A Church-Rosser Checker Tool for Maude Equational Specifications

Francisco Durán
E.T.S.I. Informática
Universidad de Málaga
Málaga, Spain
duran@lcc.uma.es

José Meseguer
Computer Science Laboratory
SRI International
Menlo Park, California
meseguer@csl.sri.com

July 28, 2000

Abstract

This document explains the design and use of a Church-Rosser checker tool, which checks whether an equational specification satisfies the Church-Rosser property. This tool can be used to prove the Church-Rosser property of order-sorted equational specifications in Maude [12, 8, 6]. The tool has been written entirely in Maude and is in fact an executable specification in rewriting logic [29] of the formal inference system that it implements. The fact that rewriting logic is reflective [13, 5], and that Maude efficiently supports reflective rewriting logic computations [7, 6] is systematically exploited in the design of the tool.

Contents

1 Introduction **2**
1.1 Reflective Design of the Tool 3
1.2 The Church-Rosser Checker 4

2 Church-Rosser Order-Sorted Specifications **6**
2.1 The Confluence Property 6
2.2 The Descent Property 8

3 The Maude Specification of a Church-Rosser Checker **10**
3.1 Auxiliary Functionality 11
3.1.1 General Operations on Terms 11

3.1.2 Order-Sorted Commutative Unification and Matching 12
3.2 Confluence Check 16
3.2.1 Nonconfluence of a Natural Numbers Specification 21
3.3 Descent Check 23
3.3.1 The Integers Fail Confluence and Descent 24
3.4 The Church-Rosser Checker 27

4 Full Maude and the Church-Rosser Checker 29
4.1 The Signature of the Checker's Interface 31
4.2 Extending the Database by Inheritance 31
4.3 The Full Maude Environment of the Checker 34

5 How to Use the Church-Rosser Checker 36
5.1 Making the Natural Numbers Specification Confluent 38
5.2 Proving the Integers Specification Ground-Church-Rosser 40
5.3 Additional Examples 42
5.3.1 Natural Numbers (II) 42
5.3.2 Integers (II) 46
5.3.3 Rational Numbers 49

6 Concluding Remarks 57

1 Introduction

This document explains the design and use of a Church-Rosser checker, a proving tool to check whether an order-sorted equational specification with an initial algebra semantics satisfies the Church-Rosser property. A previous version of this tool, together with an inductive theorem prover—to prove theorems about such specifications—was developed as part of the Cafe project [4]. It was integrated within the Full Maude environment and also, thanks to the efforts of our colleagues in Japan, within the overall Cafe environment [21]. Through that second integration, both tools can be used to prove formal properties of order-sorted equational specifications in CafeOBJ [16]. They can also be used on their own to prove properties of equational specifications in Maude [12, 8, 6], that is, of Maude functional modules. 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. We explain in Section 4 how a complete execution environment for these tools has been built in Maude, and how they have been integrated with the Full Maude system.

The tool treats equational specifications as *data* that is manipulated. The Church-Rosser checker computes critical pairs and membership asser-

tions 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 [5, 14], and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tool. The same reflective design has been followed for the inductive theorem prover [11], the Knuth-Bendix completion tool [19], and the coherence checker and completion tools [18].

The very high level of abstraction at which the tool has been developed has made it relatively easy for us to build it, makes understanding its implementation much easier, and will make its maintenance and future extensions much easier than if a conventional implementation, say in C, C++, or Java, had instead been chosen. Thanks to the high performance of the Maude engine [6], 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 it has not been optimized for performance, and in spite of using reflection and sophisticated rewriting modulo associativity and associativity-commutativity, the tool has quite competitive 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.

1.1 Reflective Design of the Tool

The design of the present tool exploits and illustrates the logical framework capabilities of rewriting logic by formally specifying the *inference system* of each tool as a rewrite theory [27, 28]. 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.

The Church-Rosser checker tool has a simple design. Let T be a functional module, which has an initial algebra semantics. Such module, that we want to check whether it is Church-Rosser, is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses T as a data structure—that is, it actually uses its metarepresentation $\overline{T}$—and therefore is a rewrite theory at the metalevel. The checking process can be described in a purely functional way, that is, the metalevel theory specifying it is in

fact an equational theory in Maude sublanguage of functional modules.

1.2 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; the result of evaluating an expression is then the canonical form that stands as a unique representative for the equivalence class of terms equal to the original term according to the equations. This approach is fully general; indeed, a well-known result of Bergstra and Tucker [1] 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 computational 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. One can prove termination by providing a reduction order. This is supported by the `check termination` command in the Knuth-Bendix completion tool [19], which can be used in combination with a Church-Rosser checker tool in an environment extending that of Full Maude. 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 even unnecessary if the specification was already ground-Church-Rosser.

Our Church-Rosser checker tool is oriented to checking specifications with an initial algebra semantics that have already been proved terminating 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 5.2). 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 5 some methodological guidelines for the use of the tool, and illustrate the use of the tool 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 where each of the operation symbols has either no equational attributes, or only the commutativity attribute[1]. Furthermore, it is assumed that such specifications do not contain any built-in sort or function, and that they have already been proved terminating. The tool attempts to establish the ground-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. We present the different modules specifying the tools with their main declarations, so that a general idea of the whole system can be gained, omitting declarations of operators not relevant for the general discussion, as well as equations describing the semantics of some functions. We add dots "..." to indicate the missing equations and declarations.

[1] As we shall see in Section 3.1.2, 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. There will be built-in unification and functions `meta-match` and `meta-unify` in Maude in the near future. Once they are available, the tool will be easily extensible to the case in which the different combinations of attributes supported in Maude are also supported by the tool.

Acknowledgements

We are very thankful for the many fruitful discussions with the other members of the Maude team, Manuel Clavel, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Quesada, with whom we have designed and built the Maude language and system.

2 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 [23], and $R \cup A$ is a set of equations such that A is a set of *sort-preserving* equational axioms, that is, whenever $t =_A t'$ we have[2] $LS(t) = LS(t')$. 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.

2.1 The Confluence Property

Let N_+^* be the set of sequences of positive integers, with Λ the empty sequence in N_+^*, and $_ \cdot _$ the concatenation operation on sequences. We call the members of N_+^* *positions*, and denote them by p, p', p'', and so on.

Given a signature Σ, a set of variables X, and a term $t \in \mathcal{T}_\Sigma(X)$ we can define its *set of positions* $\mathcal{P}(t) \subseteq N_+^*$ and *the subterm* $t|_p$ *of* t *at position* p (assuming $p \in \mathcal{P}(t)$) as follows:

1. if $t = x \in X$, then $\mathcal{P}(t) = \{\Lambda\}$ and $t|_\Lambda = t$.
2. if $t = f(t_1, \ldots, t_n)$, then $\mathcal{P}(t) = \{\Lambda\} \cup \{i \cdot p \mid 1 \leq i \leq n,\ p \in \mathcal{P}(t_i)\}$, $f(t_1, \ldots, t_n)|_\Lambda = f(t_1, \ldots, t_n)$, and $f(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$.

Finally, a term t with its subterm $t|_p$ replaced by the term t' is denoted $t[t']_p$.

A substitution is a replacement operation uniquely defined by a mapping from variables to terms, and is written out as $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. Given

[2] We denote the least sort [23] of a term t by $LS(t)$.

a set of axioms A, a substitution σ is an *A-unifier* of t and t' if $\sigma t =_A \sigma t'$, and it is an *A-match* from t to t' if $t' =_A t\sigma$.

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 substitution σ is *at least as general* as substitution ρ if there exists a substitution τ such that $\sigma\tau = \rho$. In this case we write $\rho \lessdot \sigma$. We say that substitution σ is *at least as general* as substitution ρ with respect to a subset X of $D(\sigma)$ if there exists a substitution τ such that $\sigma\tau = \rho$ when σ and ρ are restricted to X. In this case we write $\sigma\tau = \rho[X]$.

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 $p \in \mathcal{P}(t')$ 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 $l \to r$ `if` $\bar{u}\downarrow \bar{v}$ and $l' \to r'$ `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 $p \in \mathcal{P}(l)$ 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.

In the uses we will make of the above definition we will always assume that the unification and the comparison for equality have been performed *modulo* A, where A consists only of commutativity axioms for certain operators in an order-sorted signature; that is, we use order-sorted unification with commutative axioms and equality modulo such axioms. 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 [2].

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 $c\sigma =_A c'$, where, as before, we assume A to consist of commutativity axioms for some operations or to be otherwise empty.

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 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 $\mathcal{S}$—with A involving only commutativity axioms—is terminating, then, if $\mathrm{MCP}(\mathcal{S}) = \emptyset$, we are guaranteed that the specification $\mathcal{S}$ is *confluent*—in the standard sense that if t can be rewritten modulo A to u and v using the rules in $\mathcal{S}$, then u and v can be rewritten modulo A to some w—and therefore, each term t has a unique canonical form $t\downarrow_{\mathcal{S}}$. Note that, due to the presence of conditional equations, we can have $\mathrm{MCP}(\mathcal{S}) \neq \emptyset$ with $\mathcal{S}$ still confluent, because all the conditions in the critical pairs may be unsatisfiable, but establishing such unsatisfiability may require additional reasoning. More importantly for our purposes, even in the unconditional case we can have $\mathrm{MCP}(\mathcal{S}) \neq \emptyset$ with $\mathcal{S}$ *ground*-confluent, that is, confluent for all ground terms. Therefore, assuming termination, $\mathrm{MCP}(\mathcal{S}) = \emptyset$ will ensure the confluence and, a fortiori, the ground-confluence of $\mathcal{S}$, but this is only a sufficient condition.

2.2 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* [26, 22]. 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 and rewrite rules. 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 $(\forall X, l \to r \texttt{ if } c)$ is a rule $(\forall \rho(X), \rho(l) \to \rho(r) \texttt{ if } \rho(c))$ 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*

$$\{\ t'\theta : LS(t\theta) \texttt{ if } c\theta \quad \mid \quad \theta \textit{ is a specialization of } vars(t) \textit{ and } LS(t'\theta\downarrow_{\mathcal{S}}) \not\leq LS(t\theta)\ \}$$

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 $c\sigma =_A c'$. Again, we assume A to consist only of commutativity axioms for some operators, or to be otherwise empty.

Let MMA($\mathcal{S}$) denote the set of most general membership assertions of all of the equations in the specification $\mathcal{S}$. Then, given a specification $\mathcal{S}$, the tool returns

$$\textit{checking}(\mathcal{S}) = \langle\ \mathrm{MCP}(\mathcal{S}),\ \mathrm{MMA}(\mathcal{S})\ \rangle.$$

A fundamental result[3] 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 $\langle\ \mathrm{MCP}(\mathcal{S}),\ \mathrm{MMA}(\mathcal{S})\ \rangle \neq \langle\ \emptyset, \emptyset\ \rangle$ with $\mathcal{S}$

[3] 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 [2].

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.

Of course, in other cases, it may in fact be a matter of some error in the user's specification that the tool uncovers. In any case, the user has complete control on how to modify his/her specification, using the proof obligations in the output of the tool as a guide. In fact, several possibilities exist. The user can prove inductively that the critical pairs are ground-joinable, and that the membership assertions are ground-rewritable to a term with the required sort; however, at present the methods available for such proofs are quite limited. Alternatively, the user can instead modify the specification with the purpose of either correcting a found mistake, or modifying the already correct specification into a variant that the tool will hopefully certify Church-Rosser when resubmitted.

3 The Maude 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.

The input to the Church-Rosser checker tool is given by the functional module we want to check, which, as already explained, must be an order-sorted conditional specification with operators involving only the commutativity attribute and, furthermore, it cannot contain any built-in sorts or functions. As we shall see in Section 4, the tool has been completely integrated within the Full Maude environment, so the Church-Rosser property can be checked on any specification (satisfying the conditions mentioned above) previously entered to Full Maude. Furthermore, the specification in question may be *structured*, with other specifications, perhaps defined by module expressions, as submodules.

The output of the tool consists of a set of critical pairs, a set of membership assertions (corresponding to descent proof obligations), and the set of variables appearing in them.

After introducing in Section 3.1 some auxiliary functionality for the manipulation of terms and order-sorted unification and matching, we explain in Sections 3.2 and 3.3 the specifications of the confluence and descent checks. The specification of the checking of the Church-Rosser property in then

presented in Section 3.4.

3.1 Auxiliary Functionality

In this section we present some required auxiliary functions. We start by introducing some basic functions for the manipulation of terms, and then we present the unification and matching functions.

3.1.1 General Operations on Terms

We summarize in this section the definitions of several operations, in a module extending `META-LEVEL`, that manipulate terms: definitions for positions, as given in Section 2, and operations to get the subterm of a given term at a given position, to replace the subterm of a term at a given position by another term, to get all the nonvariable positions in a term, to apply a substitution to a term, and to get a copy of a term in which all the variables in it have been renamed.

```
fmod EXT-TERM is
  pr META-LEVEL .

  sorts Position PositionSet .
  subsort Position < PositionSet .

  op emptyPosSet : -> PositionSet .
  op posSet : PositionSet PositionSet -> PositionSet
          [assoc comm id: emptyPosSet] .
```

The function `getSubterm` takes a term t and a position p and returns $t|_p$ if $p \in \mathcal{P}(t)$; otherwise, it returns `error*`.

```
  op getSubterm : Term Position -> Term .
```

The function `replace` takes terms t and t' and a position p as input and returns the term $t[t']_p$ if $p \in \mathcal{P}(t)$, or `error*` otherwise.

```
  op replace : Term Term Position -> Term .
```

The function `allNonVarPos` takes a module M and a term t and returns the set $\mathcal{P}(t)$ of all nonvariable positions of t: $\{p \in \mathcal{P}(t) \mid t|_p \notin vars(M)\}$.

```
  op allNonVarPos : Module Term -> PositionSet .
```

The function `substitute` takes a term t and a substitution σ and returns the term $t\sigma$.

```
  op substitute : Term Substitution -> Term .
```

Finally, we define the function `rename`, which takes a term and renames all the variables appearing in it. The renaming is done by replacing each of the variables in the term by another variable with its name concatenated with the character `@`, that is, each variable V is replaced by V`@`.

```
  op rename : Module Term -> Term .

  ...
endfm
```

3.1.2 Order-Sorted Commutative Unification and Matching

In this section we sketch the order-sorted unification algorithm 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. We give below the inference system; their Maude counterparts can be found in [10]. Here we only summarize the basic concepts and the top-level functionality.

Following [25], 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.

Thus, a unification problem is a set of equations, also called a system, denoted $(e_1, \ldots, e_n)$, or a set of systems, also called a disjunction of systems and written $S_1 \vee \ldots \vee S_n$. The set of variables occurring in a unification problem U is denoted $vars(U)$.

Given a specification $\mathcal{S} = (\Sigma, R \cup A)$, a substitution σ is an $\mathcal{S}$-solution of the equation $t =^?_A t'$ if and only if $\sigma(t) =_A \sigma(t')$. The set of $\mathcal{S}$-solutions of an equation $t =^?_A t'$ is denoted $U(t, t', \mathcal{S})$.

A set of substitutions Φ is a complete set of $\mathcal{S}$-solutions of the equation $t =^?_A t'$ away from the set of variables W such that $vars(t) \cup vars(t') \subseteq W$ if and only if

- $\forall \sigma \in \Phi, D(\sigma) \subseteq vars(t) \cup vars(t')$ and $I(\sigma) \cap W = \emptyset$;
- $\forall \sigma \in \Phi, \sigma \in U(t, t', \mathcal{S})$; and
- $\forall \sigma \in U(t, t', \mathcal{S}), \exists \alpha \in \Phi$ such that $\alpha \lessdot_A \sigma[vars(t) \cup vars(t')]$.

Note that, as mentioned in Section 2, we do not compute a minimal set of unifiers. Also, regarding the subsort overloading of operators we assume that an operator symbol has always the same equational attributes—to rewrite modulo such attributes—in all its subsort-overloaded versions; and such equational attributes are restricted to be either the empty set, or the single equation of commutativity.

We give below 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'.

Deletion of Trivial Equations

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

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 $x = t \succ t = t'$

Check

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

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

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

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

These inference rules are mapped into Maude specifications almost without modification. As already mentioned, the complete specification can be found in [10]. We give here just the following definitions.

```
fmod UNIFICATION is
  pr META-LEVEL .
  pr EXT-TERM .
  pr EXT-BOOL .
```

The solution for a unification problem will be given as a set of substitutions, and so, we need the following declarations:

```
  sort SubstitutionSet .
  subsort Substitution < SubstitutionSet .
  op emptySubstitutionSet : -> SubstitutionSet .
  op substitutionSet : SubstitutionSet SubstitutionSet -> SubstitutionSet
          [assoc comm id: emptySubstitutionSet] .

  op unify : Module Term Term -> SubstitutionSet .

  ...
endfm
```

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.

```
fmod MATCHING is
  pr UNIFICATION .
```

For our tool we do not need the solution of the matching; it is enough to know whether there is a match or not. As for the unification case, we consider order-sorted matching with no axioms or only with the commutative axiom, and we assume that any subsort-overloaded function symbols have the same attributes.

```
  op match? : Module EquationSet -> Bool .

  ...
endfm
```

3.2 Confluence Check

We present in this section the specification for the confluence check, carried out by the `confluenceCheck` function. This function is defined in the `CONFLUENCE-CHECK` module, but before we present the following module `CRITICAL-PAIR`, which gives some basic definitions for the manipulation of critical pairs.

```
fmod CRITICAL-PAIR is
  pr MATCHING .
  pr DECL-META-PRETTY-PRINT .
```

We declare sorts for critical pairs (`CritPair`) and for sets of critical pairs (`CritPairSet`), 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 two last 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.

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

  sort CritPairSet .
  subsort CritPair < CritPairSet .
  op emptyCritPairSet : -> CritPairSet .
  op critPairSet : CritPairSet CritPairSet -> CritPairSet
```

```
        [assoc comm id: emptyCritPairSet] .

  var CP CP' : CritPair .
  vars CPS CPS' : CritPairSet .
  vars T T' T1 T1' T2 T2' : Term .
  var M : Module .

  eq critPairSet(CP, CP) = CP .
```

The following function `delete` removes the trivial critical pairs—those of the forms `cp(`t`, `t`)` or `ccp(`t`, `t`, `c`)`, for some term t and condition c—from the set of critical pairs given as argument. To simplify the exposistion, we present only the equations for the unconditional case.

```
  op delete : CritPairSet -> CritPairSet .

  eq delete(critPairSet(cp(T, T'), CPS))
    = if T == T'
      then delete(CPS)
      else critPairSet(cp(T, T'), delete(CPS))
      fi .
  eq delete(emptyCritPairSet) = emptyCritPairSet .
```

The function `simplify` reduces both sides of the critical pairs in the set given as an argument into their normal forms in the given specification.

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

  eq simplify(critPairSet(cp(T, T'), CPS), M)
    = critPairSet(cp(meta-rewrite(M, T, 0), meta-rewrite(M, T', 0)),
        simplify(CPS, M)) .
  eq simplify(emptyCritPairSet, M) = emptyCritPairSet .
```

The function `maximalCPSet` selects the set of most general critical pairs among those given as an argument. In order to get a minimal set of critical pairs we not only have to eliminate trivial ones, but we also need to take care of those that are repeated or that are less general than some other critical pair. The function `maximalCPSet` returns the set of most general critical pairs. It proceeds comparing each critical pair *cp* with all the others; if some critical pair is more general than *cp* then *cp* is removed. As mentioned in Section 2, a critical pair `ccp(`t`, `t'`, `c`)` is more general than another one `ccp(`u`, `u'`, `c'`)` if there exists a substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$, and $c\sigma =_A c'$, with A the commutativity attributes declared for some operators in the specification.

```
op maximalCPSet : CritPairSet Module -> CritPairSet .
op maximalCPSetAux :
     CritPair CritPairSet CritPairSet Module -> CritPairSet .
op moreGeneralCP? : CritPair CritPair Module -> Bool .

eq maximalCPSet(critPairSet(CP, CPS), M)
  = maximalCPSetAux(CP, CPS, emptyCritPairSet, M) .
eq maximalCPSet(emptyCritPairSet, M) = emptyCritPairSet .

eq maximalCPSetAux(CP, critPairSet(CP', CPS), CPS', M)
  = if moreGeneralCP?(CP, CP', M)
    then maximalCPSetAux(CP, CPS, CPS', M)
    else if moreGeneralCP?(CP', CP, M)
         then maximalCPSetAux(
                CP', critPairSet(CPS, CPS'), emptyCritPairSet, M)
         else maximalCPSetAux(CP, CPS, critPairSet(CP', CPS'), M)
         fi
    fi .
eq maximalCPSetAux(CP, emptyCritPairSet, critPairSet(CP', CPS), M)
  = critPairSet(CP, maximalCPSetAux(CP', CPS, emptyCritPairSet, M)) .
eq maximalCPSetAux(CP, emptyCritPairSet, emptyCritPairSet, M) = CP .

eq moreGeneralCP?(cp(T1, T1'), cp(T2, T2'), M)
  = match?(M, ((eq T1 = T2 .) (eq T1' = T2' .)))
    or-else
    match?(M, ((eq T1 = T2' .) (eq T1' = T2 .))) .
```

We define function the `varDeclSetInCritPairs` to collect the set of variables in a set of critical pairs.

```
op varDeclSetInCritPairs : Module CritPairSet -> VarDeclSet .
eq varDeclSetInCritPairs(M, critPairSet(cp(T, T'), CPS))
  = (varDeclSetInTerm(M, T)
     varDeclSetInTerm(M, T')
     varDeclSetInCritPairs(M, CPS)) .
eq varDeclSetInCritPairs(M, emptyCritPairSet) = none .
```

To be able to show the results of the checking, we need meta pretty printing functions. We define here the following `meta-pretty-print` function for critical pairs.

```
op my-meta-pretty-print : Module CritPairSet -> QidList .

eq my-meta-pretty-print(M, critPairSet(cp(T, T'), CPS))
  = ('\n '\t 'cp my-meta-pretty-print(M, T) '= my-meta-pretty-print(M, T') '.
```

```
      my-meta-pretty-print(M, CPS)) .
  eq my-meta-pretty-print(M, emptyCritPairSet) = nil .

  ...
endfm
```

We give now the `CONFLUENCE-CHECK` module, in which we specify two main functions: `confluence` and `critPairs`. We assume that the module passed as an argument to the `confluenceCheck` function 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[4]. 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 using the `rename` function presented in Section 3.1.

```
fmod CONFLUENCE-CHECK is
  pr CRITICAL-PAIR .

  var M : Module .
  vars T T' T1 T1' T2 T2' : Term .
  vars Eq Eq' : Equation .
  vars EqS EqS' EqS'' : EquationSet .
  var P : Position .
  var PS : PositionSet .
  var Subst : Substitution .
  var SubstS : SubstitutionSet .
```

Given a specification $\mathcal{S}$, the `critPairs` function 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 (`critPairs1`) with a renamed copy of each one of the other equations in the module including itself (`critPairs2`). For each pair of equations, their left sides are unified at any nonvariable position of the term of the first equation (`critPairs3`), and then a critical pair is constructed for each one of the solutions of the unification problem (`critPairs4`).

[4]It is also assumed that for each variable V of sort S in the original module there is a variable V`@@`S' for each subsort S' of S. These variables are used in the order-sorted unification algorithm.

To simplify the exposition, we present only the equations for the unconditional case. As said above, the critical pair is formed by `critPairs4` for a pair of equations with an overlapping at some position with some substitution. In the cases when one or both of the equations involved are conditional, then the conjunction of the conditions with the substitution applied to them is placed as the condition of the critical pair.

```
op critPairs : Module -> CritPairSet .
op critPairs1 : Module EquationSet EquationSet -> CritPairSet .
op critPairs2 : Module Equation EquationSet -> CritPairSet .
op critPairs3 : Module Equation Equation PositionSet -> CritPairSet .
op critPairs4 :
     Module Equation Equation Position SubstitutionSet -> CritPairSet .

eq critPairs(M) = critPairs1(M, equationSet(M), equationSet(M)) .

eq critPairs1(M, ((eq T = T' .) EqS), EqS')
  = critPairSet(
      critPairs1(M, EqS, EqS'),
      critPairs2(M, (eq rename(M, T) = rename(M, T') .), EqS')) .
eq critPairs1(M, none, EqS) = emptyCritPairSet .

eq critPairs2(M, (eq T = T' .), (Eq EqS))
  = critPairSet(
      critPairs2(M, (eq T = T' .), EqS),
      critPairs3(M, (eq T = T' .), Eq, allNonVarPos(M, T))) .
eq critPairs2(M, Eq, none) = emptyCritPairSet .

eq critPairs3(M, (eq T1 = T1' .), (eq T2 = T2' .), posSet(P, PS))
  = critPairSet(
      critPairs3(M, (eq T1 = T1' .), (eq T2 = T2' .), PS),
      critPairs4(M, (eq T1 = T1' .), (eq T2 = T2' .),
        P, unify(M, getSubterm(T1, P), T2))) .
eq critPairs3(M, Eq, Eq', emptyPosSet) = emptyCritPairSet .

eq critPairs4(M, (eq T1 = T1' .), (eq T2 = T2' .), P,
     substitutionSet(Subst, SubstS))
  = critPairSet(
      cp(substitute(T1', Subst), replace(substitute(T1, Subst),
         substitute(T2', Subst), P)),
      critPairs4(M, (eq T1 = T1' .), (eq T2 = T2' .), P, SubstS)) .
eq critPairs4(M, Eq, Eq', P, emptySubstitutionSet) = emptyCritPairSet .
```

The `confluenceCheck` is then reduced to the following.

```
  op confluenceCheck : Module -> CritPairSet .

  eq confluenceCheck(M)
    = maximalCPSet(delete(simplify(delete(critPairs(M)), M)), M) .

  ...
endfm
```

3.2.1 Nonconfluence of a Natural Numbers Specification

Our first example illustrating the Church-Rosser checker is the following very simple specification `NAT` of the natural numbers with zero and successor and with a commutative addition operator.

```
 (fmod NAT is
    sorts Zero Nat .
    subsort Zero < Nat .
    op 0 : -> Zero .
    op s_ : Nat -> Nat .
    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)
```

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. One of the solutions for the unification in the overlapping of the last of the equations with itself at the top generates a nonconfluent critical pair.

The output given by the tool is as follows.

```
Maude> (check Church-Rosser .)
rewrites: 5451 in 40ms cpu (60ms real) (136275 rewrites/second)

Church-Rosser checking of NAT
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 (none of the latter in this case) 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

```
eq s N * M = M + (N * M) .
```

and of its renamed copy

```
eq s N@ * M@ = M@ + (N@ * M@) .
```

Because of commutativity the unification of both lefthand sides at the top gives solutions

$$\{(\texttt{N} \leftarrow \texttt{N@});\ (\texttt{M} \leftarrow \texttt{M@})\} \text{ and } \{(\texttt{M} \leftarrow \texttt{s N@});\ (\texttt{M@} \leftarrow \texttt{s N})\}.$$

One critical pair is generated for each of these substitutions:

```
cp(M@ + (N@ * M@), M@ + (N@ * M@))
cp(s N + (N@ * s N), s N@ + (N * s N@))
```

The first one is trivial, and therefore is eliminated. Reducing each of the terms of the other one in the specification we obtain the critical pair

```
cp s(N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
```

Nonconfluent critical pairs generated by the tool can be used by the user to detect possible errors in the specification, or to try to complete it manually. This last possibility is similar to the process used in automatic completion tools, with the difference that the equations added in these tools are usually hard to understand, and that such automatic completion processes often diverge. Here, the user can decide if they are really needed, and then the way in which they are going to be added or can be replaced by other equations.

In this case the set of membership assertions is empty. This means that the equations are descending. We shall study descent checking in Section 3.3, and will return to this example, to make it confluent, in Section 5.1.

3.3 Descent Check

Given a specification $\mathcal{S}$, the function `descentCheck` returns 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 (generated by the function `eqInstanceSet`) 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.

As in the discussion about the specification for the generation of critical pairs, we only present here the equations for the unconditional case.

We assume that for each variable V of sort S in the original module, there is a variable V`@`S' of sort S' for each subsort S' of S, and which is not used in the module.

```
fmod DESCENT-CHECK is
  pr MATCHING .
  pr EXT-BOOL .
```

Comparing the least sort for the sides of each equation is not enough. To check that an equation satisfies the descent condition we have to generate all the possible instances of it for each of the possible specializations of its variables, and we have to verify that in each of these instances the least sort of the term in the lefthand side is greater or equal than the least sort of the term in the righthand side. Given a set of equations, the following function `eqInstanceSet` generates all these instances.

```
  op eqInstanceSet : Module EquationSet -> EquationSet .
```

We do not want all these proof obligations. We only need the most general ones. As defined in Section 2, a membership assertion `mb` t `:` s `if` c `.` is more general than another membership assertion `mb` t' `:` s' `if` c' `.` if there exists a substitution σ such that $t\sigma = t'$, $s \leq s'$, and $c\sigma = c'$. The function `maximalMASet` returns the set of maximal elements in a set of membership assertions.

```
  op maximalMASet : MembAxSet Module -> MembAxSet .
  op maximalMASetAux : MembAx MembAxSet MembAxSet Module -> MembAxSet .
  op moreGeneralMA? : MembAx MembAx Module -> Bool .

  eq maximalMASet((MA MAS), M) = maximalMASetAux(MA, MAS, none, M) .
  eq maximalMASet(none, M) = none .

  eq maximalMASetAux(MA, (MA' MAS), MAS', M)
    = if moreGeneralMA?(MA, MA', M)
      then maximalMASetAux(MA, MAS, MAS', M)
      else if moreGeneralMA?(MA', MA, M)
           then maximalMASetAux(MA', (MAS MAS'), none, M)
           else maximalMASetAux(MA, MAS, (MA' MAS'), M)
           fi
      fi .
  eq maximalMASetAux(MA, none, (MA' MAS), M)
    = (MA maximalMASetAux(MA', MAS, none, M)) .
  eq maximalMASetAux(MA, none, none, M) = MA .

  eq moreGeneralMA?((mb T1 : S1 .), (mb T2 : S2 .), M)
    = match?(M, (eq T1 = T2 .)) and-then sortLeq(M, S1, S2) .
```

The `descentCheck` function is then as follows.

```
  op descentCheck : Module -> MembAxSet .
  op descentCheck1 : Module EquationSet -> MembAxSet .

  eq descentCheck(M)
    = maximalMASet(
        descentCheck1(M, eqInstanceSet(M, equationSet(M))), M) .

  eq descentCheck1(M, ((eq T = T' .) EqS))
    = if sortLeq(M, leastSort(M, meta-reduce(M, T')), leastSort(M, T))
      then descentCheck1(M, EqS)
      else ((mb T' : leastSort(M, T) .) descentCheck1(M, EqS))
      fi .
  eq descentCheck1(M, none) = none .

  ...
endfm
```

3.3.1 The Integers Fail Confluence and Descent

Let us consider the following specification of the integers to illustrate the way in which the descent check is accomplished. Let `NAT` be the module

defined in Section 3.2.1, which is imported by our module `INT`.

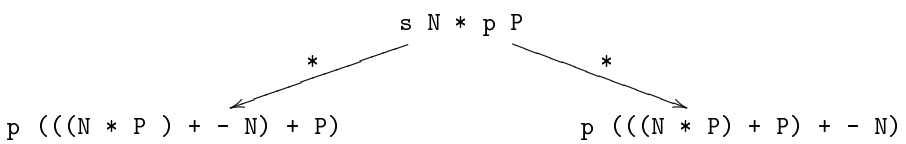
```
(fmod INT is
    sorts Int Neg .
    protecting NAT .
    subsorts Zero < Neg < Int .
    subsort Nat < Int .
    ops s_ p_ : Int -> Int .
    op p_ : Neg -> Neg .
    ops _+_ _*_ : Int Int -> Int [comm] .
    op -_ : Int -> Int .
    op square : Int -> Nat .
    vars N M : Nat .
    vars I J : Int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

As in the `NAT` example, this specification of `INT` is perfectly reasonable: it is ground-confluent, and its initial model is the ring of the integers with the usual operations, including also the successor, predecessor, and the square function. However:

- It is not confluent. For example, the equations

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

  yield an unjoinable critical pair.



- Descent also fails. The equation

```
eq square(I) = I * I .
```

  gives rise to a membership assertion, because the least sort of the term `square(I)` is `Nat`, but it is `Int` for the term in the righthand side.

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

```
eq square(I) = I * I .
```

Note that for a variable `I` of sort `Int` the *preparation step* creates variables `I@Zero`, `I@Nat`, and `I@Neg`, of sorts `Zero`, `Nat`, and `Neg`, respectively. The following equations, specializing the original one, are generated:

```
eq square(I) = I * I .
eq square(I@Zero) = I@Zero * I@Zero .
eq square(I@Nat) = I@Nat * I@Nat .
eq square(I@Neg) = I@Neg * I@Neg .
```

For each of these instances 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. The set of instances for which the descent condition is not satisfied is:

```
eq square(I) = I * I .
eq square(I@Neg) = I@Neg * I@Neg .
```

The proof obligations generated are:

```
mb I * I : Nat .
mb I@Neg * I@Neg : Nat .
```

It is easy to see that the first membership assertion is more general than the second one. Therefore, only the first is included in the output of the tool, together with three critical pairs.

```
Maude> (check Church-Rosser .)
rewrites: 296523 in 2520ms cpu (2530ms real) (117667 rewrites/second)

Church-Rosser checking of INT
Checking solution :
```

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

```
eq 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

```
eq s N * M = M + (N * M) .
```

with itself at the top. Nevertheless, only one of them is taken; the other is eliminated in the last step of the process.

The second critical pair comes from the overlapping at the top of the equations

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

And finally, the last of these critical pairs comes from the overlapping of the equation

```
eq I * p Q = (I * Q) + - I .
```

with a renamed copy of itself at the top.

We will revisit this example to find a confluent variant and to establish the ground-Church-Rosser property for it in Section 5.2.

3.4 The Church-Rosser Checker

In the present section we discuss the module `CRC`, which contains the definition of 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 carried out by the functions `confluenceCheck` and `descentCheck` presented, respectively, in Sections 3.2 and 3.3.

As already mentioned in Sections 3.2 and 3.3, the modules passed to the `confluenceCheck` and `descentCheck` functions are previously *prepared* in order to get the variables needed in the respective processes. The function `prepareModule` adds to the module the variables needed in the checking. For each variable V of sort S in the original module a variable V`@`[5]—used in the generation of critical pairs—and a variable V`@`S' for each subsort S' of S—used in the descent checking—are created.

```
fmod MODULE-PREPARATION is
  pr META-LEVEL .

  op prepareModule : Module -> Module .

  ...
endfm
```

The `CRC` module is then as follows.

```
fmod CRC is
  pr CONFLUENCE-CHECK .
  pr DESCENT-CHECK .
  pr MODULE-PREPARATION .

  var M : Module .
```

The output of the `checking` function is given as a 3-tuple 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. We define the following sort `CheckingSolution`.

```
  sorts CheckingSolution .
  op checkingSolution :
       VarDeclSet CritPairSet MembAxSet -> CheckingSolution .
```

We define the function `varDeclSetInMembAxs` to collect the set of variables in a set of membership assertions.

[5] As pointed out in Footnote 4, some other variables are added for the unification algorithm, namely, for each variable V of sort S in the original module a variable V`@@`S' for each subsort S' of S is also added.

```
  op varDeclSetInMembAxs : Module MembAxSet -> VarDeclSet .
```

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

```
  op checking : Module -> CheckingSolution .

  eq checking(M)
    = checkingSolution(
        (varDeclSetInCritPairs(prepareModule(M),
           confluenceCheck(prepareModule(M)))
         varDeclSetInMembAxs(prepareModule(M),
           descentCheck(prepareModule(M)))),
        confluenceCheck(prepareModule(M)),
        descentCheck(prepareModule(M))) .

  ...
endfm
```

4 Full Maude and the Church-Rosser Checker

We have presented in Section 3 the Maude specification of a Church-Rosser checker tool. Since this specification is executable, given the appropriate inputs, the different functions in it can be evaluated in Core Maude. However, that way of using the tool is quite inconvenient. Note that the argument of the `checking` function is a term of sort `Module`, that is, the metarepresentation of a module, and that the output returned is a term of sort `CheckingSolution`, which may contain critical pairs and membership assertions with terms at the metalevel. Just to illustrate what this means, we show below the interaction with the tool for the `NAT` specification presented in Section 3.2.1. Note that the output has been manually tabulated.

```
Maude> red checking(
             fmod 'NAT is
                including 'BOOL .
                sorts 'Zero ; 'Nat .
                subsort 'Zero < 'Nat .
                op '0 : nil -> 'Zero [none] .
                op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
                op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
                op 's_ : 'Nat -> 'Nat [none] .
                var 'M : 'Nat .
                var 'N : 'Nat .
                none
```

```
                eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
                eq '_+_[{'0}'Zero, 'N] = 'N .
                eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N,'M]] .
                eq '_*_[{'0}'Zero, 'N] = {'0}'Zero .
              endfm)
result CheckingSolution:
  checkingSolution(
    var 'N : 'Nat .
    var 'N@ : 'Nat .,
    cp('s_['_+_['N, '_+_['N@, '_*_['N, 'N@]]]],
       's_['_+_['N@, '_+_['N, '_*_['N, 'N@]]]]),
    none)
```

The interaction with the tool may become quite hard in this way. Notice that it is even worse in the case of the inductive theorem prover presented in [10, 9], because, as explained in Section 1.1, the design of the theorem prover requires two levels of reflection, that is, the interaction with the tool has to be made at the meta-metalevel of the modules on which the theorems are being proved.

In [17] we illustrated the flexibility of the Full Maude system by extending its module algebra with new module expressions. In the present section, we explain how the specification of the Church-Rosser checker, presented in the previous sections, can be integrated within the Full Maude environment, to allow checking of modules defined in Full Maude and to get a much more convenient user interface. Of course, it would have been possible to define an interface for the tool without integrating it with Full Maude. Since all the infrastructure built for Full Maude can be used by itself, just by selecting functions from that infrastructure in the needed modules, any of the two possibilities can give rise to an interface in a very short time. However, integrating the specifications of Full Maude and of the Church-Rosser checker we not only have such a needed infrastructure. With such an integration we can, for example, check the Church-Rosser property of any module in Full Maude's database. We can therefore use the tool on any module accepted by Full Maude, including structured modules, parameterized modules, etc. We still have, of course, the restrictions mentioned in the previous sections, that is, the module has to be functional and order-sorted, and each of the operation symbols in the specification has either no equational attributes or only the commutativity attribute, and the specification does not contain any built-in sort or function and has already been proved terminating using another tool. In the case of the inductive theorem prover, for example, we are not only able to prove theorems about any of the modules entered in

Full Maude, satisfying certain conditions (see, i.e., [10]); in fact, we can keep the proof developed so far as part of the persistent state. We shall see in Section 4.2 how we can extend the persistent database object with some new attributes to keep the state of the proof.

4.1 The Signature of the Checker's Interface

As explained in [17, 20], extensions to Full Maude often require extensions to the signature in which the input will be parsed. In the case of the Church-Rosser checker, we add declarations for two new commands, one of them to check a specific module in Full Maude's database, which is referred to by name, and another one to check the module by default, that is, the last module introduced in Full Maude.

Note that we introduce the operator declarations defining the new syntax in a module extending the module with the actual syntax, and then the following module `CRC-SIGN` extends `FULL-MAUDE-SIGN`.

```
fmod CRC-SIGN is
  including FULL-MAUDE-SIGN .

  op check`Church-Rosser`. : -> PreCommand .
  op check`Church-Rosser_. : ModExp -> PreCommand .

endfm
```

Now, we define a new constant `CRC-GRAMMAR`, of sort `FModule`. This constant is identified with the metarepresentation of the module to be used in the parsing process, which extends the module identified with `EXT-GRAMMAR`.

```
fmod META-CRC-SIGN is
  inc META-LEVEL .
  pr META-FULL-MAUDE-SIGN .

  op CRC-GRAMMAR : -> FModule .

  eq CRC-GRAMMAR = addImportList((including 'CRC-SIGN .), EXT-GRAMMAR) .

endfm
```

4.2 Extending the Database by Inheritance

In [17, 20], we explained how the persistent state of the Full Maude system is given by a single object, which maintains the database of the system.

Using object-oriented notation we can see this object as belonging to a class `database` defined as follows.

```
class database | db : Database, input : TermList,
                 output : QidList, default : ModId .
```

It is natural to think of extensions of the persistent database of Full Maude by defining subclasses of `database` inheriting its behavior and adding new functionalities to it, new attributes, etc. In the case of the Church-Rosser checker, we only want to add the new commands presented in the previous section. It is enough to add the following declarations, and the corresponding rules describing the behavior associated with the new commands.

```
class CRC .
subclass CRC < database .
```

In the case of the inductive theorem prover we want to keep the state of the proof as part of the persistent object, and then define the different commands acting on this proof. We do this by defining a subclass `ITP` of `database` with an attribute `goal`. The `goal` attribute is declared to be a value of sort `Term`, and keeps the state of the proof meta-metarepresented.

```
class ITP | goal : Term .
subclass ITP < database .
```

Notice that we can combine both tools by defining a new class, say `CRC-ITP`, inheriting from both `CRC` and `ITP`.

```
class CRC-ITP .
subclass CRC-ITP < CRC ITP .
```

In the following module, `CRC-DATABASE-HANDLING`, we extend the module `EXT-DATABASE-HANDLING` (see [17, 20] with the declarations for the new class and with the rules describing the behavior associated with the new commands.

```
mod CRC-DATABASE-HANDLING is
  pr EXT-DATABASE-HANDLING .
  pr CRC .

  var ME : ModExp .
  var DB : Database .
```

```
var M : Module .
var VDS : VarDeclSet .
var CPS : CritPairSet .
var MAS : MembAxSet .
var MN : ModName .
var QIL : QidList .
```

The 'desugared' object-oriented declarations presented above are as follows.

```
sort CRC .
subsort CRC < DatabaseClass .

op CRC : -> CRC .

var Atts : AttributeSet .
var X@CRC : CRC .
var O : Oid .
```

The processing of the checking commands is handled by the following function `processCRChecking`. Its definition is very simple. Given a module expression and a database, it just calls the `checking` function presented in Section 3.4 with the flat form of the specified module in the database, and then gives the output as a list of quoted identifiers ready to be passed to the read-eval-print loop.

```
op processCRChecking : ModExp Database -> QidList .
op processCRCheckingAux : Module CheckingSolution -> QidList .

eq processCRChecking(ME, DB)
  = ('\n 'Church-Rosser 'checking 'of my-meta-pretty-print(ME)
     processCRCheckingAux(getFlatUnit(ME, DB),
       checking(getFlatUnit(ME, DB)))) .

eq processCRCheckingAux(M, checkingSolution(VDS, CPS, MAS))
  = if VDS == none and CPS == emptyCritPairSet and MAS == none
    then ('\n 'Checking 'solution ':
          '\n '\t 'The 'specification 'is 'Church-Rosser '.)
    else ('\n 'Checking 'solution ':
          my-meta-pretty-print(VDS)
          my-meta-pretty-print(addVarDeclSet(VDS, M), CPS)
          my-meta-pretty-print(addVarDeclSet(VDS, M), MAS))
    fi .
```

Finally, we give the rules describing the behavior of the `CRC` object associated to the new commands. In both cases the processing is carried out by the `processCRChecking` function, which is called with the specified module name, in the case of the `check Church-Rosser_.` command, or with the name of the default module, for the `check Church-Rosser .` command.

```
  rl [CRChecking] :
     < O : X@CRC | db : DB, input : ({'check`Church-Rosser`.}'PreCommand),
        output : QIL, default : MN, Atts >
     => < O : X@CRC | db : DB, input : nilTermList,
           output : processCRChecking(MN, DB),
           default : MN, Atts > .
  rl [CRChecking] :
     < O : X@CRC | db : DB, input : ('check`Church-Rosser_.[T]),
        output : QIL, default : MN, Atts >
     => < O : X@CRC | db : DB, input : nilTermList,
           output : processCRChecking(parseModExp(T), DB),
           default : MN, Atts > .

  ...
endm
```

4.3 The Full Maude Environment of the Checker

We finish our specification redefining the `FULL-MAUDE` module. The module `FULL-MAUDE` presented in [17, 20] only changed the `in` rule to make possible the use of the extended signature in the parsing of the inputs. In general, however, we do not only want to change this. We may want to add more predefined modules in the initial database, or to change the way in which the communication with the loop is accomplished. Once Maude has full support for objects and input/output, it will be possible to adapt the present specifications just by redefining these modules.

```
mod CRC-FULL-MAUDE is
  pr META-CRC-SIGN .
  pr CRC-DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .
```

The constant `crc` is used as identifier for the object of class `CRC`.

```
op crc : -> Oid .
op init : -> System .

var Atts : AttributeSet .
var X@CRC : CRC .
var X@Database : DatabaseClass .
var O : Oid .
var DB : Database .
var MN : ModName .
vars QIL QIL' QIL'' : QidList .
var TL : TermList .
```

For the Church-Rosser checker and the theorem prover, we need to add a rule initializing the persistent object as an object of classes `CRC` and `ITP`, respectively. Notice that in the case of the inductive theorem prover, an initial value has to be specified also for the `goal` attribute.

```
rl [init] :
   init
   => [nil,
       < crc : CRC |
            db : evalUnit(CONFIGURATION,
                   evalUnit(TRIV,
                     evalUnit(UP, emptyDatabase))),
            input : nilTermList, output : nil,
            default : nullModName >,
        nil] .
```

The redefinition of the `in` rule uses the `CRC-GRAMMAR` to parse the inputs being entered to the integrated system.

```
crl [in] :
   [QIL,
    < O : X@CRC | db : DB, input : nilTermList,
         output : nil, default : MN, Atts >,
    QIL']
   => if meta-parse(CRC-GRAMMAR, QIL) == error*
      then [nil,
            < O : X@CRC | db : DB, input : nilTermList,
                 output : ('ERROR: 'incorrect 'input '.),
                 default : MN, Atts >,
            QIL']
      else [nil,
            < O : X@CRC | db : DB,
```

```
                input : meta-parse(CRC-GRAMMAR, QIL),
                output : nil, default : MN, Atts >,
            QIL'']
        fi
        if QIL =/= nil .
```

The `out` rule remains as originally specified.

```
  crl [out] :
     [QIL,
      < O : X@Database | db : DB, input : TL,
           output : QIL', default : MN, Atts >,
      QIL'']
     => [QIL,
         < O : X@Database | db : DB, input : TL,
              output : nil, default : MN, Atts >,
          (QIL' QIL'')]
        if QIL' =/= nil .
endm
```

5 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 [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—particularly for specifications with nontrivial sort orderings such as, for example, the specifications of the number hierarchy discussed in this section—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. This is illustrated in Section 5.2 with a specification of the integers with a function `square : Int -> Nat` for which a membership proof obligation

```
(∀I: Int) I * I : Nat
```

is generated by the tool. Provided that no equational lemmas are used in the proof, the inductive theorem prover presented in [10, 9] can be used for this purpose, and in fact succeeds in proving this proof obligation.

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 `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

$$\texttt{tt} = \texttt{ff}$$

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.

5.1 Making the Natural Numbers Specification Confluent

In Section 3.2.1 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 approach of classical completion systems consists in taking the critical pair generated and adding it to the specification after orienting it somehow. 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 was seen in Section 3.2.1, only one critical pair is generated by the tool for the specification of the natural numbers presented there. We saw how this critical pair comes from the overlap of the equation

```
eq s(N) * M = M + (N * M) .
```

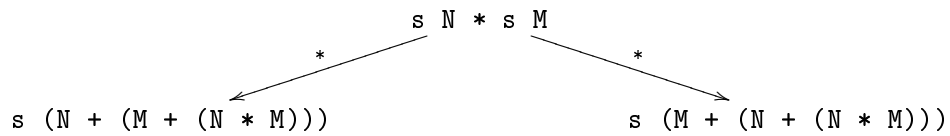
with a renamed copy of itself at the top, that is, its overlapping with this other one:

```
eq s N@ * M@ = M@ + (N@ * M@) .
```

The solution for the unification problem generating it is

$$\{(\texttt{M} \leftarrow \texttt{s N@});\ (\texttt{M@} \leftarrow \texttt{s N})\}.$$

This indicates that a term of the form `s N * s M` can be rewritten as follows.



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.

In many cases, if the specification is ground confluent, we can eliminate the critical pairs just by writing the equations in some other way, or perhaps by adding some new equations. As already mentioned, in other cases the best possibility may be to take the critical pair and add it as an equation after orienting it somehow, but, as we have explained, this critical pair cannot be oriented. In this case, what we can do is to rewrite the "problematic" equation in some other way. Looking to the unjoinable terms in the picture above, we can think of writing such equation as, for example,

```
eq 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. Note the way in which we have associated the variables in the righthand side of the equation. If instead we were to modify the equation as, for example,

```
eq s N * s M = s (N + (M + (N * M))) .
```

we would be in exactly the same situation as before, and the same critical pair would again be generated. The point is that associativity of addition has not been declared as an attribute, and therefore the order of the parentheses is crucial for achieving confluence of the critical pair generated.

Since we do not have any proof obligation for descent we can conclude that the specification is Church-Rosser. The output of the tool is now the following:

```
Maude> (check Church-Rosser .)
rewrites: 5393 in 30ms cpu (40ms real) (179766 rewrites/second)

Church-Rosser checking of NAT
Checking solution :
        The specification is Church-Rosser .
```

5.2 Proving the Integers Specification Ground-Church-Rosser

Let us continue now with the specification for the integers presented in Section 3.3.1. There, we discussed in detail from which equations each of the critical pairs were coming. Let us now apply our technique to eliminate them, so as to make the specification confluent. We will also see how the inductive theorem prover presented in [10, 9] can be used to take care of the membership proof obligation generated. After taking care in this way of all proof obligations we establish the ground-Church-Rosser property.

In Section 3.3.1 we saw that the critical pairs were produced by the equations

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

in different ways. 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. The solution generated is now as follows:

```
Maude> (check Church-Rosser .)
rewrites: 243894 in 2170ms cpu (2190ms real) (112393 rewrites/second)

Church-Rosser checking of INT
Checking solution :
        var I : Int .
        var P : Neg .
        var N : Nat .
        var N@ : Nat .
        var P@ : Neg .
        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 .
```

Observing the results and the considerations of Section 3.3.1, we realize that these critical pairs are generated for exactly the same reasons by which the critical pair in the specification for the natural numbers in Section 5.1 was generated. Therefore, we can make similar modifications in `INT`. Specifically, let us replace the two equations

```
eq I * s N = (I * N) + I .
eq 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.

```
Maude> (check Church-Rosser .)
rewrites: 77133 in 670ms cpu (680ms real) (115123 rewrites/second)

Church-Rosser checking of INT
Checking solution :
        var I : Int .
        mb I * I : Nat .
```

Regarding the descent property, 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 [10, 9], 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.

5.3 Additional Examples

We present here several additional examples to illustrate the use of the tool. No attempt is made to study each of them in full detail, although we will make some comments about the results given by the tool and about how to make them Church-Rosser. Our intention is not to give a large collection of examples, but just a representative collection. Some more examples can be found in [9]. We give new specifications for natural numbers, integers, and rational numbers.

5.3.1 Natural Numbers (II)

We give now a specification for the natural numbers quite similar to the one presented in the manual of OBJ3 ([24], page 77), although with some modifications. In particular, the `assoc` attribute is removed from the declaration of the `_+_` operator, and the declarations are changed so that no built-in operators are used. A new data type `Boolean` is defined in the following module `BOOLEAN`, the operator `_>_` in `NAT` is defined of sort `Boolean`, and the equations in which the `if_then_else_fi` operator was used are replaced by equivalent sets of conditional equations.

```
(fmod BOOLEAN is
```

```
  sort Boolean .
  op true : -> Boolean .
  op false : -> Boolean .
endfm)

(fmod NAT is
  pr BOOLEAN .
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  op _>_ : Nat Nat -> Boolean .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : NzNat NzNat -> NzNat [comm] .
  vars N M : Nat .
  vars N' M' : NzNat .
  eq p s N = N .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq N * 0 = 0 .
  eq 0 * N = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
  eq d(0, N) = N .
  eq d(s N, s M) = d(N, M) .
  ceq quot(N, M') = s quot(d(N, M'), M') if N > M' == true .
  eq quot(M', M') = s 0 .
  ceq quot(N, M') = 0 if M' > N == true .
  eq gcd(N', N') = N' .
  ceq gcd(N', M') = gcd(d(N', M'), M') if N' > M' == true .
endfm)
```

We get the following result with the `check Church-Rosser .` command.

```
Maude> (check Church-Rosser .)
rewrites: 66900 in 370ms cpu (380ms real) (180810 rewrites/second)

Church-Rosser checking of NAT
```

```
Checking solution :
   var M' : NzNat .
   var N' : NzNat .
   var M'@ : NzNat .
   var N@ : Nat .
   var N'@ : NzNat .
   var N@@NzNat : NzNat .
   ccp N'@ = gcd(N'@, d(N'@, N'@))
      if N'@ > N'@ == true .
   ccp s quot(d(M'@, N@), M'@) = 0
      if N@ > M'@ == true and M'@ > N@ == true .
   ccp s quot(d(N@@NzNat, N@@NzNat), N@@NzNat) = s 0
      if N@@NzNat > N@@NzNat == true .
   ccp s 0 = 0
      if M'@ > M'@ == true .
   ccp gcd(M'@, d(M'@, N'@)) = gcd(N'@, d(M'@, N'@))
      if N'@ > M'@ == true and M'@ > N'@ == true .
   ccp 0 = s quot(d(M'@, N@), M'@)
      if M'@ > N@ == true and N@ > M'@ == true .
   cmb gcd(d(N', M'), M') : NzNat if N' > M' == true .
```

We have some critical pairs and one proof obligation for the descent property. Let us study them.

After a quick overview of the critical pairs generated we realize that we do not have to worry about most of them. We can see that the conditions in the critical pairs cannot be satisfied. Nevertheless, to eliminate conditional critical pairs with conditions that cannot be satisfied or even to reduce the conditions if possible is a tricky matter for which we do not yet have good methods at the moment. The difficult question is how to prove that two terms are not equal if we do not yet know whether our specification is confluent. Some research needs to be done in this direction before deciding how to handle these cases. For the time being the user will have to decide what to do with the critical pairs generated, and may have to consider a modified specification.

Let us look for example at the critical pair

```
ccp s 0 = 0
  if M'@ > M'@ == true .
```

A natural number cannot be greater than itself, and the condition will never be satisfied. Therefore, in an intuitive sense we do not need to worry about this critical pair.

The same happens with the following two critical pairs:

```
ccp N'@ = gcd(N'@, d(N'@, N'@))
  if N'@ > N'@ == true .
ccp s quot(d(N@@NzNat, N@@NzNat), N@@NzNat) = s 0
  if N@@NzNat > N@@NzNat == true .
```

The conditions of the remaining three conditional critical pairs cannot be satisfied either, although in these cases it is because the conditions are given as conjunctions of two equalities that cannot be true simultaneously.

```
ccp s quot(d(M'@, N@), M'@) = 0
  if N@ > M'@ == true and M'@ > N@ == true .
ccp gcd(M'@, d(M'@, N'@)) = gcd(N'@, d(M'@, N'@))
  if N'@ > M'@ == true and M'@ > N'@ == true .
ccp 0 = s quot(d(M'@, N@), M'@)
  if M'@ > N@ == true and N@ > M'@ == true .
```

We have seen that none of the conditions in the critical pairs can be satisfied. Therefore, we can gain an intuitive assurance that our specification is ground confluent. But in order to have a full proof we still need to find proof methods to check the unsatisfiability of the condition in each of the critical pairs.

Let us study now the membership proof obligation

```
cmb gcd(d(N', M'), M') : NzNat if N' > M' == true .
```

It comes from the equation

```
ceq gcd(N', M') = gcd(d(N', M'), M') if N' > M' == true .
```

Since `gcd` has been declared as

```
op gcd : NzNat NzNat -> NzNat [comm] .
```

and the function `d` has been declared as

```
op d : Nat Nat -> Nat [comm] .
```

the term `gcd(d(N', M'), M')` does not have sort `NzNat`, which then gives rise to the proof obligation above. This membership assertion has to be proved to guarantee the descent of the specification. Unfortunately, the present inductive theorem prover does not handle proofs of conditional membership assertions, and, in addition, we also lack at present methods to handle the confluence assumption hidden in the equality of the condition.

5.3.2 Integers (II)

We give now a specification for the integers, also similar to that presented in the manual of OBJ ([24, page 77]). Let us suppose that the module `NAT` being imported is the one presented in the previous section. We apply to `INT` the same changes we made to `NAT`, namely, removing the `assoc` attribute for `_+_` and the occurrences of the `if_then_else_fi` operator.

```
(fmod INT is
   sorts Int NzInt .
   protecting NAT .
   subsort Nat < Int .
   subsorts NzNat < NzInt < Int .
   op -_ : Int -> Int .
   op -_ : NzInt -> NzInt .
   op _+_ : Int Int -> Int [comm] .
   op _*_ : Int Int -> Int .
   op _*_ : NzInt NzInt -> NzInt .
   op quot : Int NzInt -> Int .
   op gcd : NzInt NzInt -> NzNat [comm] .
   vars I J : Int .
   vars I' J' : NzInt .
   vars N' M' : NzNat .
   eq - - I = I .
   eq - 0 = 0 .
   eq I + 0 = I .
   eq M' + (- M') = 0 .
   ceq M' + (- N') = - d(N', M') if N' > M' == true .
   ceq M' + (- N') = d(N', M') if M' > N' == true .
   eq (- I) + (- J) = - (I + J) .
   eq I * 0 = 0 .
   eq 0 * I = 0 .
   eq I * (- J) = - (I * J) .
   eq (- J) * I = - (I * J) .
   eq quot(0,I') = 0 .
   eq quot(- I',J') = - quot(I',J') .
   eq quot(I',- J') = - quot(I',J') .
   eq gcd(- I',J') = gcd(I',J') .
endfm)
```

The result generated by the tool is:

```
Maude> (check Church-Rosser .)
rewrites: 280791 in 2300ms cpu (2380ms real) (122083 rewrites/second)
```

```
Church-Rosser checking of INT
Checking solution :
   var I : Int .
   var N'@NzNat : NzNat .
   var M'@NzNat : NzNat .
   var M'@ : NzNat .
   var N'@ : NzNat .
   var I@ : Int .
   var N'@NzNat@ : NzNat .
   var N@Nat@ : Nat .
   var M'@NzNat@ : NzNat .
   cp - (I@ + - I) = I + - I@ .
   cp I@ * I = I * I@ .
   ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
     if N'@NzNat@ > N'@NzNat@ == true .
   ccp - d(M'@, N'@) = d(M'@, N'@)
     if N'@ > M'@ == true and M'@ > N'@ == true .
   ccp - d(N'@, N'@) = 0
     if N'@ > N'@ == true .
   ccp s s 0 = 0
     if 0 > M'@NzNat@ == true .
   ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
     if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
   ccp s quot (d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
     if M'@NzNat@ > M'@NzNat@ == true .
   ccp s 0 = 0
     if M'@NzNat@ > M'@NzNat@ == true .
   ccp d(N'@, N'@) = 0
     if N'@ > N'@ == true .
   ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
      = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
     if N'@NzNat@ > M'@NzNat@ == true
        and M'@NzNat@ > N'@NzNat@ == true .
   cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
     if N'@NzNat > M'@NzNat == true .
```

We have some critical pairs, among which only two are unconditional, and one proof obligation for descent. Let us study them.

None of the conditions in the conditional critical pairs can be satisfied—in an intuitive sense, but with the remaining proof obligation of showing that, indeed, those conditions are unsatisfiable—so, in some sense, we only need to worry about the unconditional critical pairs.

The critical pair

```
cp I@ * I = I * I@ .
```

is due to the fact that the `_*_` operator is not commutative. We can add this attribute, but taking care of adding it not only to the operators declared on sorts `Int` and `NzInt`, but also to the multiplication operators declared on `Nat` and `NzNat` in `NAT`. Note that this will cause the generation of new critical pairs as the ones discussed in Sections 3.2.1 and 3.3.1. We remove them by making the changes introduced in Section 5.2.

The second unconditional critical pair is the following.

```
cp - (I@ + - I) = I + - I@ .
```

A classical completion system would add the equation

```
eq - (I@ + - I) = I + - I@ .
```

to the specification, but this would introduce more critical pairs, and the process would have to be repeated several times without any guarantee of success or even of termination. However, looking at the critical pair we can easily see that the problem is that there is no way to reduce a sum where one of the arguments is a negative number. Moreover, if we try to proceed by adding the oriented critical pair, we obtain critical pairs in which the same situation appears. The question is that a term of the form `I + (- J)` cannot be reduced if `J` is of sort `Int`. But if `J` is a nonzero natural number then some equation could be applied, and if it is smaller than zero then it is of the form `- N'` for some `N'` of sort `NzNat`. But then `- (- N')` can be reduced to `N'`. The only thing we can say then is that probably the specification is confluent, although nothing can be concluded from these critical pairs. However, we can modify the specification not just to reduce the critical pairs, but in fact to avoid generating them. This can be easily done, just by changing the equation

```
eq (- I) + (- J) = - (I + J) .
```

by this other one:

```
eq (- N') + (- M') = - (N' + M') .
```

Assuming that we have checked the termination of the modified specification, after resubmitting it to the tool, we obtain the following result.

```
Maude> (check Church-Rosser .)
rewrites: 284192 in 2510ms cpu (2520ms real) (113223 rewrites/second)
```

```
Church-Rosser checking of INT
Checking solution :
  var N'@NzNat : NzNat .
  var M'@NzNat : NzNat .
  var M'@ : NzNat .
  var N'@ : NzNat .
  var N'@NzNat@ : NzNat .
  var N@Nat@ : Nat .
  var M'@NzNat@ : NzNat .
  ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
     if N'@NzNat@ > N'@NzNat@ == true .
  ccp - d(M'@, N'@) = d(M'@, N'@)
     if N'@ > M'@ == true and M'@ > N'@ == true .
  ccp - d(N'@, N'@) = 0
     if N'@ > N'@ == true .
  ccp s s 0 = 0
     if 0 > M'@NzNat@ == true .
  ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
     if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
  ccp s quot ( d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
     if M'@NzNat@ > M'@NzNat@ == true .
  ccp s 0 = 0
     if M'@NzNat@ > M'@NzNat@ == true .
  ccp d(N'@, N'@) = 0
     if N'@ > N'@ == true .
  ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
     = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
     if N'@NzNat@ > M'@NzNat@ == true
        and M'@NzNat@ > N'@NzNat@ == true .
  cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
     if N'@NzNat >
  M'@NzNat == true .
```

All the critical pairs obtained are now conditional, with their conditions unsatisfiable. Therefore, modulo checking termination and the unsatisfiability of the conditions, we could conclude that it is confluent.

The conditional membership proof obligation is the same one we had for the specification of the natural numbers earlier in this section.

5.3.3 Rational Numbers

Finally we present a specification for the rational numbers, similar to that presented in the manual of OBJ ([24, page 78]). The specification imports the module `INT` with the modifications introduced in the previous section

to make the specification Church-Rosser. The specification is modified, as `NAT` and `INT`, to make it satisfy the requirements of the tool, and the `comm` attribute is added to the declarations of the `_*_` operator.

```
(fmod RAT is
  sorts Rat NzRat .
  protecting INT .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat .
  op -_  : NzRat -> NzRat .
  op _+_ : Rat Rat -> Rat [comm] .
  op _*_ : Rat Rat -> Rat [comm] .
  op _*_ : NzRat NzRat -> NzRat [comm] .
  vars I' J' : NzInt .
  vars R S : Rat .
  vars R' S' : NzRat .
  eq R / (R' / S') = (R * S') / R' .
  eq (R / R') / S' = R / (R' * S') .
  ceq J' / I' = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
     if gcd(J',I') =/= s 0 .
  eq R / s 0 = R .
  eq 0 / R' = 0 .
  eq R / (- R') = (- R) / R' .
  eq - (R / R') = (- R) / R' .
  eq R + (S / R') = ((R * R') + S) / R' .
  eq R * (S / R') = (R * S) / R' .
  eq (S / R') * R = (R * S) / R' .
 endfm)
```

The output of the tool is as follows.

```
Maude> (check Church-Rosser .)
rewrites: 1234819 in 24300ms cpu (24450ms real) (50815 rewrites/second)

Church-Rosser checking of RAT
Checking solution :
  var S' : NzRat .
  var I' : NzInt .
  var N'@NzNat : NzNat .
  var M'@NzNat : NzNat .
  var J' : NzInt .
  var N'@NzNat@ : NzNat .
```

```
var N@Nat@ : Nat .
var M'@NzNat@ : NzNat .
var J'@NzNat : NzNat .
var I'@NzNat : NzNat .
var R : Rat .
var R' : NzRat .
var S@ : Rat .
var S'@ : NzRat .
var R@ : Rat .
var R'@ : NzRat .
var R'@NzInt : NzInt .
var R'@@NzInt : NzInt .
var R@NzInt : NzInt .
var R@@NzInt : NzInt .
var I@Int@NzInt : NzInt .
var S'@NzInt : NzInt .
var S'@@NzInt : NzInt .
var S@@NzInt : NzInt .
cp S@ + (R@ * s 0) = S@ + R@ .
cp R@ / (S'@ * - R') = - R@ / (R' * S'@) .
cp - - R@ / I@Int@NzInt = R@ / I@Int@NzInt .
cp - (S@ + (R@ * - R')) / R' = (- S@ + (R' * R@)) / R' .
cp - (S@ * R@) / R' = (R@ * - S@) / R' .
cp (R + (R' * (R@ * R'@))) / (R' * R'@)
  = (R + (R@ * (R' * R'@))) / (R' * R'@) .
cp (0 + (R@ * R'@)) / R'@ = 0 + R@ .
cp (S' * ((S' * S@) + (R' * R@))) / (S' * R')
  = ((S' * S@) + (R' * R@)) / R' .
cp (S' * (S@ * R@)) / R' = (R@ * (S' * S@)) / R' .
cp (0 * S'@) / R'@ = 0 .
cp (0 * R@) / R'@ = 0 * R@ .
ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
    if N'@NzNat@ > N'@NzNat@ == true .
ccp R@@NzInt
  = quot(R@@NzInt, gcd(R@@NzInt, s 0)) / quot(s 0, gcd(R@@NzInt, s 0))
    if gcd(R@@NzInt, s 0) =/= s 0 .
ccp - d(N'@NzNat@, M'@NzNat@) = d(N'@NzNat@, M'@NzNat@)
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
ccp - d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp R@@NzInt / (S'@ * R'@@NzInt)
  = (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
     / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))) / S'@
    if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
```

```
      / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
  = - (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
       / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt)))
    if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
      / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
  = quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
    / - quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
    if gcd(R@@NzInt, - R'@@NzInt) =/= s 0 .
ccp (S@@NzInt + (R@ * R'@@NzInt)) / R'@@NzInt
  = R@ + (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
    if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp (R@ * S'@@NzInt) / R'@@NzInt
  = R@ / (quot(R'@@NzInt, gcd(R'@@NzInt, S'@@NzInt))
          / quot(S'@@NzInt, gcd(R'@@NzInt, S'@@NzInt)))
    if gcd(R'@@NzInt, S'@@NzInt) =/= s 0 .
ccp (R@ * S@@NzInt) / R'@@NzInt
  = R@ * (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
    if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp s s 0 = 0
    if 0 > M'@NzNat@ == true .
ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
    if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
ccp s quot(d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp s 0 = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
  = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
mb R@NzInt / (R'@NzInt * S'@NzInt) : NzRat .
mb - R@NzInt / R'@NzInt : NzRat .
mb (R@NzInt * S'@NzInt) / R'@NzInt : NzRat .
cmb quot(J', gcd(J', I')) / quot(I', gcd(J', I')) : NzRat
   if gcd(J', I') =/= s 0 .
cmb quot(J', gcd(J', I'@NzNat)) / quot(I'@NzNat, gcd(J', I'@NzNat))
      : NzRat
   if gcd(J', I'@NzNat) =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I')) / quot(I', gcd(J'@NzNat, I'))
      : NzRat
   if gcd(J'@NzNat, I') =/= s 0 .
```

```
cmb quot(J'@NzNat, gcd(J'@NzNat, I'@NzNat))
    / quot(I'@NzNat, gcd(J'@NzNat, I'@NzNat))
      : NzRat
   if gcd(J'@NzNat, I'@NzNat) =/= s 0 .
cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
   if N'@NzNat > M'@NzNat == true .
```

Studying in full detail this specification to make it Church-Rosser is a challenging problem. We are not going to do it here; we just present a general discussion about the methods that can be used for this purpose.

The output of the tool includes twenty seven critical pairs, out of which sixteen are conditional, and also eight membership assertions. Ten of the conditional critical pairs have nonsatisfiable conditions, and so, modulo checking the unsatisfiability of the conditions, we do not have to worry about them.

From the critical pairs

```
cp (0 * S'@) / R'@ = 0 .
cp (0 * R@) / R'@ = 0 * R@ .
```

we see that terms of the form `R * 0` cannot be further reduced. The same happens with terms of forms `s 0 * R` and `0 + R`, as can be observed in the critical pairs

```
cp S@ + (R@ * s 0) = S@ + R@ .
cp (0 + (R@ * R'@)) / R'@ = 0 + R@ .
```

We can eliminate these critical pairs by adding the following equations:

```
eq R * 0 = 0 .
eq R * s 0 = R .
eq R + 0 = R .
```

From the critical pairs

```
cp - - R@ / I@Int@NzInt = R@ / I@Int@NzInt .
cp - (S@ + (R@ * - R')) / R' = (- S@ + (R' * R@)) / R' .
```

we realize the need for the equations

```
eq - - R = R .
eq - R * S = - (R * S) .
```

After introducing these equations there are still twenty critical pairs and eight membership assertions left.

```
(fmod RAT is
   sorts Rat NzRat .
   protecting INT .
   subsort Int < Rat .
   subsorts NzInt < NzRat < Rat .
   op _/_ : Rat NzRat -> Rat .
   op _/_ : NzRat NzRat -> NzRat .
   op -_  : Rat -> Rat .
   op -_  : NzRat -> NzRat .
   op _+_ : Rat Rat -> Rat [comm] .
   op _*_ : Rat Rat -> Rat [comm] .
   op _*_ : NzRat NzRat -> NzRat [comm] .
   vars I' J' : NzInt .
   vars R S : Rat .
   vars R' S' : NzRat .
   eq R / (R' / S') = (R * S') / R' .
   eq (R / R') / S' = R / (R' * S') .
   ceq J' / I' = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
      if gcd(J',I') =/= s 0 .
   eq R / s 0 = R .
   eq 0 / R' = 0 .
   eq R / (- R') = (- R) / R' .
   eq - (R / R') = (- R) / R' .
   eq R + (S / R') = ((R * R') + S) / R' .
   eq R * (S / R') = (R * S) / R' .
   eq (S / R') * R = (R * S) / R' .
   eq R * 0 = 0 .
   eq R * s 0 = R .
   eq R + 0 = R .
   eq - - R = R .
   eq - R * S = - (R * S) .
  endfm)

 Maude> (check Church-Rosser .)
 rewrites: 1239363 in 24040ms cpu (25300ms real) (51554 rewrites/second)

 Church-Rosser checking of RAT
 Checking solution :
   var S' : NzRat .
   var I' : NzInt .
   var N'@NzNat : NzNat .
   var M'@NzNat : NzNat .
```

```
var J' : NzInt .
var N'@NzNat@ : NzNat .
var N@Nat@ : Nat .
var M'@NzNat@ : NzNat .
var J'@NzNat : NzNat .
var I'@NzNat : NzNat .
var R : Rat .
var R' : NzRat .
var S@ : Rat .
var S'@ : NzRat .
var R@ : Rat .
var R'@ : NzRat .
var R'@NzInt : NzInt .
var R'@@NzInt : NzInt .
var R@NzInt : NzInt .
var R@@NzInt : NzInt .
var S'@NzInt : NzInt .
var S'@@NzInt : NzInt .
var S@@NzInt : NzInt .
cp - (S@ + - (R' * R@)) / R' = (- S@ + (R' * R@)) / R' .
cp (R + (R' * (R@ * R'@))) / (R' * R'@)
  = (R + (R@ * (R' * R'@))) / (R' * R'@) .
cp (S' * ((S' * S@) + (R' * R@))) / (S' * R')
  = ((S' * S@) + (R' * R@)) / R' .
cp (S' * (S@ * R@)) / R' = (R@ * (S' * S@)) / R' .
ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
    if N'@NzNat@ > N'@NzNat@ == true .
ccp R@@NzInt
  = quot(R@@NzInt, gcd(R@@NzInt, s 0)) / quot(s 0, gcd(R@@NzInt, s 0))
    if gcd(R@@NzInt, s 0) =/= s 0 .
ccp - d(N'@NzNat@, M'@NzNat@) = d(N'@NzNat@, M'@NzNat@)
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
ccp - d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp R@@NzInt / (S'@ * R'@@NzInt)
  = (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
     / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))) / S'@
    if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
      / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
  = - (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
       / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt)))
    if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
      / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
```

```
  = quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
    / - quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
    if gcd(R@@NzInt, - R'@@NzInt) =/= s 0 .
ccp (S@@NzInt + (R@ * R'@@NzInt)) / R'@@NzInt
  = R@ + (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
    if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp (R@ * S'@@NzInt) / R'@@NzInt
  = R@ / (quot(R'@@NzInt, gcd(R'@@NzInt, S'@@NzInt))
          / quot(S'@@NzInt, gcd(R'@@NzInt, S'@@NzInt)))
    if gcd(R'@@NzInt, S'@@NzInt) =/= s 0 .
ccp (R@ * S@@NzInt) / R'@@NzInt
  = R@ * (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
    if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp s s 0 = 0
    if 0 > M'@NzNat@ == true .
ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
    if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
ccp s quot(d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp s 0 = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
  = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
mb R@NzInt / (R'@NzInt * S'@NzInt) : NzRat .
mb - R@NzInt / R'@NzInt : NzRat .
mb (R@NzInt * S'@NzInt) / R'@NzInt : NzRat .
cmb quot(J', gcd(J', I')) / quot(I', gcd(J', I')) : NzRat
   if gcd(J', I') =/= s 0 .
cmb quot(J', gcd(J', I'@NzNat)) / quot(I'@NzNat, gcd(J', I'@NzNat))
      : NzRat
   if gcd(J', I'@NzNat) =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I')) / quot(I', gcd(J'@NzNat, I'))
      : NzRat
   if gcd(J'@NzNat, I') =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I'@NzNat))
    / quot(I'@NzNat, gcd(J'@NzNat, I'@NzNat))
      : NzRat
   if gcd(J'@NzNat, I'@NzNat) =/= s 0 .
cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
   if N'@NzNat > M'@NzNat == true .
```

Ten of the sixteen conditional critical pairs have unsatisfiable conditions. Some of the critical pairs have been removed, but others may have been generated, or some of the critical pairs that already appeared may be simplified to a different form. Continuing with this process we can either show that some conditions are unsatisfiable or that some critical pairs are ground joinable by additional theorem proving methods, or we can gradually transform the specification into an equivalent one that is confluent, by making completely intuitive and clear changes, decided by the user himself/herself. Although it can be somewhat tedious we think that the advantages are clear. Regarding descent, the proof obligations have also to be either proved or avoided by an adequate transformation of the specification.

6 Concluding Remarks

Our experience in building and using the Church-Rosser checker described in this document has been very encouraging. As it was the development of the inductive theorem prover [11, 9, 10], the termination checker and the Knuth-Bendix completion tool [19], and the coherence checker and completion tools [18].

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 [2, 3], the tool should be generalized so as to deal with membership equational logic specifications and not just order-sorted specifications.

More broadly, the present experience suggests that the reflective approach that we have taken in building the present tools is a promising general methodology for building many other theorem proving and formal analysis tools based on formal systems for different logics. For example, a mechanization of linear logic in Maude has already been demonstrated [5]; and one could also build in this way a tool supporting temporal logic reasoning about nonexecutable properties of Maude modules or of systems defined in other languages.

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. 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* [30], 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.

References

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

[2] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.

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

[4] CafeOBJ-Project. *Proceedings of the CafeOBJ Symposium'98, Numazu, Japan.* April 1998.

[5] Manuel Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD thesis, Universidad de Navarra, 1998.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, 1999. Available at `http://maude.csl.sri.com`.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[9] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.

[10] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Manuscript, SRI International, February 1998.

[11] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection in rewriting logic. In Kokichi Futatsugi, Ataru Nakagawa, and Tetsuo Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. To appear.

[12] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of 1st International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[13] Manuel Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288, 1996.

[14] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proceedings of 1st International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

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

[16] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.

[17] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, June 1999.

[18] Francisco Durán. Coherence checker and completion tools for Maude specifications. Manuscript, 2000.

[19] Francisco Durán. A Knuth-Bendix completion tool for Maude equational specifications. Manuscript, 2000.

[20] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Manuscript, SRI International. Available at `http://maude.csl.sri.com`, February 1999.

[21] Kokichi Futatsugi and Toshimi Sawada. Cafe as an extensible specification environment. In *Proceeding of the Kunming International CASE Symposium, Kunming, China*, November 1994.

[22] Isabelle Gnaedig, Claude Kirchner, and Hélène Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[23] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. Also as Technical Report SRI-CSL-89-10, July, 1989.

[24] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000. Also as Technical Report SRI-CSL-92-03, SRI International, 1992.

[25] Claude Kirchner. Order-sorted equational unification. In *Proceedings of 5th International Conference on Logic Programming*, 1988. Also as Technical Report INRIA 954, December 1988.

[26] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, editors, *Proceedings of 15th International Coll. on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 1988.

[27] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, August 1993. To appear in D.M. Gabbay, editor, *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[28] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.

[29] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[30] José Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence*, 1998.

[31] Patrick Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proceedings Sixth Int. Conference on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer-Verlag, 1994.