Coherence Checker and Completion Tools for Maude Specifications

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

July 28, 2000

Abstract

This document explains the design and use of a coherence checker tool and of a coherence completion tool. The coherence checker tool checks whether a rewrite logic specification is coherent, and the coherence completion tool tries to complete a rewrite logic specification in order to make it coherent. These tools can be used to prove the coherence property or to coherence complete order-sorted rewrite specifications in Maude [7, 5, 2]. The tools have been written entirely in Maude and are in fact executable specifications in rewriting logic [14] of the formal inference system that they implement. The fact that rewriting logic is reflective [8, 1], and that Maude efficiently supports reflective rewriting logic computations [3, 2] is systematically exploited in the design of the tools.

Contents

1 Introduction

Traditionally, a rewrite system is a set of directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest possible form is obtained. This interpretation of a rewrite system gives an equational semantics to a rewrite system, and also a way of executing equational specifications by rewriting. But rewriting is also useful for specifying nonequational relations, such as transitions between states or reduction steps. Rewriting logic [14] suggests keeping all rules with an equational interpretation as a set E of equations, and considering the remaining rules as defining rewrite steps over classes modulo E.

A rewrite logic signature is an equational specification. But, rewriting logic is parameterized by the choice of its underlying equational logic. For Maude [2], the underlying equational logic is membership equational logic, so that signatures are of the form (Ω, E), where $\Omega = (K, \Sigma, S)$ is a membership equational logic signature and E is a set of (conditional) membership axioms and equations. Such a signature (Ω, E) makes explicit the set of equations in order to emphasize that rewriting will operate on congruence classes of terms *modulo* E. This is precisely what Maude does, using the equational attributes given in operation declarations (associativity, commutativity, and identity) to rewrite modulo such axioms (see [2]).

Given a signature (Ω, E), sentences of the logic are sequents of the form $[t]_E \longrightarrow [t']_E$, where t and t' are Ω-terms possibly involving some variables.

A *rewriting logic specification* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a signature, L is a set of *labels*, and R is a set of labeled rewrite rules written

$$r : [t]_E \longrightarrow [t']_E,$$

where r is a label and $[t]_E$ and $[t']_E$ are congruence classes of terms in $\mathcal{T}_{\Omega,E}(X)$, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables.

The logic and its theoretical results can be extended to more expressive conditional rules [14] of the form

$$r : [t]_E \longrightarrow [t']_E \ \ \textit{if} \ \ [u_1]_E \longrightarrow [v_1]_E \ \wedge \ldots \wedge \ [u_k]_E \longrightarrow [v_k]_E,$$

but in the current implementation of Maude [2] only Boolean conditions are supported for rules of the form $r : [t]_E \longrightarrow [t']_E$ *if* b, where b is a Boolean term. A future Maude release will support general rules with rewrites, equations, and membership axioms in their conditions.

A rewrite theory has both rules and equations, so that rewriting is performed modulo such equations. However, this does not mean that the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary theory is undecidable. What Maude instead requires for rewrite theories in system modules is that:

- The equations are divided into a set A of structural axioms, for which matching algorithms exist in the Maude implementation, and a set E of equations that are Church-Rosser and terminating modulo A; that is, the equational part must be equivalent to a functional module. For some axioms A, this can be checked using a Church-Rosser checker as the one presented in [12, 9, 6].

- The rules R in the module are *coherent* [15, 16] with the equations E modulo A. This means that appropriate critical pairs exist between rules and equations, allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A. In particular, a simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R. This is precisely the strategy adopted by the Maude interpreter.

The purpose of the coherence checker tool discussed in this document is to check whether a rewrite theory is coherent or not. The purpose of the coherence completion tool is to complete a noncoherent specification so that it becomes coherent.

The present tools only handle conditional order-sorted specifications, where each of the operation symbols has either no equational attributes, or only the commutativity attribute. Furthermore, it is assumed that such specifications do not contain any built-in sorts or functions, and that the equational part has already been proved terminating and Church-Rosser.

The coherence checker tool attempts to establish the coherence property *modulo* the commutativity of some of the operators by checking a sufficient condition. The tool's output consists of a set of critical pairs that must be shown ground-joinable. Regarding the coherence completion tool, it tries to complete, also *modulo* the commutativity of some of the operators, by including the critical pairs oriented as rules. In this case, the output of the tool, if any, is a coherent specification.

After introducing some basic formal concepts and results underlying the tools' design, we explain key parts of each of the tools and give some examples. We present the different modules specifying the tools with their main declarations, so that a general idea of the whole system can be gained, omitting declarations of operators not relevant for the general discussion, as well as equations describing the semantics of some functions. We add dots "..." to indicate the missing equations and declarations.

Acknowledgements

The author is grateful to José Meseguer for his advice and collaboration in the long-term joint project of which this work is a part. We are very thankful for the many fruitful discussions with the other members of the Maude team, Manuel Clavel, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Quesada, with whom we have designed and built the Maude language and system.

2 Coherent Specifications

Given a framework as rewriting logic, in which we have a set of equations executed by rewriting, which can also be seen as a set of rules interpreted equationally, and a set of rewrite rules without an equational interpretation, the property of *coherence* ensures a correspondence between operational behavior and intended semantics [15, 16].

Given a set of rules $R = \{l_i \longrightarrow r_i\}$, the rewrite relation $\longrightarrow_R$ is the smallest relation stable by context and instantiation containing $l_i \longrightarrow_R r_i$ for each $l_i \longrightarrow r_i \in R$. For any relations $\longrightarrow_R$ and $\longrightarrow_S$, we define $\longrightarrow_{R/S} = \longrightarrow_S^* \longrightarrow_R \longrightarrow_S^*$. In particular, when $\longrightarrow_S$ is the replacement relation $\longleftrightarrow_S$ generated by a set E of equations, $\longrightarrow_{R/E}$ is the basis for defining rewriting over classes modulo E:

$$[t]_E \longrightarrow_{[R]_E} [t']_E \quad \textit{iff} \quad t \longrightarrow_{R/E} t'.$$

Finally, we define $\longrightarrow_{[R(S)]_E}$ as $\longrightarrow_{[R(S)]_E} = \longrightarrow_{[S]_E}^* \longrightarrow_{[R]_E}$.

Let a rewrite theory $\mathcal{R}$ be a 4-tuple $\mathcal{R} = (\Omega, A \cup E, L, R)$, with A a set of structural axioms, E a set of equations that are Church-Rosser and terminating modulo A, L a set of labels, and R a set of labeled rewrite rules.

Although the axioms in E have an equational interpretation, they will be used as reduction rules from left to right. This leads to two different views:

- An operational view, in which we consider sequents of the rewrite theory $(A, E \cup R)$:
$$[t]_A \longrightarrow_{[R(E)]_A} [t']_A$$
- A semantic view, in which we consider sequents of the rewrite theory $(A \cup E, R)$:
$$[t]_{A \cup E} \longrightarrow_{[R]_{A \cup E}} [t']_{A \cup E}$$

$[t]_A \longrightarrow_{[R(E)]_A} [t']_A$ implies $[t]_{A \cup E} \longrightarrow_{[R]_{A \cup E}} [t']_{A \cup E}$, but the opposite is not true in general. The notion of coherence allows as to establish the grade of satisfiability of the relation in the opposite direction. There are different notions of coherence, which correspond to different correspondences between $[t]_A \longrightarrow_{[R(E)]_A} [t']_A$ and $[t]_{A \cup E} \longrightarrow_{[R]_{A \cup E}} [t']_{A \cup E}$.

We focus here on *equational coherence*, which states that the set of normal forms is preserved, except possibly for some cycles. See [17] for a discussion on the different types of coherence, where some interesting results related to coherence are given.

Equational coherence is the most important of the coherence properties, since it can be checked by looking at critical pairs. Weak coherence and strong coherence may be more difficult to establish.

Definition 1 $\longrightarrow_{[R(E)]_A}$ *is equational coherent if*

$$\begin{tikzcd}
& {} \arrow[dl, "{[E]_A}"', "*"] \arrow[dr, "{[R(E)]_A}"] & \\
{} \arrow[d, dashed, "{[R(E)]_A}"', "*"] & & {} \arrow[d, dashed, "{[R(E)]_A}", "*"'] \\
{} \arrow[dr, dashed, "{[E]_A}"', "*"] & & {} \arrow[dl, dashed, "{[E]_A}", "*"'] \\
& {} &
\end{tikzcd}$$

Theorem 2 *(From [17]) If $\longrightarrow_{[R(E)]_A}$ is equational coherent, then if $[t]_E$ is a normal form for $\longrightarrow_{[R(E)]_A}$, then either $[t]_{A\cup E}$ is a normal form for $\longrightarrow_{[R]_{A\cup E}}$, or there is a nonempty cycle $[t]_{A\cup E} \longrightarrow_{[R]_{A\cup E}} [t]_{A\cup E}$.*

In the case of equational coherence the set of normal forms is not exactly preserved, since a cycle in $\longrightarrow_{[R]_{A\cup E}}$ may be matched by an empty derivation of $\longrightarrow_{[R(E)]_A}$.

3 Coherence Check

In order to be able to check coherence syntactically by looking at the rewrite systems R and E it is useful to consider the concept of local coherence.

Definition 3 $\longrightarrow_{[R]_A}$ *is local equational coherent with* $\longrightarrow_{[E]_A}$ *if*

$$\begin{tikzcd}
& {} \arrow[dl, "{[E]_A}"'] \arrow[dr, "{[R(E)]_A}"] & \\
{} \arrow[d, dashed, "{[R(E)]_A}"', "*"] & & {} \arrow[d, dashed, "{[R(E)]_A}", "*"'] \\
{} \arrow[dr, dashed, "{[E]_A}"', "*"] & & {} \arrow[dl, dashed, "{[E]_A}", "*"'] \\
& {} &
\end{tikzcd}$$

The following theorem relates local equational coherence with equational coherence.

Theorem 4 *(From [17]) If* $\longrightarrow_{[E]_A}$ *and* $\longrightarrow_{[R\cup E]_A}$ *are terminating, then* $\longrightarrow_{[R(E)]_A}$ *is equational coherent if and only if* $\longrightarrow_{[R]_A}$ *is local equational coherent with* $\longrightarrow_{[E]_A}$.

Note the hypothesis of termination of $\longrightarrow_{[R\cup E]_A}$.

Now, checking equational coherence reduces to ensuring that for all critical peaks ${}_{[E]_A}\swarrow \searrow_{[R(E)]_A}$, the definition of local equational coherence is verified.

3.1 The Maude Specification of a Coherence Checker

The design and specification of our coherence checker tool follow quite closely the design and specification of the Church-Rosser checker tool presented in [6, 9, 11, 12]. In fact, some of the modules are common to both tools, so we refer to the discussions on them in [6]. In particular, we assume the general operations on terms [12, Section 3.1.1], order-sorted unification and matching [12, Section 3.1.2], the `CRITICAL-PAIR` module [12, Section 3.2], which includes some basic definitions on critical pairs, and the `MODULE-PREPARATION` module [12, Section 3.4], which includes basic operations for preparing the modules to be checked.

3.1.1 The Coherence Check

The main difference with respect to the Church-Rosser checker tool is that the critical pairs to check confluency are obtained by superposing the equations of the functional specification given as input, while when checking coherence we superpose each of the equations with each of the rules. Thus, in the following module `COHERENCE-CHECK`, we define the `cohCritPairs` function, which mirrors quite closely the function `critPairs` in the module `CONFLUENCE-CHECK` in [12, Section 3.4]. As in [12], we only give here the equations for the unconditional case.

```
fmod COHERENCE-CHECK is
  pr UNIFICATION .
  pr EXT-TERM .
  pr CRITICAL-PAIR .
  pr MODULE-PREPARATION .

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

```
op cohCritPairs : Module -> CritPairSet .
op cohCritPairs1 : Module EquationSet RuleSet -> CritPairSet .
op cohCritPairs2 : Module Equation RuleSet -> CritPairSet .
op cohCritPairs3 : Module Equation Rule PositionSet
      -> CritPairSet .
op cohCritPairs4 : Module Equation Rule Position SubstitutionSet
      -> CritPairSet .

op lhs : Equation -> Term .
op lhs : Rule -> Term .
eq lhs(eq T = T' .) = T .
eq lhs(rl [L] : T => T' .) = T .

eq cohCritPairs(M) = cohCritPairs1(M, equationSet(M), ruleSet(M)) .

eq cohCritPairs1(M, ((eq T = T' .) EqS), RlS)
  = critPairSet(
      cohCritPairs1(M, EqS, RlS),
      cohCritPairs2(M, (eq rename(M, T) = rename(M, T') .), RlS)) .
eq cohCritPairs1(M, none, RlS) = emptyCritPairSet .

eq cohCritPairs2(M, Eq, (Rl RlS))
  = critPairSet(
      cohCritPairs2(M, Eq, RlS),
      cohCritPairs3(M, Eq, Rl, allNonVarPos(M, lhs(Rl)))) .
eq cohCritPairs2(M, Eq, none) = emptyCritPairSet .

eq cohCritPairs3(M, Eq, Rl, posSet(P, PS))
  = critPairSet(
      cohCritPairs3(M, Eq, Rl, PS),
      cohCritPairs4(M, Eq, Rl, P,
         unify(M, getSubterm(lhs(Rl), P), lhs(Eq)))) .
eq cohCritPairs3(M, Eq, Rl, emptyPosSet) = emptyCritPairSet .

eq cohCritPairs4(M, (eq T1 = T1' .), (rl [L] : T2 => T2' .), P,
     substitutionSet(Subst, SubstS))
  = critPairSet(
      cp(replace(substitute(T2, Subst), substitute(T1', Subst), P),
         substitute(T2', Subst)),
      cohCritPairs4(M, (eq T1 = T1' .),
         (rl [L] : T2 => T2' .), P, SubstS)) .
eq cohCritPairs4(M, Eq, Rl, P, emptySubstitutionSet)
  = emptyCritPairSet .

op cohDelete : CritPairSet -> CritPairSet .

eq cohDelete(critPairSet(cp(T, T'), CPS))
  = if T == T'
    then cohDelete(CPS)
```

```
      else critPairSet(cp(T, T'), cohDelete(CPS))
      fi .
  eq cohDelete(critPairSet(ccp(T, T', T'', {'true}'Bool, CPS))
    = if (T == T') or (T'' == {'false}'Bool)
      then cohDelete(CPS)
      else critPairSet(ccp(T, T', T'', {'true}'Bool), cohDelete(CPS))
      fi .
  eq cohDelete(emptyCritPairSet) = emptyCritPairSet .

  op cohSimplify : CritPairSet Module -> CritPairSet .

  eq cohSimplify(critPairSet(cp(T, T'), CPS), M)
    = critPairSet(cp(meta-rewrite(M, T, 0), meta-rewrite(M, T', 0)),
        cohSimplify(CPS, M)) .
  eq cohSimplify(critPairSet(ccp(T, T', T'', {'true}'Bool), CPS), M)
    = critPairSet(
        ccp(meta-rewrite(M, T, 0), meta-rewrite(M, T', 0),
            meta-rewrite(M, T'', 0), {'true}'Bool),
        cohSimplify(CPS, M)) .
  eq cohSimplify(emptyCritPairSet, M) = emptyCritPairSet .

  op coherence : Module -> CritPairSet .

  eq coherence(M)
    = maximalCPSet(
         cohDelete(
            cohSimplify(
               cohDelete(cohCritPairs(M)), M)), M) .

  sorts CohCheckSol .
  op cohCheckSol : VarDeclSet CritPairSet -> CohCheckSol .

  op coherenceCheck : Module -> CohCheckSol .

  eq coherenceCheck(M)
    = cohCheckSol(
         varDeclSetInCritPairs(prepareModule(M),
            coherence(prepareModule(M))),
         coherence(prepareModule(M))) .
  ...

endfm
```

3.1.2 The Environment of the Tool

To simplify the implementation of an environment for the coherence checker tool, as we already did for other tools [12, 10], we have built it on top of Full Maude [2]. As explained in [9], extensions to Full Maude often require extensions to the signature in which the input will be parsed. In the case of the coherence checker, we add declarations for two new commands, `check coherence .` and `check coherence_.`, to check the coherence of the default specification and of an explicitly specified module, respectively.

```
fmod COHERENCE-CHECK-SIGN is
  including FULL-MAUDE-SIGN .

  op check coherence . : -> PreCommand .
  op check coherence_. : ModExp -> PreCommand .
endfm

fmod META-COHERENCE-CHECK-SIGN is
  inc META-LEVEL .
  pr META-FULL-MAUDE-SIGN .

  op COHERENCE-CHECK-GRAMMAR : -> FModule .

  eq COHERENCE-CHECK-GRAMMAR
    = addImportList((including 'COHERENCE-CHECK-SIGN .),
         EXT-GRAMMAR) .
endfm
```

We extend the persistent database of Full Maude by defining a subclass of `DatabaseClass` inheriting its behavior, and adding new functionalities to it. In this case, we only want to add the new commands `check coherence .` and `check coherence_.`. It is enough to add the declarations making the class `CohCh` a subclass of `DatabaseClass`, and the corresponding rules describing the behavior associated with the new commands.

```
mod COHERENCE-CHECK-DATABASE-HANDLING is
  pr EXT-DATABASE-HANDLING .
  pr COHERENCE-CHECK .

  var ME : ModExp .
  var DB : Database .
  var M : Module .
  var VDS : VarDeclSet .
  var CPS : CritPairSet .
  var MN : ModName .
  var QIL : QidList .

  sort CohCh .
  subsort CohCh < DatabaseClass .

  op CohCh : -> CohCh .

  var Atts : AttributeSet .
  var X@CohCh : CohCh .
  var O : Oid .
  var T : Term .
```

The processing of the checking commands is handled by the following function `processCohChecking`. Its definition is very simple. Given a module expression and a database, it just calls the `coherenceCheck` function presented in Section 2 with the flat form of the specified module in the database, and then gives the output as a list of quoted identifiers ready to be passed to the read-eval-print loop.

```
  op processCohChecking : ModExp Database -> QidList .
  op processCohCheckingAux : Module CohCheckSol -> QidList .

  eq processCohChecking(ME, DB)
    = ('\n 'Coherence 'checking 'of my-meta-pretty-print(ME)
       processCohCheckingAux(getFlatUnit(ME, DB),
         coherenceCheck(getFlatUnit(ME, DB)))) .

  eq processCohCheckingAux(M, cohCheckSol(VDS, CPS))
    = if CPS == emptyCritPairSet
      then ('\n 'Checking 'solution ':
            '\n '\t 'The 'specification 'is 'Coherent '.)
      else ('\n 'Checking 'solution ':
            my-meta-pretty-print(VDS)
            my-meta-pretty-print(addVarDeclSet(VDS, M), CPS))
      fi .
```

Finally, we give the rules describing the behavior of the `CohCh` object associated to the new commands. In both cases the processing is carried out by the `processCohChecking` function, which is called with the specified module name, in the case of the `check coherence_.` command, or with the name of the default module, for the `check coherence .` command.

```
  rl [CohChhecking] :
     < O : X@CohCh | db : DB, input : ({'check`coherence`.}'PreCommand),
        output : QIL, default : MN, Atts >
     => < O : X@CohCh | db : DB, input : nilTermList,
           output : processCohChecking(MN, DB),
           default : MN, Atts > .
  rl [CohChhecking] :
     < O : X@CohCh | db : DB, input : ('check`coherence_.[T]),
        output : QIL, default : MN, Atts >
     => < O : X@CohCh | db : DB, input : nilTermList,
           output : processCohChecking(parseModExp(T), DB),
           default : MN, Atts > .

  ...

endm
```

We finish our specification redefining the `FULL-MAUDE` module.

```
mod COHERENCE-CHECK-FULL-MAUDE is
```

```
pr META-COHERENCE-CHECK-SIGN .
pr COHERENCE-CHECK-DATABASE-HANDLING .
pr PREDEF-UNITS .
inc LOOP-MODE .

subsort Object < State .
```

The constant `cohch` is used as identifier for the object of class `CohCh`.

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

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

For the coherence checker, we need to add a rule initializing the persistent object as an object of classes `CohCh`.

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

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

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

```
            < O : X@CohCh | db : DB,
                 input : meta-parse(COHERENCE-CHECK-GRAMMAR, QIL),
                 output : nil, default : MN, Atts >,
            QIL']
      fi
      if QIL =/= nil .
```

The out rule remains as originally specified.

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

3.2 Some Examples of the Use of the Tool

The use of the coherence checker and the methods suggested for establishing the coherence property of a specification are quite similar to the use and methods suggested for the Church-Rosser checker [9, 12].

3.2.1 The Use of the Tool

We assume that the user has developed an *executable specification* of his/her intended system, and that this specification has already been *tested* with examples, so that the user is in fact confident that the specification is *ground-coherent*, 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 coherent, and not just ground-coherent. That is, not generating any proof obligations is only a *sufficient* condition. But in many cases of interest—particularly for specifications with nontrivial sort orderings—the specification will typically be *ground* coherent, but not coherent, so that a collection of critical pairs 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 is not always appropriate is to let an automatic completion process add new rules to the user's specification in a mindless way. In many cases it will certainly lead to a nonterminating process. But, 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 the examples studied indicate, by analyzing the critical pairs returned, the user can understand why they could not be joined. It may be a mistake that must be corrected. More often, however, it is not a matter of a mistake, but of an axiom that is either *too general*—so that its very generality makes joining the critical pair impossible, because no more equations can apply to it—or *amenable to an equivalent formulation* that is unproblematic 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.

3.2.2 Rewriting Modulo Commutativity with an Identity Equation

Our first example is based on one given by Viry in [17], in which we consider a specification with a constant `0`, a unary operator `f`, and a commutative binary operator `_+_`. We define the operator `_+_` with `0` as its identity element by including the equation `X + 0 = X`, which must be oriented from left to right in order to avoid nontermination. Let us consider then the following specification.

```
(mod IDENTITY is
   sort S .
   op 0 : -> S .
   op _+_ : S  S -> S [comm] .
   op f : S -> S .
   vars X Y : S .
   eq X + 0 = X .
   rl [r1] : f(X + Y) => f(X) + f(Y) .
 endm)
```

The functional part of this specification is Church-Rosser and terminating. However, $[R]_A$ is not coherent with $[E]_A$, as the tool shows.

```
Maude> (check coherence .)
rewrites: 648 in 0ms cpu (10ms real) (~ rewrites/second)

Church-Rosser checking of IDENTITY
Checking solution :
        var X@@S : S .
        cp f ( X@@S ) = f ( 0 ) + f ( X@@S ) .
```

The result of the tool suggests the addition of the critical pair, once appropriately oriented, as a rule.

```
(mod IDENTITY-2 is
   sort S .
   op 0 : -> S .
   op _+_ : S  S -> S [comm] .
   op f : S -> S .
   vars X Y : S .
```

```
   eq X + 0 = X .
   rl [r1] : f(X + Y) => f(X) + f(Y) .
   rl [r2] : f(X) + f(0) => f(X) .
 endm)

Maude> (check coherence .)
rewrites: 968 in 0ms cpu (10ms real) (~ rewrites/second)

Church-Rosser checking of IDENTITY-2
Checking solution :
        The specification is Coherent .
```

However, as suggested by Viry [17], there is a simpler solution.

```
(mod IDENTITY-3 is
   sort S .
   op 0 : -> S .
   op _+_ : S  S -> S [comm] .
   op f : S -> S .
   vars X Y : S .
   eq X + 0 = X .
   rl [r1] : f(X + Y) => f(X) + f(Y) .
   rl [r2'] : f(0) => 0 .
 endm)

Maude> rewrites: 699 in 0ms cpu (10ms real) (~ rewrites/second)

Church-Rosser checking of IDENTITY-3
Checking solution :
        The specification is Coherent .
```

3.2.3 Sequent Calculus for Propositional Logic

Let us consider now the description of the representation within rewriting logic of a sequential calculus for classical propositional logic, as first given in [13] and then in [4], where the basic atoms are denoted by identifiers.

We can take advantage of several properties enjoyed by classical logic in order to have a more succinct and elegant sequent calculus. First, using the double negation and De Morgan laws, we can push negation to the atomic level. Also, we can use one-sided sequents of the form $\vdash A_1, \ldots, A_n$, instead of the more usual double-sided sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, because such a sequent is equivalent to the one-sided sequent $\vdash \sim A_1, \ldots, \sim A_n, B_1, \ldots, B_m$, where the symbol $\sim$ denotes negation. One of the advantages of this presentation is that the number of rules is reduced to a half. Furthermore, the contraction and permutation structural rules can be built into the sequent data structure by using (nonempty) sets of propositions instead of lists.

The first *functional* module defines the syntax of classical propositional logic, using the connectives of conjunction, disjunction, and negation. Equations are used in order to push negation to the atomic level. A one-sided sequent is defined as a set of propositions of the form `|- A1,...,An`.

In [4], `_,_` was declared commutative and associative. Here, we declare it just commutative to be able to use the tool.

```
(fmod PROP-LOG is
  protecting QID .
  sorts Atom Prop .
  subsorts Qid < Atom < Prop .   *** atomic propositions
  ops tt ff : -> Prop .          *** truth values
  op ~_ : Prop -> Prop .         *** negation
  op _\/_ : Prop Prop -> Prop .  *** disjunction
  op _/\_ : Prop Prop -> Prop .  *** conjunction
  vars P Q : Prop .
  eq ~ tt = ff .
  eq ~ ff = tt .
  eq ~ ~ P = P .                  *** double negation
  eq ~ (P \/ Q) = ~ P /\ ~ Q .    *** De Morgan laws
  eq ~ (P /\ Q) = ~ P \/ ~ Q .

  sorts PropSet Sequent .
  subsort Prop < PropSet .
  op _`,_ : PropSet PropSet -> PropSet [comm] .
  op |-_ : PropSet -> Sequent .
  var S : PropSet .
  eq S,S = S .
 endfm)
```

The specification is terminating, and also Church-Rosser, as shows the output of the Church-Rosser checker tool.

```
Maude> (check Church-Rosser .)
rewrites: 8496 in 50ms cpu (60ms real) (169920 rewrites/second)

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

Having defined all the necessary syntax, the following *system module* defines the sort `Configuration` of multisets of sequents representing the premises and conclusion of deduction rules, and then gives the corresponding one-sided sequent calculus rules for classical propositional logic as rewrite rules on the sort `Configuration`. In [4], `__` is declared associative, commutative, and with `empty` as identity element. Here, we declare it just commutative to be able to use the tool.

```
(mod SEQUENT-RULES-PROP-LOG is
  protecting PROP-LOG .
  sort Configuration .
  subsort Sequent < Configuration .
```

```
  op empty : -> Configuration .
  op __ : Configuration Configuration -> Configuration [comm] .
  vars R S : PropSet .
  vars P Q : Prop .

  rl [Identity] :        empty
                   => -----------
                      |- (P, ~ P) .

  rl [Cut] :          |- (R, P) |- (S, ~ P)
                   => ---------------------
                           |- (R, S) .

  rl [Weakening] :      |- R
                   => ---------
                      |- (R, P) .

  rl [Disjunction] :    |- (R, (P, Q))
                   => ----------------
                       |- (R, (P \/ Q)) .

  rl [Conjunction] :  |- (R, P) |- (S, Q)
                   => -------------------
                     |- (R, (S, (P /\ Q))) .

  rl [Truth] :         |- tt
                   => -------
                       empty .
 endm)
```

The result given by the tool is as follows.

```
Maude> (check coherence .)
rewrites: 86681 in 1230ms cpu (1250ms real) (70472 rewrites/second)

Coherence checking of SEQUENT-RULES-PROP-LOG
Checking solution :
        var Q@Prop : Prop .
        var P@Prop : Prop .
        var S@PropSet : PropSet .
        var R@PropSet : PropSet .
        var Q@Prop@@Prop : Prop .
        var P@Prop@@Prop : Prop .
        var S@PropSet@@Prop : Prop .
        cp |- S@PropSet@@Prop |- (Q@Prop, S@PropSet)
          = |- (S@PropSet@@Prop,
                (S@PropSet, (S@PropSet@@Prop /\ Q@Prop))) .
        cp |- S@PropSet@@Prop |- (P@Prop, R@PropSet)
          = |- (R@PropSet,
                (S@PropSet@@Prop \/ (P@Prop /\ S@PropSet@@Prop))) .
```

```
cp |- (Q@Prop, P@Prop)
  = |- ((Q@Prop \/ P@Prop), (P@Prop \/ Q@Prop)) .
cp |- (S@PropSet, (R@PropSet, (ff /\ tt)))
  = |- (S@PropSet, R@PropSet) .
cp |- (R@PropSet,
       (S@PropSet,
        ((P@Prop@@Prop \/ Q@Prop@@Prop)
         /\
         (~ P@Prop@@Prop /\ ~ Q@Prop@@Prop))))
  = |- (S@PropSet, R@PropSet) .
cp |- (R@PropSet,
       (S@PropSet,
        ((P@Prop@@Prop /\ Q@Prop@@Prop)
         /\
         (~ P@Prop@@Prop \/ ~ Q@Prop@@Prop))))
  = |- (S@PropSet, R@PropSet) .
```

The specification is not coherent, as shows the existence of critical pairs. A completion tool would try to add these critical pairs as rules once oriented appropriately. However, as we will see in Section 4.2.2, this process does not terminate.

By observing the critical pairs generated one can very easily guest how to avoid some of them. For example, in the third critical pair given by the tool there is a subterm `ff /\ tt` which cannot be further reduced. Effectively, there are no equations for reducing such a kind of terms. We can add the equations

```
eq tt \/ P = tt .
eq ff \/ P = P .
eq tt /\ P = P .
eq ff /\ P = ff .
```

to the module `PROP-LOG`, and rules

```
rl [rl1] :           |- (ff, R)
                  => ------------
                       |- R .

rl [rl2] :           |- (S, (ff, R))
                  => -----------------
                       |- (S, R) .

rl [rl3] :           |- (tt, R)
                  => ------------
                       |- tt .

rl [rl4] :           |- (S, (tt, R))
                  => -----------------
                       |- tt .
```

to the module `SEQUENT-RULES-PROP-LOG`. Note that rules `rl2` and `rl4` are necessary because `_,_` is not associative.

We first must prove termination of the new PROP-LOG module and then that it satisfies the Church-Rosser property.

```
Maude> (check Church-Rosser .)
rewrites: 19210 in 110ms cpu (120ms real) (174636 rewrites/second)

Church-Rosser checking of PROP-LOG-2
Checking solution :
        The specification is Church-Rosser .
```

We are now ready to check coherence again.

```
Maude> (check coherence .)
rewrites: 113405 in 1320ms cpu (1330ms real) (85912 rewrites/second)

Coherence checking of SEQUENT-RULES-PROP-LOG
Checking solution :
        var Q@Prop : Prop .
        var P@Prop : Prop .
        var S@PropSet : PropSet .
        var R@PropSet : PropSet .
        var Q@Prop@@Prop : Prop .
        var P@Prop@@Prop : Prop .
        var S@PropSet@@Prop : Prop .
        cp |- S@PropSet@@Prop |- (Q@Prop, S@PropSet)
          = |- (S@PropSet@@Prop,
                (S@PropSet, (S@PropSet@@Prop /\ Q@Prop))) .
        cp |- S@PropSet@@Prop |- (P@Prop, R@PropSet)
          = |- (R@PropSet,
                (S@PropSet@@Prop \/ (P@Prop /\ S@PropSet@@Prop))) .
        cp |- (Q@Prop, P@Prop)
          = |- ((Q@Prop \/ P@Prop), (P@Prop \/ Q@Prop)) .
        cp |- (R@PropSet,
               (S@PropSet,
                ((P@Prop@@Prop \/ Q@Prop@@Prop)
                 /\
                 (~ P@Prop@@Prop /\ ~ Q@Prop@@Prop))))
          = |- (S@PropSet, R@PropSet) .
        cp |- (R@PropSet,
               (S@PropSet,
                ((P@Prop@@Prop /\ Q@Prop@@Prop)
                 /\
                 (~ P@Prop@@Prop \/ ~ Q@Prop@@Prop))))
          = |- (S@PropSet, R@PropSet) .
```

The result of the coherence checker gives one critical pair less. In order to make the specification coherent, we have to repeat this process as many times as necessary. By observing the critical pairs generated by the tool we may conclude that the specification is ground-coherent although some critical pairs are still given. It may also happen that this process never terminates, because

$$\textbf{Deduce} \quad \frac{R}{R \cup \{s \to t\}} \quad \text{if } s = t \in \ \mathrm{cp}(R, E)$$

$$\textbf{SimplifyR} \quad \frac{R \cup \{s \to t\}}{R \cup \{s \to u\}} \quad \text{if } t \to^{*}_{E(A)} u$$

$$\textbf{SimplifyL} \quad \frac{R \cup \{s \to t\}}{R \cup \{u \to t\}} \quad \text{if } s \to^{*}_{E(A)} u$$

Figure 1: Inference rules for coherence completion.

the specification cannot be made coherent, but in most of the cases the critical pairs just highlight obvious or trivial modifications and improvements to the original specifications.

4 Coherence Completion

If a rewrite theory is not coherent, that is, if there are critical pairs that cannot be joined, one can try to achieve coherence by adding these critical pairs oriented as rules. This process is very similar to the Knuth-Bendix completion process, although with a major difference. The orientation of the new rules is not given by any reduction order, but is fixed by the orientation of the rules of the theory. We give in Figure 4 the inference rules for coherence completion, where, in addition to the **Deduce** rule that adds new rules, we also have simplification rules **SimplifyL** and **SimplifyR**. Applying these rules as much as possible to an original system R_0 may not terminate, but if it does it provides an equivalent system which is coherent.

Theorem 5 *(From [17]) Suppose that $R_0 \vdash \ldots \vdash R_\infty$ using the inference rules of Figure 4, and that no rule applies to R_∞, then $\longrightarrow_{[R_\infty]_{A \cup E}} = \longrightarrow_{[R_0]_{A \cup E}}$ and $\longrightarrow_{[R_\infty(E)]}$ is equational coherent.*

In order to consider conditional specifications we need to extend the inference rules of Figure 4. Such an extended inference rule is given in Figure 4, where in addition to the **Deduce** rule dealing with conditional critical pairs, we find three simplification rules, namely, **SimplifyL**, **SimplifyR**, and **SimplifyC**. Note that we assume that E is Church-Rosser and terminating.

4.1 The Maude Specification of a Coherence Completion Tool

A coherence completion tool can be implemented in a very straightforward manner by just applying the rules in Figure 4 as much as possible to the system given as input. This process may not terminate, but if it does it provides an equivalent coherent system, which is the output of the tool.

$$
\begin{array}{lcl}
\textbf{Deduce} & \dfrac{R}{R \cup \{s \to t \textit{ if } c\}} & \text{if } s = t \textit{ if } c \in \text{ cp}(R, E) \\
\textbf{SimplifyR} & \dfrac{R \cup \{s \to t \textit{ if } c\}}{R \cup \{s \to u \textit{ if } c\}} & \text{if } t \to^*_{E(A)} u \\
\textbf{SimplifyL} & \dfrac{R \cup \{s \to t \textit{ if } c\}}{R \cup \{u \to t \textit{ if } c\}} & \text{if } s \to^*_{E(A)} u \\
\textbf{SimplifyC} & \dfrac{R \cup \{s \to t \textit{ if } c\}}{R \cup \{s \to t \textit{ if } u\}} & \text{if } c \to^*_{E(A)} u
\end{array}
$$

Figure 2: Inference rules for coherence completion.

The following module `COHERENCE-COMPLETION` specifies this process. In particular, the function `coherence-complete` implements the main loop, in which all critical pairs of the sets of rules and equations of the current module are generated with the `cohCritPairs` function of module `COHERENCE-CHECK` (see Section 3.1.1). Then, the function `reduce+rule` is called with all these critical pairs, which is in charge of simplifying them using the `meta-reduce` function, that is, using the equations of the module, and then transforming them into rules. The process terminates when the process stops generating new rules. Note that since the same equations and rules are used repeatedly to generate critical pairs, and since these are not simplified using the rules—we do not intend to show the confluency of the set of rules—the same critical pairs, and therefore the same rules, can be generated once and again. A procedure similar to Huet's completion procedure could be used to avoid this.

```
fmod COHERENCE-COMPLETION is
  pr COHERENCE-CHECK .
  pr META-LEVEL .

  op coherence-complete : Module -> Module .
  op reduce+rule : Module CritPairSet -> RuleSet .

  var M : Module .
  vars T T' T'' T''' : Term .
  var CPS : CritPairSet .
  vars Rl Rl' : Rule .
  vars RlS RlS' : RuleSet .

  eq reduce+rule(M, critPairSet(cp(T, T'), CPS))
    = ((if meta-reduce(M, T) =/= meta-reduce(M, T')
        then (rl ['r] : meta-reduce(M, T) => meta-reduce(M, T') .)
        else none
        fi)
       reduce+rule(M, CPS)) .
  eq reduce+rule(M, critPairSet(ccp(T, T', T'', T'''), CPS))
```

```
    = ((if meta-reduce(M, T) =/= meta-reduce(M, T')
        then (crl ['r] : meta-reduce(M, T) => meta-reduce(M, T')
                 if meta-reduce(M, T'') = meta-reduce(M, T''') .)
        else none
        fi)
       reduce+rule(M, CPS)) .
  eq reduce+rule(M, emptyCritPairSet) = none .

  eq coherence-complete(M)
    = if (reduce+rule(prepareModule(M), cohCritPairs(prepareModule(M)))
             in ruleSet(prepareModule(M)))
      then M
      else coherence-complete(
              addRuleSet(
                 reduce+rule(prepareModule(M),
                    cohCritPairs(prepareModule(M))),
                 prepareModule(M)))
      fi .

  op _in_ : RuleSet RuleSet -> Bool .
  eq Rl RlS in Rl' RlS'
    = ((Rl == Rl') or (Rl in RlS')) and (RlS in Rl' RlS') .
  eq none in RlS = true .
  eq Rl RlS in none = false .

  eq Rl Rl = Rl .

endfm
```

As we did for the coherence check in Section 3.1.2, we have built an environment for the coherence completion tool on top of Full Maude. In particular, we extend Full Maude with two new commands, `coherence complete .` and `coherence complete_.`, to coherence complete the default specification and the explicitly specified module, respectively.

```
fmod COHERENCE-COMPLETION-SIGN is
  including FULL-MAUDE-SIGN .

  op coherence complete . : -> PreCommand .
  op coherence complete_. : ModExp -> PreCommand .
endfm
```

We extend the persistent database of Full Maude by defining a subclass of `DatabaseClass` inheriting its behavior, and adding new functionalities to it. In this case, we only want to add the new commands `coherence complete .` and `coherence complete_.`. It is enough to add the declarations making the class `CohCompl` a subclass of `DatabaseClass`, and the corresponding rules describing the behavior associated with the new commands.

```
mod COHERENCE-COMPLETION-DATABASE-HANDLING is
```

```
pr DATABASE-HANDLING .
pr UNIT .

vars ME ME' : ModExp .
var DB : Database .
var MN : ModName .
var QIL : QidList .
vars T : Term .
var TL : TermList .

sort CohCompl .
subsort CohCompl < DatabaseClass .

op CohCompl : -> CohCompl .

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

The processing of the `coherence complete_.` and `coherence complete .` commands consists in executing the `coherence-complete` function (see Section 4.1) in a module including the `COHERENCE-COMPLETION` module.

```
op processCohCompl : ModExp Database -> QidList .
op processCohComplAux : Term -> QidList .

eq processCohCompl(ME, DB)
  = ('\n 'Coherence 'completion 'of my-meta-pretty-print(ME)
     processCohComplAux(
        meta-reduce(
           fmod 'FOO is
              including 'COHERENCE-COMPLETION .
              sorts none .
              none
              none
              none
              none
              none
           endfm,
           'coherence-complete[
               up(getFlatUnit(ME, evalModExp(ME, DB)))]))) .

eq processCohComplAux(error*)
  = ('\n 'An 'error 'has 'occurred 'during 'completion '.) .
eq processCohComplAux('fail[T])
  = ('\n 'The 'specification 'cannot 'be 'completed '.
     '\n 'The 'equation
     '\n '\t
     my-meta-pretty-print(
        fmod 'FOO is
```

```
            including 'META-LEVEL .
            sorts none .
            none
            none
            none
            none
            none
         endfm, T)
      '\n 'cannot 'be 'oriented 'with 'the 'order 'given) .
  eq processCohComplAux('fmod_is_______endfm[TL])
    = ('\n 'Completion 'solution ':
      my-meta-pretty-print(
         downModule('fmod_is_______endfm[TL]),
         downModule('fmod_is_______endfm[TL]))) .
  eq processCohComplAux('mod_is________endm[TL])
    = ('\n 'Completion 'solution ':
      my-meta-pretty-print(
         downModule('mod_is________endm[TL]),
         downModule('mod_is________endm[TL]))) .

  rl [CohCompl] :
     < O : X@CohCompl | db : DB,
        input : ({'coherence`complete`.}'PreCommand),
        output : QIL, default : MN, Atts >
     => < O : X@CohCompl | db : DB, input : nilTermList,
           output : processCohCompl(MN, DB),
           default : MN, Atts > .
  rl [CohCompl] :
     < O : X@CohCompl | db : DB, input : ('coherence`complete_.[T]),
        output : QIL, default : MN, Atts >
     => < O : X@CohCompl | db : DB, input : nilTermList,
           output : processCohCompl(parseModExp(T), DB),
           default : MN, Atts > .
endm
```

As in [12], we define a new constant COHERENCE-COMPLETION-GRAMMAR, of sort FModule. This constant is identified with the metarepresentation of the module to be used in the parsing process, which extends the module identified with EXT-GRAMMAR.

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

  op COHERENCE-COMPLETION-GRAMMAR : -> FModule .

  eq COHERENCE-COMPLETION-GRAMMAR
    = addImportList((including 'COHERENCE-COMPLETION-SIGN .),
         EXT-GRAMMAR) .
```

```
endfm
```

We finish our specification with the following module.

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

  subsort Object < State .

  op kbc : -> Oid .
  op init : -> System .

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

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

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

```
            QIL']
        fi
        if QIL =/= nil .

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

4.2 Some Examples of the Use of the Coherence Completion Tool

4.2.1 Rewriting Module Identity

The result of executing the `coherence complete_.` command on the `IDENTITY` module presented in Section 3.2.2 is as follows.

```
Maude> (coherence complete IDENTITY .)
rewrites: 1834 in 20ms cpu (30ms real) (91700 rewrites/second)

Coherence completion of IDENTITY
Completion solution : mod IDENTITY is
        including BOOL .
        sorts S .
        op _+_ : S S -> S [ comm ] .
        op 0 : -> S .
        op f : S -> S .
        var X : S .
        var Y : S .
        var Y@S : S .
        var X@S : S .
        var X@ : S .
        var X@@S : S .
        var Y@ : S .
        var Y@@S : S .
        eq X + 0 = X .
        rl [ r ] : f ( X@@S ) => f ( 0 ) + f ( X@@S ) .
        rl [ r1 ] : f ( X + Y ) => f ( X ) + f ( Y ) .
endm
```

4.2.2 Sequent Calculus for Propositional Calculus

The completion process does not terminate when executing the command `coherence complete_.` on the `SEQUENT-RULES-PROP-LOG` module presented in Section 3.2.3.

4.2.3 A Nondeterministic Choice Operator

Our last example is a very simple example taken from [15]. First, we consider the module `NAT` specifying the natural numbers in Peano notation with successor, predecessor, and addition functions.

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

This specification is terminating, as shown by the termination checker tool in [10], and Church-Rosser, as shown by the Church-Rosser checker in [12]. For the termination proof we have used the lexicographic path order induced by the module `NAT-ORDER` below.

```
(fmod NAT-ORDER is
   pr QID .
   op _>>_ : Qid Qid -> Bool .
   eq '_+_ >> '0 = true .
   eq '_+_ >> 's_ = true .
   eq '_+_ >> 'p_ = true .
   eq 's_ >> 'p_ = true .
   eq 's_ >> '0 = true .
   eq 'p_ >> '0 = true .
 endfm)

(view NAT-ORDER from STRICT-ORDER to NAT-ORDER is
   op _>>_ to _>>_ .
 endv)

Maude> (check termination NAT with LPO[NAT-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 16204 in 80ms cpu (90ms real) (202550 rewrites/second)

Termination check of NAT
The specification is terminating.
```

```
Maude> (check Church-Rosser .)
rewrites: 6806 in 30ms cpu (40ms real) (226866 rewrites/second)

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

Now, we consider the following system module NAT-CHOICE, which adds a nondeterministic choice operator to the natural numbers.

```
(mod NAT-CHOICE is
   inc NAT .
   op _?_ : Nat Nat -> Nat [comm] .
   vars N M : Nat .
   rl [ch] : N ? M => N .
 endm)
```

This specification is coherent, as shows the result given by the coherence checker.

```
Maude> (check coherence NAT-CHOICE .)
rewrites: 570 in 0ms cpu (10ms real) (~ rewrites/second)

Coherence checking of NAT-CHOICE
Checking solution :
        The specification is Coherent .
```

Obviously, the result of coherence completing such a specification is the same specification.

```
Maude> (coherence complete NAT-CHOICE .)
rewrites: 1018 in 120ms cpu (130ms real) (8483 rewrites/second)

Coherence completion of NAT-CHOICE
Completion solution : mod NAT-CHOICE is
        including BOOL .
        sorts NzNat Nat .
        subsort NzNat < Nat .
        op _+_ : Nat Nat -> Nat [ comm ] .
        op _?_ : Nat Nat -> Nat [ comm ] .
        op 0 : -> Nat .
        op s_ : Nat -> NzNat .
        op p_ : NzNat -> Nat .
        var M : Nat .
        var N : Nat .
        var N@Nat : Nat .
        var M@Nat : Nat .
        eq N@Nat + 0 = N@Nat .
        eq s N@Nat + s M@Nat = s s ( N@Nat + M@Nat ) .
        eq p s N@Nat = N@Nat .
        rl [ ch ] : N ? M => N .
endm
```

References

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

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

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

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. A Maude tutorial. Manuscript, 2000.

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

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

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

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

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

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

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

[12] Francisco Durán and José Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, 2000.

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

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

[15] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

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

[17] Patrick Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Department of Computer Science, University of Pisa, December 1995.