CRC 3:
A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications

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

José Meseguer
Computer Science Dept.
University of Illinois at
Urbana-Champaign, IL, USA
`meseguer@uiuc.edu`

December 10, 2009

Abstract

This document explains the design and use of the Church-Rosser checker tool CRC 3, which checks whether a (possibly conditional) equational specification satisfies the Church-Rosser property modulo any combination of associativity, and/or commutativity, and/or identity axioms (combinations of associativity without commutativity are handled only under certain conditions). This tool can be used to prove the Church-Rosser property of order-sorted equational specifications in Maude [11, 8, 12]. The tool has been written entirely in Maude and is in fact an executable specification in rewriting logic [42] of the formal inference system that it implements. The fact that rewriting logic is reflective [5, 17], and that Maude efficiently supports reflective rewriting logic computations [7, 9] is systematically exploited in the design of the tool.

Contents

1 Introduction **3**

1.1 Reflective Design of the Tool . . . 3

1.2 The Church-Rosser Check . . . 4

1.3 Use of the Tool and Commands Available . . . 6

2 Church-Rosser Order-Sorted Specifications Modulo Axioms **9**

2.1 The Confluence Property . . . 10

2.2 The Descent Property . . . 12

2.3 The Result of the Check . . . 13

3 How to Use the Church-Rosser Checker **13**

3.1 Confluence of a Natural Numbers Specification . . . 15

3.2 An Integers Specification Failing Confluence and Descent . . . 18

3.3 Hereditary Finite Sets . . . 21

3.4 Lists and Sets . . . 28

3.5 Booleans . . . 31

3.6 Another Specification of the Natural Numbers . . . 31

4 Concluding Remarks **36**

A Commands Available in the CRC Tool **41**

1 Introduction

This document explains the design and use of a Church-Rosser checker, CRC, a proving tool to check whether a (possibly conditional) order-sorted equational specification modulo equational axioms satisfies the Church-Rosser property. It can be used to prove properties of equational specifications in Maude [11, 8, 12]. Although the current version of the tool significantly improves the previous version [26], it still has some restrictions on the kind of modules that can handle. In particular, the Maude modules entered to the tool must satisfy the following requirements:

1. Only order-sorted specifications are allowed; modules cannot contain membership axioms.
2. Built-in modules are not supported, and built-in operations such as, e.g., `_==_` and `_=/=_` in the `BOOL` predefined module are not allowed.
3. The `owise` attribute cannot be used.

The tool checks that none of these features is used in the specifications entered, and gives an error message in case any of them is detected. Moreover, the specification is assumed terminating, a property for which the Maude Termination Tool (MTT) could be used [23]. If a module with membership axioms or non-equational rewrite rules is entered, the CRC just discards them. This may be useful for checking the Church-Rosser property of the equational part of a system module, that typically contains both equations and non-equational rules.

The main improvements of the tool with respect to its previous version [26] are the following:

- Thanks to the availability in Maude 2.4 of unification modulo commutativity (C) and associativity and commutativity (AC) [12], and the new techniques for dealing with frequently used combinations of equational theories in [24], the CRC 3 can now handle modules with operators declared C, CU, AC, and ACU, that is, any combination of A, C, and U (including A without C in many frequent cases).
- The tool now discards those non-joinable conditional critical pairs that are context-joinable or unfeasible (see Section 2.1).
- The tool is more robust.

An important feature of this tool is that it is written entirely in Maude, and is in fact an *executable specification* in rewriting logic of the formal inference system that it implements. The tool treats equational specifications as *data* that is manipulated. The Church-Rosser checker computes critical pairs and membership assertions by inspecting the equations in the original specification. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. Indeed, the fact that rewriting logic is a reflective logic [5, 17], and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tool. The same reflective design has been followed for other Maude tools, as, e.g., the inductive theorem prover [5, 13, 14, 15, 16], the Knuth-Bendix completion tool [22, 13, 14, 15], the coherence checker and completion tools [?, 20, 13, 14, 15, 16]. Real-Time Maude [45, 16], the Maude Termination Tool (MTT) [23, 16], and the Sufficient-Completeness Checker (SCC) [37, 16].

1.1 Reflective Design of the Tool

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++ or Java, had instead been chosen. Thanks to the high performance of the Maude engine [9, 10, 6], these important benefits in ease of development, understandability, maintainability, and extensibility are achieved with a reasonable performance, in spite of using reflection, sophisticated rewriting modulo associativity and associativity-commutativity, and a combination of built-in (AC and C) unification algorithms and theory transformations to deal with identities and the A without C case. We expect that a future version of Maude will also support U-unification in a built-in way, which will further increase CRC's performance.

The design of the present tool exploits and illustrates the logical framework capabilities of rewriting logic by formally specifying the *inference system* of the tool as a rewrite theory [41, 40]. The fact that the CRC reasons 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.

As other tools in the Maude formal environment [16], the CRC has been implemented as an extension of Full Maude [25, 19]. Details on how to extend Full Maude in different forms can be found in, e.g., [19, 21, 27]. Following these techniques, the CRC has been 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 CRC we not only have such a needed infrastructure, but in addition 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 order-sorted, the operation symbols in the specification can have any combination of equational attributes except `assoc` without `comm`, the specification does not contain any built-in function, and has already been proved terminating using another tool.

1.2 The Church-Rosser Check

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 [3] shows that *any* computable algebraic data type can be specified by means of a finite set of ground-Church-Rosser and operationally 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 the operational termination of his/her (possibly conditional) Maude equational specification by using the MTT tool [23]. 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 particularly well-suited for checking specifications with an initial algebra semantics that have already been proved terminating and now need to be checked to be ground-Church-Rosser, although can of course be used to check the Church-Rosser property of conditional order-sorted specifications that do not have an initial algebra semantics, such as, for example, those specified in Maude functional theories [11]. 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 guide in the attempt to establish the ground-Church-Rosser property. Since this property is in fact inductive, in some cases the Maude inductive theorem prover can be enlisted to prove some of these proof obligations (see Section 3.1). 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 3 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 CRC tool only accepts order-sorted conditional specifications, where each of the operation symbols has either no equational attributes, or any combination of associativity/commutativity/identity, except those having associativity without commutativity.[1] Furthermore, it is assumed that such specifications do not contain any built-in function, do not use the `owise` attribute, and that they have already been proved terminating. The tool attempts to establish the ground-Church-Rosser property *modulo* the equational axioms specified for each 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.

[1]In the current version of the tool the critical pairs are computed using the narrowing infrastructure developed by S. Escobar for the Maude-NPA tool [28, 29].

1.3 Use of the Tool and Commands Available

The current version of the tool is CRC 3. The tool is entirely written in Maude as an extension of Full Maude. CRC 3[2] works on Maude 2.4 (alpha 92a)[3] and Full Maude 2.4o.[4] To start the tool one just needs to load the Maude code of the CRC after starting Full Maude. If `maude` is your Maude executable, the Full Maude file is `full-maude.maude`, and the CRC program is in `crc3.maude`, you can start the CRC as follows:

```
$ maude full-maude crchc3

           \||||||||||||||||||/
         --- Welcome to Maude ---
           /||||||||||||||||||\
        Maude alpha92a built: Nov 12 2009 18:47:47
        Copyright 1997-2009 SRI International
           Tue Dec  8 03:45:12 2009

        Full Maude 2.4o December 5th 2009

        Church-Rosser Checker 3h - December 7th 2009

set include BOOL off

set include TRUTH-VALUE on
```

Notice that since the modules entered to the tool cannot use built-in modules, at start up, the CRC drops the default inclusion of the `BOOL` module. The `TRUTH-VALUE` module only contains sort and constant declarations (cf. [12]) and therefore is fine.

The commands available in the CRC tool are the following:

(help .) shows the list of commands available in the tool.

(check Church-Rosser .) checks the Church-Rosser property of the default module.

(check Church-Rosser <module-name> .) checks the Church-Rosser property of the specified module. This feature can be used to check the Church-Rosser property of any (Core) Maude or Full Maude module specified in the same Maude session (see below).

(show CRC critical pairs .) shows the reduced form of the critical pairs that could not be joined in the last `check Church-Rosser` command. Those joined are not shown.

(show all CRC critical pairs .) shows the unreduced form of all critical pairs computed in the last `check Church-Rosser` command.

Their use and the syntax of the results is described in the following sections.

Any module available in Full Maude that satisfies the above-mentioned requirements (1)-(4) can be checked by the Church-Rosser checker. This includes any (Core) Maude module satisfying requirements (1)-(4) that has been entered *before* loading (or re-initializing) Full Maude and CRC 3, which can also be checked using the `check Church-Rosser` <`module-name`> command.

[2]CRC 3 is available at `http://maude.lcc.uma.es/CRChC`.

[3]Maude 2.4 is available at `http://maude.cs.uiuc.edu`.

[4]Full Maude 2.4o is available at `http://maude.lcc.uma.es/FullMaude`.

Once CRC 3 has been started, we can enter a module and check whether it satisfies the Church-Rosser property as follows.

```
Maude> (fmod CNAT 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)

Introduced module CNAT

Maude> (check Church-Rosser .)

Church-Rosser checking of CNAT
Checking solution:
The following critical pairs cannot be joined:
  cp s(N:Nat + (#2:Nat + (N:Nat * #2:Nat)))
    = s(#2:Nat + (N:Nat + (N:Nat * #2:Nat))) .
The specification is sort-decreasing.
```

We will discuss the output above in the following sections. For now, let us say that the tool gives as result a set of critical pairs and a set of membership assertions. Some of the critical pairs and membership assertions may be conditional. The critical pairs given come from the overlapping of the equations in the specification being checked. If labels are used, each critical pair contains the labels of the equations that generated it.

```
Maude> (fmod CNAT is
          sorts Zero Nat .
          subsort Zero < Nat .
          op 0 : -> Zero .
          op s_ : Nat -> Nat .
          ops _+_ _*_ : Nat Nat -> Nat [comm] .
          vars N M : Nat .
          eq [nat01] : 0 + N = N .
          eq [nat02] : s N + M = s (N + M) .
          eq [nat03] : 0 * N = 0 .
          eq [nat04] : s N * M = M + (N * M) .
        endfm)

Introduced module CNAT

Maude> (check Church-Rosser .)

Church-Rosser checking of CNAT
Checking solution:
The following critical pairs cannot be joined:
  cp for nat04 and nat04
    s(N:Nat + (#2:Nat + (N:Nat * #2:Nat)))
```

```
      = s(#2:Nat + (N:Nat + (N:Nat * #2:Nat))) .
  The specification is sort-decreasing.
```

As in Full Maude, if we want to check the Church-Rosser property of a (Core) Maude module, we must select the `CRC` module and restart the loop [12, Section 15.2].

```
  Maude> set include BOOL off .

  Maude> set include TRUTH-VALUE on .

  Maude> fmod CNAT is
            sorts Zero Nat .
            subsort Zero < Nat .
            op 0 : -> Zero .
            op s_ : Nat -> Nat .
            ops _+_ _*_ : Nat Nat -> Nat [comm] .
            vars N M : Nat .
            eq [nat01] : 0 + N = N .
            eq [nat02] : s N + M = s (N + M) .
            eq [nat03] : 0 * N = 0 .
            eq [nat04] : s N * M = M + (N * M) .
          endfm

  Maude> select CRCHC .

  Maude> loop init .

    Church-Rosser Checker 3h - December 7th 2009

  Maude> (check Church-Rosser CNAT .)

  Church-Rosser checking of CNAT
  Checking solution:
  The following critical pairs cannot be joined:
    cp for nat04 and nat04
      s(N:Nat + (#2:Nat + (N:Nat * #2:Nat)))
      = s(#2:Nat + (N:Nat + (N:Nat * #2:Nat))) .
  The specification is sort-decreasing.
```

After introducing some basic formal concepts and results underlying the tool's design, we give recommendations for its use and some examples.

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 Carolyn Talcott, with whom we have designed and built the Maude language and system. We are very thankful to Santiago Escobar, for the narrowing infrastructure he developed for the Maude-NPA tool, part of which has been used in the CRC and ChC tools. The first version of this tool, together with an inductive theorem prover—to prove theorems about such specifications—was developed as part of the Cafe project [1]. It was integrated within the Full Maude environment and also, thanks to the efforts of our colleagues in Japan, within the overall Cafe environment [31]. Through that second integration, both tools could be used to prove formal properties of order-sorted

equational specifications in CafeOBJ [18]. We are particularly grateful to Santiago Escobar for his development of the narrowing modulo axioms feature in Maude, which is used in an essential way as a component of the CRC 3 reflective implementation.

2 Church-Rosser Order-Sorted Specifications Modulo Axioms

In this section we introduce the notion of Church-Rosser order-sorted specification [34]. We assume specifications of the form $\mathcal{R} = (\Sigma, R \cup E)$ where Σ is an E-preregular order-sorted signature and R is E-coherent. Let us start by introducing the notions of E-preregularity and E-coherence, where E is a set of equational axioms that are both regular and linear.

An order-sorted signature $(\Sigma, S, \leq)$ consists of a poset of sorts $(S, \leq)$ and an $S^* \times S$-indexed family of sets $\Sigma = \{\Sigma_{s_1 \ldots s_n, s}\}_{(s_1 \ldots s_n, s) \in S^* \times S}$ of function symbols. Given an S-sorted set $\mathcal{X} = \{\mathcal{X}_s \mid s \in S\}$ of *disjoint* sets of variables, the set $\mathcal{T}(\Sigma, \mathcal{X})_s$ denotes the Σ-algebra of Σ-terms of sort s with variables in $\mathcal{X}$. We denote $[t]_E$ the E-equivalence class of t.

We call an order-sorted signature *E-preregular* if the set of sorts $\{s \in S \mid \exists w' \in [w]_E \text{ s.t. } w' \in \mathcal{T}(\Sigma, \mathcal{X})_s\}$ has a least upper bound, denoted $ls[w]_E$, which can be effectively computed. Indeed, the Maude system automatically checks the E-preregularity of a signature Σ for E any combination of A, C, and U axioms (see [11, Chapter 22.2.5]).

We denote $\mathcal{P}(t)$ the set of positions of a term t, and $t|_p$ *the subterm of* t *at position* p (with $p \in \mathcal{P}(t)$). A term t with its subterm $t|_p$ replaced by the term t' is denoted $t[t']_p$.

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

Given a rewrite theory $\mathcal{R}$ as above, $t \to_{R/E} t'$ iff there exist u, v such that $t =_E u$ and $u \to_R v$ and $v =_E t'$. In general, of course, given terms t and t' with sorts in the same connected component, the problem of whether $t \to_{R/E} t'$ holds is undecidable. For this reason, a much simpler relation $\to_{R,E}$ is defined, which becomes decidable if an E-matching algorithm exists. For any terms u, v with sorts in the same connected component, the relation $u \to_{R,E} v$ holds if there is a position p in u, a rule $l \to r$ in R, and a substitution σ such that $u|_p =_E l\sigma$ and $v = u[r\sigma]_p$ (see [46]).

Of course, $\to_{R,E} \subseteq \to_{R/E}$, but the question is whether any $\to_{R/E}$-step can be simulated by a $\to_{R,E}$-step. We say that $\mathcal{R}$ satisfies this *E-completeness* property if for any u, v with sorts in the same connected component we have:

$$\begin{tikzcd}
u \arrow[r, "R/E"'] \arrow[dr, dotted, "R{,}E"'] & v \arrow[d, dotted, no head, "E"] \\
 & v'
\end{tikzcd}$$

where here and in what follows dotted lines indicate existential quantification.

It is easy to check that E-completeness is equivalent to the following (strong) E-coherence property:

$$\begin{tikzcd}
u \arrow[r, "R/E"'] \arrow[d, equal, "E"'] & v \arrow[d, dotted, no head, "E"] \\
u' \arrow[r, dotted, "R{,}E"'] & v'
\end{tikzcd}$$

If a theory $\mathcal{R}$ is not coherent, we can try to make it so by completing the set of rules R to a set of rules $\widetilde{R}$ by a Knuth-Bendix-like completion procedure (see, e.g., [38, 47, 32]). For theories

E that are combinations of A, C, and U axioms, we can make any specification E-coherent by using a very simple procedure which can be described as below.

In what follows, and for the purposes of the present tool, the axioms E will consist of any combination of associativity, and/or commutativity, and/or identity axioms for certain binary operators in Σ, except associativity without commutativity. Given a specification $\mathcal{R} = (\Sigma, R \cup E)$, we can make it E-coherent by the following procedure:

- If $f(t_1, \ldots, t_n) \to r$ `if` *cond* is a rule and f is *AC* of kind k then should *add* the rule: $f(t_1, \ldots, t_n, x : k) \to f(r, x : k)$ `if` *cond.*
- If $f(t_1, \ldots, t_n) \to r$ `if` *cond* is a rule and f is *ACU* of kind k then should *replace* the given rule by the rule: $f(t_1, \ldots, t_n, x : k) \to f(r, x : k)$ `if` *cond.*

where k is the "kind" in the connected component of the result sort of f.[5] The equations and rules introduced have the same labels than the equations and rules they come from.

To deal with any combination of A, C, and U axioms we combine different techniques now available. Maude 2.4 supports unification modulo commutativity (C) and associativity and commutativity (AC) [12]. Identity axioms and associativity without commutativity are handled using the theory transformations presented in [24]. As pointed out in [24], the transformation cannot be used in practice for the A case because it does not have the finite variant property. However, the alternative semi-algorithm given there can be used in many practical situations. We refer the reader to [24] for further details, but the idea is that if for each operator in a module we cannot narrow on any equation lefthand side using one of the two possible orientations of the associativity equation, then we can handle it just by adding the corresponding associativity equation. See Section 3.4 for an example.

The problem, then, is to check whether our specification $\mathcal{R}$, satisfying above requirements (1)-(4), has the Church-Rosser property. After giving some auxiliary definitions, we introduce the notion of Church-Rosser conditional order-sorted specifications, and describe the sufficient condition used by our tool to attempt checking the Church-Rosser property.

2.1 The Confluence Property

We say that a term t *E-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 E-unified.

Definition 1 *Given an order-sorted equational specification $\mathcal{R} = (\Sigma, R \cup E)$, with Σ E-preregular and R E-coherent, and given conditional rewrite rules $l \to r$* `if` *cond and $l' \to r'$* `if` *cond$'$ in R such that $vars(l) \cap vars(l') = \emptyset$ and $l|_p\sigma =_E l'\sigma$, for some nonvariable position $p \in \mathcal{P}(l)$ and E-unifier σ, then the triple*

$$ccp(l\sigma[r'\sigma]_p,\ r\sigma,\ cond\,\sigma \wedge cond'\sigma)$$

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 E*. Note also that the critical pairs accumulate the substitution instances of the conditions in the two rules, as in [4].

[5]We assume that $(\Sigma, S \leq)$ has been completed by adding a "kind sort" strictly above each sort in each connected component of the poset $(S, \leq)$.

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

Then, given a specification $\mathcal{R}$, let MCP($\mathcal{R}$) denote the set of most general critical pairs between rules in $\mathcal{R}$ that, after simplifying both sides of the critical pair using the equations in $\mathcal{R}$, are not identical critical pairs modulo E of the form $ccp(t, t, cond)$. Under the assumption that the order-sorted equational specification $\mathcal{R}$ is operationally terminating, then, if MCP($\mathcal{R}$) $= \emptyset$, we are guaranteed that the specification $\mathcal{R}$ is *confluent*—in the standard sense that if t can be rewritten modulo E to u and v using the rules in $\mathcal{R}$, then u and v can be rewritten modulo E to some w—and therefore, each term t has a unique canonical form $t\downarrow_{\mathcal{R}}$. Note that, due to the presence of conditional equations, we can have MCP($\mathcal{R}$) $\neq \emptyset$ with $\mathcal{R}$ 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 MCP($\mathcal{R}$) $\neq \emptyset$ with $\mathcal{R}$ *ground*-confluent, that is, confluent for all ground terms. Therefore, assuming termination, MCP($\mathcal{R}$) $= \emptyset$ will ensure the confluence and, a fortiori, the ground-confluence of $\mathcal{R}$, but this is only a sufficient condition.

From those conditional critical pairs which are not satisfiable, the tool can currently discard those that are *context-joinable* or *unfeasible*, based on a result by Avenhaus and Loría-Sáenz [2], which we generalize here to the order-sorted case and modulo E. Let us first introduce some notation.

Let a *context* $C = \{u_1 \to v_1, \ldots, u_n \to v_n\}$ be a set of oriented equations. We denote by $\overline{C}$ the result of replacing each variable x by a new constant $\overline{x}$. And given a term t, $\overline{t}$ results from replacing each variable $x \in vars(C)$ by the constant $\overline{x}$.

Definition 2 *Let R be an order-sorted deterministic term rewrite systems (DTRS) that is quasi-reductive wrt. $\succ$ and let $ccp(s, t, cond)$ be a critical pair resulting from $l_i \to r_i$ `if` $cond_i$ for $i = 1, 2$, and $\sigma \in mgu_E(l_1|_p, l_2)$. We call $ccp(s, t, cond)$* unfeasible *if there are terms t_0, t_1, t_2 such that $\sigma(l_1) \succ_{st} t_0$, $\overline{t_0} \to^*_{R\cup\overline{cond},E} t_1$, $\overline{t_0} \to^*_{R\cup\overline{cond},E} t_2$, and t_1, t_2 are not unifyable and strongly irreducible. We call $ccp(s, t, cond)$* context-joinable *if there is some t_0 such that $\overline{s} \to^*_{R\cup\overline{cond},E} t_0$, $\overline{t} \to^*_{R\cup\overline{cond},E} t_0$*

A Maude order-sorted conditional specification can be converted into an order-sorted DTRS with a simple procedure. Given an *acceptable* conditional rule (see [11]) of the form:

$$t \to t' \ \texttt{if} \ u_1 = v_1 \wedge \ldots \wedge u_k = v_k \wedge v_{k+1} := u_{k+1} \wedge \ldots \wedge v_{k+r} := u_{k+r}$$

(of course the *order* of ordinary and matching equations can be *mixed*) we can perform the following transformation:

(a) Any $v_i := u_i$ becomes a condition $u_i \to v_i$.

(b) Any $u_i = v_i$ where, say, v_i is a *ground term in canonical form* becomes $u_i \to v_i$.

(c) For all other $u_i = v_i$ introduce a *fresh new variable* x_i of the smallest of the sorts of u_i and v_i so that the rules are *sort decreasing*, and *two conditions* $u_i \to x_i$ and $v_i \to x_i$. If the sorts are not comparable, then we can pick its kind.

Maude checks that conditional equational specifications entered are 3-CTRS [44], and we assume it is operationally terminating, and therefore there exists a well-founded order $\succ_{st}$ such that we can use the results in [2] and their extension to the Maude case, whose details will appear elsewhere, to discard some of the conditional critical pairs generated.

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 3 *We call a confluent and terminating conditional order-sorted specification $\mathcal{R} = (\Sigma, R \cup E)$* Church-Rosser *modulo E if and only if it additionally satisfies the following* descent *property: for each term t we have $ls[t]_E \geq ls[t\downarrow_{\mathcal{R}}]_E$. Similarly, we call a ground-confluent and terminating conditional order-sorted equational specification $\mathcal{R} = (\Sigma, R \cup E)$* ground-Church-Rosser *modulo E iff for each ground term t we have $ls[t]_E \geq ls[t\downarrow_{\mathcal{R}}]_E$.*

Note that these notions are more general and flexible than the requirement of confluence and *sort-decreasingness* [39, 33]. 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 $(\Sigma, S, \leq)$, 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.

Thus, being *E-sort-decreasing* means that, for each rewrite rule $l \to r$ and for each specialization substitution ν, we have $ls[r\nu]_E \leq ls[l\nu]_E$. The checkable conditions that we have to add to the critical pairs to test for the descent property are called membership assertions.

Definition 4 *Let $\mathcal{R}$ 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` *cond is defined as*

$$\{\ t'\theta : ls[t\theta]_E \texttt{ if } cond\,\theta \quad \mid \quad \theta \textit{ is a specialization of } vars(t) \textit{ and } ls[t'\theta\downarrow_{\mathcal{R}}]_E \not\leq ls[t\theta]_E\ \}$$

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

2.3 The Result of the Check

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

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

A fundamental result[6] underlying our tool is that the absence of critical pairs and of membership assertions in such an output is a sufficient condition for an operationally terminating specification $\mathcal{R}$ to be *Church-Rosser*. In fact, for terminating unconditional specifications this check is a necessary and sufficient condition; however, for conditional 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 $\langle\ \mathrm{MCP}(\mathcal{R}),\ \mathrm{MMA}(\mathcal{R})\ \rangle \neq \langle\ \emptyset, \emptyset\ \rangle$ with $\mathcal{R}$ 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 $\mathcal{R}$ is ground-Church-Rosser, we may sometimes have specifications that satisfy this property, but for which the tool returns a nonempty set 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 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.

We assume a context of use quite 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 some 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 may be *ground* Church-Rosser, but

[6]A detailed proof of this result is beyond the scope of this manual, and will be presented elsewhere. For related results in membership equational logic see [4].

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 a rule that is either *too general*—so that its very generality makes joining an associated 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 usually has an *initial* algebra semantics and the most common 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 3.2 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 [15] 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 an unconditional critical pair `cp(`t`, `t'`)` generated by the tool for a module M, this already shows that $M \vdash t = t'$ and, a fortiori, that $M \vdash_{\text{ind}} t = t'$. But this is useless for ensuring joinability. What must instead be proved inductively is

$$\vec{M} \vdash_{\text{ind}} t \downarrow t'$$

where $\vec{M}$ is the *rewrite theory* associated to the (oriented) equational theory M. But this requires inductive methods that, as far as we know, are much less developed.

A related unresolved methodological issue is what to do with *conditional* critical pairs or membership assertions whose conditions are *unsatisfiable*. We currently discard critical pairs which the tool can show are *unfeasible* or *context-joinable*, but all remaining unjoinable pairs are left to the user. If we already *knew* that the specification was Church-Rosser, we could use standard induction methods to discard them. 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.

3.1 Confluence 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 commutative addition and product operators.

```
(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 [nat01] : 0 + N = N .
   eq [nat02] : s N + M = s (N + M) .
   eq [nat03] : 0 * N = 0 .
   eq [nat04] : 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 .)

Church-Rosser checking of NAT
Checking solution:
The following critical pairs cannot be joined:
  cp for nat04 and nat04
    s(N:Nat + (#2:Nat + (N:Nat * #2:Nat)))
    = s(#2:Nat + (N:Nat + (N:Nat * #2:Nat))).
The specification is sort-decreasing.
```

Note that the critical pair given as result includes fresh new variables. These new variables are introduced to make sure that there are no shared variables between the overlapped equations. The membership assertions (none of the latter in this case) may also include new variables.

We can obtain all the critical pairs with the `show all CRC critical pairs` command:

```
Maude> (show all CRC critical pairs .)

These are all the critical pairs:
cp for nat01 and nat01
  N:Nat
  = N:Nat .
cp for nat01 and nat02
  s N:Nat
```

```
    = s(0 + N:Nat).
  cp for nat01 and nat04
    M:Nat * s N:Nat
    = M:Nat + (N:Nat * M:Nat).
  cp for nat02 and nat01
    s(0 + #2:Nat)
    = s #2:Nat .
  cp for nat02 and nat02
    s(#2:Nat + s N:Nat)
    = s(N:Nat + s #2:Nat).
  cp for nat02 and nat02
    s(N:Nat + M:Nat)
    = s(N:Nat + M:Nat).
  cp for nat02 and nat04
    M:Nat * s N:Nat
    = M:Nat + (N:Nat * M:Nat).
  cp for nat03 and nat02
    M:Nat + s N:Nat
    = s(N:Nat + M:Nat).
  cp for nat03 and nat03
    0
    = 0 .
  cp for nat03 and nat04
    0
    = 0 + (0 * N:Nat).
  cp for nat04 and nat02
    M:Nat + s N:Nat
    = s(N:Nat + M:Nat).
  cp for nat04 and nat03
    0 + (0 * #2:Nat)
    = 0 .
  cp for nat04 and nat04
    M:Nat + (N:Nat * M:Nat)
    = M:Nat + (N:Nat * M:Nat).
  cp for nat04 and nat04
    s N:Nat + (#2:Nat * s N:Nat)
    = s #2:Nat + (N:Nat * s #2:Nat).
```

The non-joinable critical pair given as result was generated from one of the solutions of the unification corresponding to overlapping at the top the lefthand sides of the equation

```
  eq [nat04] : s N * M = M + (N * M) .
```

and a copy of itself. The overlapping of `nat04` with itself at the top generated one critical pair for each of solutions of the unification problem:

```
  cp for nat04 and nat04
    M:Nat + (N:Nat * M:Nat)
    = M:Nat + (N:Nat * M:Nat).
  cp for nat04 and nat04
    s N:Nat + (#2:Nat * s N:Nat)
    = s #2:Nat + (N:Nat * s #2:Nat).
```

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:Nat + (#2:Nat + (N:Nat * #2:Nat)))
  = s (#2:Nat + (N:Nat + (N:Nat * #2:Nat))) .
```

In this case the set of membership assertions is empty. This means that the equations are descending. 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.

Let us reconsider it and analyze in more detail the checker's output, and how we can make the specification confluent. Nonconfluent critical pairs generated by the tool can help the user detect possible errors in the specification, or to try to complete it manually.

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.

The overlap from which the critical pair comes from indicates that a term of the form `s N * s M` can be rewritten as follows.

$$\begin{tikzcd}
 & \texttt{s N * s M} \arrow[dl, "*"'] \arrow[dr, "*"] & \\
\texttt{s (N + (M + (N * M)))} & & \texttt{s (M + (N + (N * M)))}
\end{tikzcd}$$

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 at the unjoinable terms in the picture above, we can think of writing such equation as, for example,

```
eq [nat04-1] : s N * s M = s ((N + M) + (N * M)) .
```

Let us call `NAT-1` the module resulting from replacing the original `nat04` equation by this one. The CRC does not return any critical pair for `NAT-1`:

```
Church-Rosser checking of NAT-1
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

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 [nat04-2] : 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.

Of course, since all the problems came from the lack of associativity and natural number addition is associative, yet another alternative is to keep the original equations but *add* the `assoc` attribute to `_+_`. In this way, we get

```
(fmod NAT-2 is
   sorts Zero Nat .
   subsort Zero < Nat .
   op 0 : -> Zero .
   op s_ : Nat -> Nat .
   op _+_ : Nat Nat -> Nat [assoc comm] .
   op _*_ : Nat Nat -> Nat [comm] .
   vars N M : Nat .
   eq [nat01] : 0 + N = N .
   eq [nat02] : s N + M = s (N + M) .
   eq [nat03] : 0 * N = 0 .
   eq [nat04] : s N * M = M + (N * M) .
 endfm)

Maude> (check Church-Rosser .)

Church-Rosser checking of NAT-2
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

3.2 An Integers Specification Failing 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-1` be the Church-Rosser module defined in Section 3.1, which is imported by our module `INT`.

```
(fmod INT is
   sorts Int Neg .
   protecting NAT-1 .
   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 [int01] : s p I = I .
    eq [int02] : p s I = I .
    eq [int03] : I + 0 = I .
    eq [int04] : I + s N = s (I + N) .
    eq [int05] : I + p P = p (I + P) .
    eq [int06] : I * 0 = 0 .
    eq [int07] : I * s N = (I * N) + I .
    eq [int08] : I * p P = (I * P) + - I .
    eq [int09] : - 0 = 0 .
    eq [int10] : - s N = p - N .
    eq [int11] : - p P = s - P .
    eq [int12] : square(I) = I * I .
  endfm)
```

As in the NAT example, this specification INT of the integers 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 [int07] : I * s N = (I * N) + I .
  eq [int08] : I * p P = (I * P) + - I .
  ```

  yield an unjoinable critical pair.

$$\begin{tikzcd}
 & \texttt{s N * p P} \arrow[dl, "*"'] \arrow[dr, "*"] & \\
\texttt{p (((N * P) + - N) + P)} & & \texttt{p (((N * P) + P) + - N)}
\end{tikzcd}$$

- Descent also fails. The equation

  ```
  eq [int12] : 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 int12.

Note that for a variable I of sort Int the tool uses 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 [int12] : square(I:Int) = I:Int * I:Int .
eq [int12] : square(I:Zero) = I:Zero * I:Zero .
eq [int12] : square(I:Nat) = I:Nat * I:Nat .
eq [int12] : 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 [int12] : square(I:Int) = I:Int * I:Int .
  eq [int12] : square(I:Neg) = I:Neg * I:Neg .
```

The proof obligations generated are:

```
  mb I:Int * I:Int : 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 four critical pairs.

```
Maude> (check Church-Rosser .)

Church-Rosser checking of INT
Checking solution:
The following critical pairs cannot be joined:
  cp for int07 and int07
    s(N:Nat +(#2:Nat +(N:Nat * #2:Nat)))
    = s(#2:Nat +(N:Nat +(N:Nat * #2:Nat))).
  cp for int07 and int08
    p(P:Neg +(- #2:Nat +(P:Neg * #2:Nat)))
    = p - #2:Nat +(P:Neg +(P:Neg * #2:Nat)).
  cp for int07 and nat04-1
    s(M:Nat +(N:Nat +(N:Nat * M:Nat)))
    = s((N:Nat * M:Nat)+(N:Nat + M:Nat)).
  cp for int08 and int08
    s - P:Neg +(- #2:Neg +(P:Neg * #2:Neg))
    = s - #2:Neg +(- P:Neg +(P:Neg * #2:Neg)).
There are non-sort-decreasing equations.
The following proof obligations must be checked:
  mb I:Int * I:Int : Nat .
```

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 [15] 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.

Observing the results and the considerations above, 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 3.1 was generated. Therefore, we can make similar modifications in INT. Specifically, let us replace the two equations

```
eq [int07] : I * s N = (I * N) + I .
eq [int08] : I * p P = (I * P) + - I .
```

by the following three ones:

```
eq [int07-1] : s N * s M = s ((N + M) + (N * M)) .
eq [int07-2] : p P * s N = s ((P + - N) + (P * N)) .
eq [int07-3] : 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 .)

Church-Rosser checking of INT-1
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
There are non-sort-decreasing equations.
The following proof obligations must be checked:
  mb I:Int * I:Int : Nat .
```

Alternatively, as done for the NAT example, we could declare _+_ to also be associative. However, with such a change we get 64 critical pairs, therefore requiring further modifications.

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 $\texttt{INT} \vdash_{\text{ind}} (\forall\texttt{I})\ \texttt{I * I : Nat}$. This can be done[7] using the theorem prover presented in [15, 14], as was shown in [14]. Therefore, we have successfully transformed our original INT specification into one that is confluent and ground-descending, and therefore ground-Church-Rosser.

3.3 Hereditary Finite Sets

The following module HF-SETS specifies hereditarily finite sets. It was developed by Ralf Sasse and José Meseguer and is inspired by the generalized sets module in the Maude prelude [11, Section 9.12.5]. It declares sorts Set and Magma, with Set a subsort of Magma. Terms of sort Set are generated by constructors {}, the empty set, and {_}, which makes a set out of a term of sort Magma. Magmas have two overloaded associative-commutative operators _,_, one a constructor taking a set and a magma, and the other one taking two magmas. Then, operations _in_, to check whether a magma is in another magma, _equiv_, to check whether two magmas are equivalent, and _<=_, in module HF-SETS-INC, to check whether a magma is contained in another magma. A magma is contained in another magma if all its sets are included; a non-empty set is a subset of a magma if all its element are in the other magma.

```
(fmod HF-SETS is
   sort Magma .
   sort Set .
   subsort Set < Magma .
   op _`,_ : Set Magma -> Magma [ctor assoc comm] .
   op _`,_ : Magma Magma -> Magma [assoc comm] .
   op `{_`} : Magma -> Set [ctor] .
   op `{`} : -> Set [ctor] .

   vars M M' N : Magma .
   vars S S' : Set .
   eq [01]: M, M, M' = M, M' .
   eq [02]: M, M = M .
```

[7]Implicitly, one is taking advantage of the fact that the theorem prover only uses rewriting to prove the goal. More precisely, what is proved is that $\vec{\texttt{INT}} \vdash_{ind} (\forall\texttt{I})(\exists\texttt{z : Nat})\ \texttt{I} * \texttt{I} \longrightarrow^{*} \texttt{z}$, where $\vec{\texttt{INT}}$ is the rewrite theory associated to the (oriented) equations in INT.

```
   op _in_ : Magma Magma -> Bool .

   eq [03]: M in {} = false .
   eq [04]: {} in {{M}} = false .
   eq [05]: {} in {{}} = true .
   eq [06]: {} in {{}, M} = true .
   eq [07]: {} in {{M}, N} = {} in {N} .

   eq [08]: {M} in {M'} = M in M' .

  ceq [09]: S in S', N = true if S in S' = true .
  ceq [10]: S in S', N = S in N if S in S' = false .

  ceq [11]: S, M in M' = M in M' if S in M' = true .
  ceq [12]: S, M in M' = false if S in M' = false .

   op _equiv_ : Magma Magma -> Bool [comm] .
   eq [13]: M equiv M = true .
   eq [14]: {} equiv {M} = false .
   eq [15]: {M} equiv {M'} = M equiv M' .
   eq [16]: S equiv S, M' = S equiv M' .
  ceq [17]: S equiv M = false if S in {M} = false .
  ceq [18]: S, M equiv M' = false if S in {M'} = false .
  ceq [19]: M equiv M' = false if {M} in {{M'}} = false .
 endfm)

(fmod HF-SETS-INC is
   protecting HF-SETS .

   op _<=_ : Magma Magma -> Bool .

   vars M M' : Magma .
   vars S S' : Set .

   eq [20]: {} <= M = true .
  ceq [21]: {M} <= M' = true  if M in M' = true .
  ceq [22]: {M} <= M' = false if M in M' = false .
  ceq [23]: S, M <= M' = M <= M' if S <= M' = true .
  ceq [24]: S, M <= M' = false   if S <= M' = false .
 endfm)
```

The Church-Rosser check gives the following result.

```
Maude> (check Church-Rosser HF-SETS-INC .)

Church-Rosser checking of HF-SETS-INC
Checking solution:
The following critical pairs cannot be joined:
  ccp for 13 and 17
    true
    = false
    if S:Set in {S:Set} = false .
  ccp for 13 and 18
    true
    = false
    if S:Set in {M:Magma, S:Set} = false .
The specification is sort-decreasing.
```

If we use the `show all CRC critical pairs` command, we obtain 1241 critical pairs. Many of them are joinable, and therefore discarded. From the remaining 28 critical pairs, all of which are conditional, 26 are discarded because they can be proved either context-joinable or unfeasible. Let us take a look at some of these.

Let us consider the following critical pair:

```
ccp for 01 and 09
  S:Set in (#1:Magma, S':Set)
  = true
  if S:Set in S':Set = true .
```

It is context joinable. If we add the condition as an equation with its variables `S` and `S'` turned into constants, of sort `Set`, `#S` and `#S'`.

```
  eq #S in #S' = true .
```

Then, the terms `#S in (##1, #S')` and `true`, with `##1` a new constant of sort `Magma`, can be joined.

We found a similar situation for the following critical pair.

```
ccp for 01 and 10
  S:Set in (#1:Magma, S':Set)
  = S:Set in #1:Magma
  if S:Set in S':Set = false
```

Introducing the equation

```
eq #S in #S' = false .
```

the terms `#S in (##1, #S')` and `#S in ##1` can be joined, since the lefthand side of the critical pair can then be rewritten to `S:Set in #1:Magma` by using the equation

```
ceq [10]: S in S', N = S in N if S in S' = false .
```

The following critical pairs are eliminated by using a similar procedure.

```
ccp 01 and 11
  (#1:Magma, S:Set) in M':Magma]
  = #1:Magma in M':Magma
  if S:Set in M':Magma = true
ccp 01 and 12
  (#1:Magma, S:Set) in M':Magma
  false
  if S:Set in M':Magma = false
ccp 01 and 18
  M':Magma equiv (#1:Magma, S:Set)
  = false
  if S:Set in {M':Magma} = false
```

```
ccp 01 and 23
  (#1:Magma, S:Set) <= M':Magma
  = #1:Magma <= M':Magma
  if S:Set <= M':Magma = true
ccp 01 and 24
  (#1:Magma, S:Set) <= M':Magma
  = false
  if S:Set <= M':Magma = false
ccp 02 and 18
  M':Magma equiv S:Set
  = false
  if S:Set in {M':Magma} = false
ccp 15 and 17
  #1:Magma equiv #2:Magma
  = false
  if {#1:Magma} in {{#2:Magma}} = false
ccp 23 and 23
  S:Set <= M':Magma
  = #2:Set <= M':Magma
  if S:Set <= M':Magma = true /\ #2:Set <= M':Magma = true
```

The following critical pairs are discarded for similar reasons, although in these cases, the equations introduced are the only ones used to join them.

```
ccp 02 and 09
  S:Set in S':Set
  = true
  if S:Set in S':Set = true
ccp 02 and 12
  S:Set in M':Magma
  = false
  if S:Set in M':Magma] = false
ccp 02 and 24
  S:Set <= M':Magma
  = false
  if S:Set <= M':Magma = false
```

The following cases are similar, although for these ones an equation is introduced for each of the terms in the conjunctions.

```
ccp 10 and 10
  S:Set in S':Set
  = S:Set in #3:Set
  if S:Set in S':Set = false /\ S:Set in #3:Set = false
ccp 10 and 10
  S:Set in (#10:Magma, S':Set)
  = S:Set in (#10:Magma, #3:Set)
  if S:Set in S':Set = false /\ S:Set in #3:Set = false
ccp 11 and 11
  S:Set in M':Magma
  = #2:Set in M':Magma
  if S:Set in M':Magma = true /\ #2:Set in M':Magma = true
ccp 11 and 11
  (#8:Magma, S:Set) in M':Magma
  = (#8:Magma, #2:Set) in M':Magma)
```

```
    if S:Set in M':Magma = true /\ #2:Set in M':Magma = true
  ccp 23 and 23
    {#8:Magma, S:Set} <= M':Magma
    = {#8:Magma, #2:Set} <= M':Magma
    if S:Set <= M':Magma = true /\ #2:Set <= M':Magma = true
```

The following critical pairs are discarded because they are unfeasible.

```
  ccp 09 and 10
    true
    = S:Set in N:Magma
    if S:Set in S':Set = false /\ S:Set in S':Set = true
  ccp 10 and 09
    S:Set in N:Magma = true
    if S:Set in S':Set = true /\ S:Set in S':Set = false
```

To prove unfeasibility we focus on the conditions. With the rules

```
  #S in #S' = false
  #S in #S' = true
```

the term `#S in #S'` can be rewritten both to `false` and `true`. Since they do not unify and are strongly irreducible, we can conclude that the critical pair is unfeasible.

The following critical pairs are discarded similarly.

```
  ccp 11 and 12
    M:Magma in M':Magma
    = false
    if S:Set in M':Magma = false /\ S:Set in M':Magma = true
  ccp 12 and 11
    false
    = M:Magma in M':Magma
    if S:Set in M':Magma = true /\ S:Set in M':Magma = false
  ccp 21 and 22
    true = false
    if M:Magma in M':Magma = false /\ M:Magma in M':Magma = true
  ccp 22 and 21
    false = true
    if M:Magma in M':Magma = true /\ M:Magma in M':Magma = false
  ccp 23 and 24
    M:Magma <= M':Magma
    = false
    if S:Set <= M':Magma = false /\ S:Set <= M':Magma = true
  ccp 24 and 23
    false
    = M:Magma <= M':Magma
    if S:Set <= M':Magma = true /\ S:Set <= M':Magma = false
```

Therefore, we are left with only two critical pairs:

```
    ccp for 13 and 17
      true
      = false
      if S:Set in {S:Set} = false .
    ccp for 13 and 18
      true
      = false
      if S:Set in {M:Magma, S:Set} = false .
```

These critical pairs are neither context-joinable nor unfeasible, and at present we do not have methods to eliminate them. But we can still introduce new equations, that should be deducible from the specification, or replace the ones we have, to eliminate the critical pairs.

Let us start with the first critical pair. Given its condition, we may think about adding the following equation 25.

```
(fmod HF-SETS-EXTRAS-1 is
   pr HF-SETS-INC .
   var S : Set .
   eq [25]: S in {S} = true .
 endfm)
```

The tool gives us two critical pairs again, although in this case one of them is unconditional.

```
 Maude> (check Church-Rosser HF-SETS-EXTRAS-1 .)

 Church-Rosser checking of HF-SETS-EXTRAS-1
 Checking solution:
 The following critical pairs cannot be joined:
   cp for 08 and 25
     #1:Magma in {#1:Magma}
     = true .
   ccp for 13 and 18
     true
     = false
     if S:Set in {M:Magma, S:Set} = false .
 The specification is sort-decreasing.
```

We can try with the same equation but for magmas.

```
(fmod HF-SETS-EXTRAS-2 is
   pr HF-SETS-INC .
   var M : Magma .
   eq [25']: M in {M} = true .
 endfm)
```

The critical pairs are avoided, but two new ones are generated when trying to check the Church-Rosser property.

```
 Maude> (check Church-Rosser HF-SETS-EXTRAS-2 .)

 Church-Rosser checking of HF-SETS-EXTRAS-2
 Checking solution:
 The following critical pairs cannot be joined:
   ccp for 11 and 25'
     #1:Magma in {#1:Magma, #2:Set}
```

```
    = true
    if #2:Set in {#1:Magma, #2:Set} = true .
  ccp for 12 and 25'
    false
    = true
    if #2:Set in {#1:Magma, #2:Set} = false .
The specification is sort-decreasing.
```

Perhaps we can have more luck by trying something more ambitious.

```
(fmod HF-SETS-EXTRAS-3 is
   pr HF-SETS-INC .
   vars M M' : Magma .
   eq [26]: M in {M, M'} = true .
 endfm)
```

But we now get three critical pairs:

```
Maude> (check Church-Rosser HF-SETS-EXTRAS-3 .)

Church-Rosser checking of HF-SETS-EXTRAS-3
Checking solution:
The following critical pairs cannot be joined:
  cp for 02 and 26
    M':Magma in {M':Magma}
    = true .
  cp for 08 and 26
    #1:Magma in M':Magma, {#1:Magma}
    = true .
  ccp for 13 and 17
    true
    = false
    if S:Set in {S:Set} = false .
The specification is sort-decreasing.
```

Both `25'` and `26` seem to be useful. Let us try with both.

```
(fmod HF-SETS-EXTRAS-4 is
   pr HF-SETS-INC .
   vars M M' : Magma .
   eq [25']: M in {M} = true .
   eq [26]: M in {M, M'} = true .
 endfm)
```

Now we get just one critical pair.

```
Maude> (check Church-Rosser HF-SETS-EXTRAS-4 .)

Church-Rosser checking of HF-SETS-EXTRAS-4
Checking solution:
The following critical pairs cannot be joined:
  cp for 08 and 26
    #1:Magma in M':Magma, {#1:Magma}
    = true .
The specification is sort-decreasing.
```

Let us add the critical pair as an equation.

```
(fmod HF-SETS-EXTRAS-5 is
   pr HF-SETS-INC .
   vars M M' : Magma .
   eq [25']: M in {M} = true .
   eq [26]: M in {M, M'} = true .
   eq [27]: M in {M}, M' = true .
 endfm)
```

The new equation 27 overlaps with equation 11 giving a new critical pair.

```
Maude> (check Church-Rosser HF-SETS-EXTRAS-5 .)

Church-Rosser checking of HF-SETS-EXTRAS-5
Checking solution:
The following critical pairs cannot be joined:
  ccp for 11 and 27
    #2:Magma in M':Magma, {#2:Magma, #1:Set}
    = true
    if #1:Set in M':Magma, {#1:Set, #2:Magma} = true .
The specification is sort-decreasing.
```

Let us try adding one more equation.

```
(fmod HF-SETS-EXTRAS-6 is
   pr HF-SETS-INC .
   vars M M' M'' : Magma .
   eq [25']: M in {M} = true .
   eq [26]: M in {M, M'} = true .
   eq [27]: M in {M}, M' = true .
   eq [28]: M in {M, M'}, M'' = true .
 endfm)
```

Finally, we have managed to eliminate all critical pairs.

```
Maude> (check Church-Rosser HF-SETS-EXTRAS-6 .)

Church-Rosser checking of HF-SETS-EXTRAS-6
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

Once proved operationally terminating, we can conclude that the HF-SETS-EXTRAS-6 module is confluent.

3.4 Lists and Sets

Let us consider the following specification of lists and sets.

```
(fmod LIST&SET is
   sorts MBool Nat List Set .
   subsorts Nat < List Set .
   ops true false : -> MBool .
   ops _and_ _or_ : MBool MBool -> MBool [assoc comm] .
   op 0 : -> Nat .
   op s_ : Nat -> Nat .
   op _;_ : List List -> List [assoc] .
   op null : -> Set .
   op __ : Set Set -> Set [assoc comm id: null] .
   op _in_ : Nat Set -> MBool .
   op _==_ : List List -> MBool [comm] .
   op list2set : List -> Set .
   var  B : MBool .
   vars N M : Nat .
   vars L L' : List .
   var  S : Set .
   eq [LS01]: N N = N .
   eq [LS02]: true and B = B .
   eq [LS03]: false and B = false .
   eq [LS04]: true or B = true .
   eq [LS05]: false or B = B .
   eq [LS06]: 0 == s N = false .
   eq [LS07]: s N == s M = N == M .
   eq [LS08]: N ; L == M = false .
   eq [LS09]: N ; L == M ; L' = (N == M) and L == L' .
   eq [LS10]: L == L = true .
   eq [LS11]: list2set(N) = N .
   eq [LS12]: list2set(N ; L) = N list2set(L) .
   eq [LS13]: N in null = false .
   eq [LS14]: N in M S = (N == M) or N in S .
 endfm)
```

It has four sorts: `MBool`, `Nat`, `List`, and `Set`, with `Nat` included in both `List` and `Set` as a subsort. The terms of each sort are, respectively, Booleans, natural numbers (in Peano notation), lists of natural numbers, and finite sets of natural numbers. The rewrite rules in this module then define various functions such as `_and_` and `_or_`, a function `list2set` associating to each list its corresponding set, the set membership predicate `_in_`, and an equality predicate `_==_` on lists. Furthermore, the idempotency of set union is specified by the first equation. The operators `_and_` and `_or_` have been declared associative and commutative, the list concatenation operator `_;_` has been declared associative, the set union operator `__` has been declared associative, commutative and with `null` as its identity, and the `_==_` equality predicate has been declared commutative using the `comm` keyword.

The tool gives us the following result.

```
Maude> (check Church-Rosser .)

Church-Rosser checking of LIST&SET
Checking solution:
The following critical pairs cannot be joined:
  cp for LS01 and LS14
    N:Nat == M:Nat
    = (N:Nat == M:Nat) or N:Nat == M:Nat .
  cp for LS01 and LS14
```

```
     (N:Nat == M:Nat) or N:Nat in #5:Set
     = (N:Nat == M:Nat) or (N:Nat == M:Nat) or N:Nat in #5:Set .
The specification is sort-decreasing.
```

These critical pairs are completely harmless. They can in act be removed by introducing a idempotency equation for the _or_ operator.

```
(fmod LIST&SET-2 is
   pr LIST&SET .
   var  B : MBool .
   eq [LS15]: B or B = B .
 endfm)
```

The tool now tells us that the specification is locally confluent and sort-decreasing, and since it is terminating (see [24]), we can conclude it is Church-Rosser.

```
Maude> (check Church-Rosser .)

Church-Rosser checking of LIST&SET-2
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

As explained in Section ??, to handle this specification, we apply several transformations on the original module to remove identity attributes and associativity attributes that do not come with commutativity ones. Although these transformations are applied by the tools without user interaction, since new equations may be introduced in our specifications, we explain how the transformations proceed on this example.

The transformation that removes the identity axioms remove the identity attribute of the __ operator and adds two equations to the specification, the identity axiom and a variant for one of the equations defining _in_, given new variables X, Y, and Z of kind [List,Set]:

```
eq [idEq]: null X = X .
eq [LS14]: N in M = (N == M) or N in null .
```

Notice that identity equations are introduced with label idEq, and variants of existing equations take the labels of the equations they come from.

Regarding the associativity of _;_, we consider one orientation of the associativity axiom after another. The only left-hand sides that could potentially be narrowed with the rule (X ; Y) ; Z = X ; (Y ; Z) are the left-hand sides of the rules

```
eq [LS12]: list2set(N ; L) = N list2set(L) .
eq [LS08]: N ; L == M = false .
eq [LS09]: N ; L == M ; L' = (N == M) and L == L' .
```

However, no such narrowing steps are possible, since the terms N ; L and (X ; Y) ; Z have *no* order-sorted unifiers (likewise for M ; L'). The transformed module is therefore obtained by removing the *assoc* attribute from _;_, and by adding the following associativity rule, given new variables X, Y, and Z of kind [List,Set].

```
eq [assocEq]: (X ; Y) ; Z = X ; (Y ; Z) .
```

Notice that associativity equations are introduced with label `assocEq`.

Once the only combinations of axioms of operators remaining in the module are C and AC, the tool proceeds as for any other module.

3.5 Booleans

Let us consider the following specification of the booleans with a sort `Bool`, constants `true` and `false`, and operators `_and_`, `_or_`, and `not_`.

```
(fmod MBOOL is
   op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
   op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
   op not_ : Bool -> Bool [prec 53] .
   vars A B C : Bool .
   eq true and A = A .
   eq false and A = false .
   eq not true = false .
   eq not false = true .
   eq not not A = A .
   eq A or B = not (not A and not B) .
 endfm)
```

Notice the definition of the operators `_and_` and `_or_` as associative and commutative. The interaction with the CRC is as follows:

```
Maude> (check Church-Rosser .)

Church-Rosser checking of MBOOL
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

The CRC says that the specification is locally confluent. Since the specification also is terminating (it can be proved using the MTT [23]), we can conclude that it is confluent. And since it is also sort decreasing, it is Church-Rosser.

3.6 Another Specification of the Natural Numbers

We give now a specification for the natural numbers quite similar to the one presented in the manual of OBJ3 ([35], page 77), although with some modifications. In particular, instead of using the built-in `if_then_else_fi` and `_==_` operators for the definitions of `quot` and `gcd`, we split their equations for the different cases, making some of them conditional. The following `MNAT` module includes the specification of the booleans given in Section 3.5.

```
(fmod MNAT is
   pr MBOOL .
   sorts Nat NzNat Zero .
```

```
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [comm assoc] .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  ops _>_ : Nat Nat -> Bool .
  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  [nat01] : p s N = N .
  eq  [nat02] : N + 0 = N .
  eq  [nat03] : (s N) + (s M) = s s (N + M) .
  eq  [nat04] : N * 0 = 0 .
  eq  [nat05] : 0 * N = 0 .
  eq  [nat06] : (s N) * (s M) = s (N + (M + (N * M))) .
  eq  [nat07] : 0 > M = false .
  eq  [nat08] : N' > 0 = true .
  eq  [nat09] : s N > s M = N > M .
  eq  [nat10] : d(0, N) = N .
  eq  [nat11] : d(s N, s M) = d(N, M) .
  ceq [nat12] : quot(N, M')
    = s quot(d(N, M'), M')
    if N > M' = true .
  eq  [nat13] : quot(M', M') = s 0 .
  ceq [nat14] : quot(N, M') = 0 if M' > N = true .
  eq  [nat15] : gcd(N', N') = N' .
  ceq [nat16] : gcd(N', M')
    = gcd(d(N', M'), M')
    if N' > M' = true .
endfm)
```

The Church-Rosser check returns the following result:

```
Maude> (check Church-Rosser .)

Church-Rosser checking of MNAT
Checking solution:
The following critical pairs cannot be joined:
  cp for nat03 and nat03
    #8:[Nat]+ s s(M:Nat + s(N:Nat + #2:Nat))
    = #8:[Nat]+ s s(#2:Nat + s(N:Nat + M:Nat)).
  cp for nat03 and nat03
    s s(M:Nat + s(N:Nat + #2:Nat))
    = s s(#2:Nat + s(N:Nat + M:Nat)).
  ccp for nat12 and nat13
    s quot(d(M':NzNat,M':NzNat),M':NzNat)
    = s 0
    if M':NzNat > M':NzNat = true .
  ccp for nat12 and nat14
    s quot(d(N:Nat,M':NzNat),M':NzNat)
    = 0
    if M':NzNat > N:Nat = true /\ N:Nat > M':NzNat = true .
  ccp for nat13 and nat14
    s 0
    = 0
    if M':NzNat > M':NzNat = true .
  ccp for nat14 and nat12
    0
    = s quot(d(N:Nat,M':NzNat),M':NzNat)
    if N:Nat > M':NzNat = true /\ M':NzNat > N:Nat = true .
  ccp for nat15 and nat16
    M':NzNat
    = gcd(M':NzNat,d(M':NzNat,M':NzNat))
    if M':NzNat > M':NzNat = true .
  ccp for nat16 and nat16
    gcd(N':NzNat,d(N':NzNat,M':NzNat))
    = gcd(M':NzNat,d(N':NzNat,M':NzNat))
    if N':NzNat > M':NzNat = true /\ M':NzNat > N':NzNat = true .
There are non-sort-decreasing equations.
The following proof obligations must be checked:
  cmb gcd(M':NzNat,d(N':NzNat,M':NzNat)): NzNat
    if N':NzNat > M':NzNat = true .
```

We have eight critical pairs, out of which six are conditional, and one proof obligation for the descent property. Let us study them.

The first two come from equation `nat03`, and the critical pairs indicate that terms of the form `N + s M` cannot be reduced for `N` and `M` variables of sort `Nat`. They vanish if equation `nat03` is replaced by the following one:

```
    eq  [nat03-1] : s N + M = s (N + M) .
```

The Church-Rosser checker gives the following output with this new module:

```
  Church-Rosser checking of MNAT-2
  Checking solution:
  The following critical pairs cannot be joined:
    ccp for nat12 and nat13
      s quot(d(M':NzNat,M':NzNat),M':NzNat)
      = s 0
      if M':NzNat > M':NzNat = true .
    ccp for nat12 and nat14
      s quot(d(N:Nat,M':NzNat),M':NzNat)
      = 0
      if M':NzNat > N:Nat = true /\ N:Nat > M':NzNat = true .
    ccp for nat13 and nat14
      s 0
      = 0
      if M':NzNat > M':NzNat = true .
    ccp for nat14 and nat12
      0
      = s quot(d(N:Nat,M':NzNat),M':NzNat)
      if N:Nat > M':NzNat = true /\ M':NzNat > N:Nat = true .
    ccp for nat15 and nat16
      M':NzNat
      = gcd(M':NzNat,d(M':NzNat,M':NzNat))
      if M':NzNat > M':NzNat = true .
    ccp for nat16 and nat16
      gcd(N':NzNat,d(N':NzNat,M':NzNat))
      = gcd(M':NzNat,d(N':NzNat,M':NzNat))
      if N':NzNat > M':NzNat = true /\ M':NzNat > N':NzNat = true .
  There are non-sort-decreasing equations.
  The following proof obligations must be checked:
    cmb gcd(M':NzNat,d(N':NzNat,M':NzNat)): NzNat
      if N':NzNat > M':NzNat = true .
```

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 that are neither unfeasible nor context-joinable but have 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 for nat13 and nat14
      s 0
      = 0
      if M':NzNat > M':NzNat = 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 for nat12 and nat13
      s quot(d(M':NzNat, M':NzNat), M':NzNat)
      = s 0
      if M':NzNat > M':NzNat = true .
    ccp for nat15 and nat16
      M':NzNat
      = gcd(M':NzNat, d(M':NzNat, M':NzNat))
      if M':NzNat > M':NzNat = true .
```

But we can still do something. Notice, for example, that in these three critical pairs the conditions are of the form `M' > M'`. Let us introduce the equation

```
  eq  [nat09-1] : N > N = false .
```

The complete module is then as follows.

```
 (fmod MNAT-3 is
    pr MNAT-2 .
    var  N : Nat .
    eq [nat09-1] : N > N = false .
  endfm)
```

And the output given by the tool is the following.

```
Maude> (check Church-Rosser .)

Church-Rosser checking of MNAT-3
Checking solution:
The following critical pairs cannot be joined:
  ccp for nat12 and nat14
    s quot(d(N:Nat,M':NzNat),M':NzNat)
    = 0
    if M':NzNat > N:Nat = true /\ N:Nat > M':NzNat = true .
  ccp for nat14 and nat12
    0
    = s quot(d(N:Nat,M':NzNat),M':NzNat)
    if N:Nat > M':NzNat = true /\ M':NzNat > N:Nat = true .
  ccp for nat16 and nat16
    gcd(N':NzNat,d(N':NzNat,M':NzNat))
    = gcd(M':NzNat,d(N':NzNat,M':NzNat))
    if N':NzNat > M':NzNat = true /\ M':NzNat > N':NzNat = true .
There are non-sort-decreasing equations.
The following proof obligations must be checked:
  cmb gcd(M':NzNat,d(N':NzNat,M':NzNat)): NzNat
    if N':NzNat > M':NzNat = true .
```

The three critical pairs have vanished.

The conditions of the remaining 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.

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.

The membership assertion can be proved using the ITP. Notice that since `N'` is strictly greater than `M'`, and both are greater than zero, the greater common denominator of `M'` and the difference between `N'` and `M'` must be positive.

Let us study now the membership proof obligation

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

It comes from the equation

```
  ceq [nat16] : 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, we lack at present proof methods to handle the joinability assumption hidden in the equality of the condition.

4 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 [5, 15, 13, 14, 36], the Maude Termination Tool [23], the Knuth-Bendix completion tool [22], and the coherence checker and completion tools [20].

The present experience suggests that the reflective approach that we have taken in building the present Maude formal 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 various other tools are reviewed in [11, Chapter 21].

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* [43], 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] *Proceedings of the CafeOBJ Symposium'98*, April 1998.

[2] Jürgen Avenhaus and Carlos Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In Frank Pfenning, editor, *Logic Programming and Automated Reasoning, 5th International Conference, LPAR'94, Kiev, Ukraine, July 16-22, 1994, Proceedings*, volume 822 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 1994.

[3] 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, 1980.

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

[5] Manuel Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Lecture Notes. CSLI Publications, 2000.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Unification and narrowing in Maude 2.4. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

[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 Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 3–24. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications, 10th International Conference, RTA'99, Trento, Italy, July 2–4, 1999, Proceedings*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer, 1999.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87, Berlin, June 2003. Springer.

[11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic.* Lecture Notes in Computer Science. Springer, 2007.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude 2.4 manual. Available in `http://maude.cs.uiuc.edu`, November 2008.

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

[14] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Technical report, Computer Science Laboratory, SRI International, March 1998. `http://maude.csl.sri.com/papers`.

[15] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection in rewriting logic. In Kokichi Futatsugi, Ataru T. Nakagawa, and Tetsuo Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 1–31. Elsevier, 2000. `http://maude.csl.sri.com/papers`.

[16] Manuel Clavel, Francisco Durán, Joe Hendrix, Salvador Lucas, José Meseguer, and Peter Ölveczky. The Maude formal tool environment. In Till Mossakowski, Ugo Montanari,

and Magne Haveraaen, editors, *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007, Bergen, Norway, August 20-24, 2007, Proceedings*, volume 4624 of *Lecture Notes in Computer Science*, pages 173–178. Springer, 2007.

[17] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245 – 288, 2002.

[18] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[19] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Spain, June 1999. `http://maude.csl.sri.com/papers`.

[20] Francisco Durán. Coherence checker and completion tools for Maude specifications. Technical Report ITI-2000-7, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.cs.uiuc.edu`.

[21] Francisco Durán. The extensibility of Maude's module algebra. In Teodor Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2000.

[22] Francisco Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Technical Report ITI-2000-6, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.cs.uiuc.edu`.

[23] Francisco Durán, Salvador Lucas, and José Meseguer. MTT: The Maude termination tool (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.

[24] Francisco Durán, Salvador Lucas, and José Meseguer. Termination modulo combinations of equational theories. In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2009.

[25] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Manuscript, SRI International. Available at `http://maude.cs.uiuc.edu`, February 1999.

[26] Francisco Durán and José Meseguer. A Church-Rosser checker tool for Maude equational specifications. Technical Report ITI-2000-5, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.cs.uiuc.edu`.

[27] Francisco Durán and Peter Csaba Ölveczky. A guide to extending Full Maude illustrated with the implementation of Real-Time Maude. In Grigore Roşu, editor, *Proceedings 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[28] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.

[29] Santiago Escobar, Catherine Meadows, and José Meseguer. Equational cryptographic reasoning in the Maude-NRL protocol analyzer. *Electronic Notes in Theoretical Computer Science*, 171(4):23–36, 2007.

[30] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and extreme termination. Technical Report UIUCDCS-R-2009-3049, University of Illinois, March 2009.

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

[32] Jürgen Giesl and Deepak Kapur. Dependency pairs for equational rewriting. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2001.

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

[34] 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.

[35] 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.

[36] Joe Hendrix. *Decision Procedures for Equationally Based Reasoning.* PhD thesis, University of Illinois at Urbana-Champaign, 2008. Available at `http://hdl.handle.net/2142/11487`.

[37] Joe Hendrix, José Meseguer, and Hitoshi Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In Ulrich Furbach and Natarajan Shankar, editors, *A Sufficient Completeness Checker for Linear Order-Sorted Specifications Modulo Axioms*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[38] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[39] 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.

[40] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. M. Gabbay, editor, *What is a Logical System?*, volume 4 of *Studies in Logic and Computation*, pages 355–392. Oxford University Press, 1994.

[41] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002.

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

[43] José Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence*, Fort Laurerdale, Florida, January 1998. `http://rutcor.rutgers.edu/~amai/Proceedings.html`. Presented also at the *14th IMACS World Congress*, Atlanta, Georgia, July 1994.

[44] E. Ohlebusch. *Advanced Topics in Term Rewriting.* Springer, 2002.

[45] Peter Csaba Ölveczky and José Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[46] Gerald Peterson and Mark Stickel. Complete sets of reductions for some equational theories. *Journal of ACM*, 28(2):233–264, 1981.

[47] Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

A Commands Available in the CRC Tool

(help .) shows the list of commands available in the tool.

(check Church-Rosser .) checks the Church-Rosser property of the default module.

(check Church-Rosser <module-name> .) checks the Church-Rosser property of the specified module.

(show CRC critical pairs .) shows the reduced form of the critical pairs that could not be joined in the last `check Church-Rosser` command. Those joined are not shown.

(show all CRC critical pairs .) shows the unreduced form of all critical pairs computed in the last `check Church-Rosser` command.