuscript, August 1997.

[24] 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.

[25] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, 1992.

[26] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[27] C. Kirchner. Order-sorted equational unification. *Procs. of 5th Intl. Conf. on Logic Programming*, 1988.

[28] C. Kirchner and H. Kirchner, eds. *Procs. 2nd Intl. Workshop on Rewriting Logic and its Applications*, vol. 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[29] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, eds., *Procs. of 15th Intl. Coll. on Automata, Languages and Programming*, vol. 317 of *Lecture Notes in Computer Science*, pp. 287–301. Springer, 1988.

[30] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, August 1993. To appear in D.M. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[31] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In Meseguer [34].

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

[33] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 314–390. MIT Press, 1993.

[34] J. Meseguer, editor. *Procs. of the 1st International Workshop on Rewriting Logic and its Applications*, vol. 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[35] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, eds., *Proc. of 7th Intl. Conf. on Concurrency Theory*, vol. 1119 of *Lecture Notes in Computer Science*, pp. 331–372. Springer-Verlag, 1996.

[36] J. Meseguer. Formal interoperability. In *Procs. of the 1998 Conf. on Mathematics in Artificial Intelligence*, 1998.

[37] J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., *Recent Trends in Algebraic Development Techniques*, vol. 1376 of *Lecture Notes in Computer Science*, pp. 18–61. Springer-Verlag, 1998.

[38] J. Meseguer and M.-O. Stehr. Pure type systems in rewriting logic — meta-logical and meta-operational views. Submitted for publication.

[39] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. L. Mètayer, eds., *Research Directions in High-level Parallel Programming Languages*, vol. 574 of *Lecture Notes in Computer Science*, pp. 253–293. Springer-Verlag, 1992.

[40] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., eds., *Proc. of 6th Int. Conf. on Parallel Architectures and Languages Europe*, vol. 817 of *Lecture Notes in Computer Science*, pp. 648–660. Springer-Verlag, 1994.