The Maude Formal Tool Environment

Francisco Durán

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
duran@lcc.uma.es

JAIST-FSSV, March 2010

Maude's formal environment

Sufficient completeness

Sort decreasingness

Termination

Confluence

Maude

Coherence

Execution

Theorem proving

Model checking

Reachability analysis

Maude's formal environment: part of Maude or around it

Sufficient completeness

Sort decreasingness

Termination

Confluence

Maude

Coherence

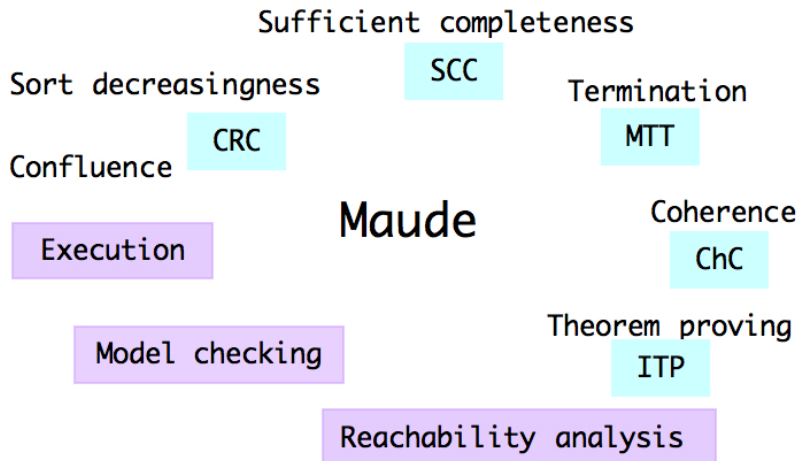
Execution

Theorem proving

Model checking

Reachability analysis

Maude's formal environment: tools around Maude



Based on rewriting logic

- Because of its logical basis and its initial model semantics, a Maude module defines a precise mathematical model.
- This means that Maude and its formal tool environment can be used in three, mutually reinforcing ways:
  - as a declarative programming language,
  - as an executable formal specification language, and
  - as a formal verification system.
- The Maude system, its documentation, and related papers and applications are available from the Maude website

  `http://maude.cs.uiuc.edu`.

The Maude formal environment

- Besides the built-in support for verifying invariants and LTL formulas, the following tools are also available as part of the Maude formal environment:
  - the Church-Rosser Checker (CRC) can be used to check the Church-Rosser property of functional modules;
  - the Maude Termination Tool (MTT) can be used to prove termination of system modules;
  - the Coherence Checker (ChC) can be used to check the coherence (or ground coherence) of system modules; and
  - the Inductive Theorem Prover (ITP) can be used to verify inductive properties of functional modules;
  - the Sufficient Completeness Checker (SCC) can be used to check that defined functions have been fully defined in terms of constructors.

Developers

- Many researches have contributed in one way or another to the tools. This list includes the main developers of each of the tools:
    - ITP: M. Clavel, J. Hendrix, and J. Meseguer
    - MTT: F. Durán, S. Lucas and J. Meseguer
    - CRC: F. Durán and J. Meseguer
    - ChC: F. Durán and J. Meseguer
    - SCC: J. Hendrix and J. Meseguer
- Maude team:
    - Manuel Clavel
    - Francisco Durán
    - Steven Eker
    - Pat Lincoln
    - Narciso Martí-Oliet
    - José Meseguer
    - Carolyn Talcott

Construction of the tools

- Full Maude, an extension of Maude, written in Maude itself, has played a key role in the construction of some of these tools.
- The ITP is a Maude program. It comprises over 8000 lines of Maude code that make extensive use of the reflective capabilities of Maude.
- The MTT tool implementation distinguishes two parts:
  - a family of theory transformations written in Maude, and
  - a Java application connects Maude to back-end termination tools and provides a graphical user interface.
- The CRC and ChC tools are written in Maude, and are in fact *executable specifications* of the formal inference systems that they implement. A complete execution environment for the tools has been integrated within Full Maude.
- The SCC is written in Maude, and internally constructs propositional tree automatons. Their emtiness check is performed by a C++ tree automata library named CETA.

Equational simplification

Let $T = (\Sigma, E)$ be an OS equational theory. We say that the equations in E are admissible as equational simplification rules if each equation $(\forall X)\ t = t' \Leftarrow u_1 = v_1 \wedge \ldots \wedge u_n = v_n$ in E satisfies the following two properties:

- *fewer variables on the right side and condition*, that is, the families of variables vars(t'), vars(u_i), and vars(v_i), $1 \leq i \leq n$, are all contained in vars$(t) = X$, where, vars(t) denotes the family of variables actually appearing in t.
- sort decreasingness, that is, for any substitution $\theta : X \longrightarrow T_\Sigma(Y)$, and any $s \in S$, if $\overline{\theta}(t) \in T_\Sigma(Y)_s$, then $\overline{\theta}(t') \in T_\Sigma(Y)_s$.

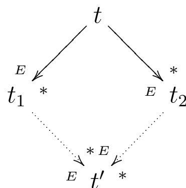
Confluence

Suppose we have a theory $T = (\Sigma, E)$ whose equations are admissible as equational simplification rules. Then we can do sound inference by equational simplification with E, but the equations E may still be quite unusable, because depending on the order and place of equation application we may get different results, or even no result at all.

In general, equational simplification can be nondeterministic.

The minimum requirement to make equational simplification deterministic is confluence.

We say that E is confluent iff whenever we have $t \xrightarrow{*}_E t_1$ and $t \xrightarrow{*}_E t_2$, then $t_1 \downarrow_E t_2$.



We call E ground confluent iff the above property is guaranteed only for terms without variables $t \in T_\Sigma$.

Termination

- Terminating equational programs are obtained when the rewriting relation is terminating.
- We say that the equations E are terminating as simplification rules when there is no infinite chain of rewrites

$$t \longrightarrow_E t_1 \longrightarrow_E t_2 \ldots t_{n-1} \longrightarrow_E t_n \longrightarrow \ldots$$

The Church-Rosser property

- The (ground) Church-Rosser property, together with termination, is essential for an equational specification to have good executability conditions, and also for having a complete agreement between the specification's initial algebra, mathematical semantics, and its operational semantics by rewriting.
- If specifications are *ground*-Church-Rosser and terminating, then 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.
- 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 ensuring that the normal form will have the least possible sort among those of all other equivalent terms.

CRC: a Church-Rosser checker for Maude specifications

- Under the assumption of termination, the Church-Roser Checker tries to check the confluence property.
- The tool
  - may succeed, in which case it responds with a confirmation that the module is confluent, or
  - it may fail to check it, in which case it responds with a set of proof obligations required to ensure that the specification is ground confluent.
- The present CRC tool accepts order-sorted conditional specifications, where each of the operation symbols has either no equational attributes, or any combination of associativity/commutativity/identity. 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 (operationally) terminating.

Confluence and ground confluence

- The specification can often be ground-confluent without being confluent for arbitrary terms with variables.
- If we submit to the tool a module fmod(Σ, E)endfm (satisfying the previous restrictions) and such that the equations E are unconditional, then if the equations E are indeed sort-decreasing and confluent, the checker tool will succeed.
- Failure to pass the check may mean one of three things:
    - (Σ, E) is ground confluent, but not confluent;
    - (Σ, E) is confluent, but further reasoning is needed, because some of the equations in E are conditional;
    - (Σ, E) fails to be ground confluent.

Confluence and ground confluence

- Blindly applying a completion procedure that is trying to establish the Church-Rosser property for arbitrary terms may be both
  - quite hopeless and
  - even unnecessary.
- Our tool attempts to establish the ground-Church-Rosser property modulo the equational axioms specified by checking a sufficient condition.
- The tool's output consists of a set of critical pairs and a set of membership assertions that must be shown, resp., ground-joinable, and ground-rewritable to a term with the required sort.
- User interaction is essential, completion is not attempted.
- Instead, proof obligations are generated and are given back to the user.

In summary

- What the Church-Rosser Checker does is:
  - it checks that the equations E are sort-decreasing;
  - it forms all the critical pairs for the equations E and tries to join them;
  - it returns as proof obligations those equation specializations that it could not prove sort-decreasing, and those simplified critical pairs that it could not join.
- In case the check fails, the proof obligations returned can be very useful for further analysis, either to establish the property, or to find a conterexample.

Example: a Church-Rosser one

```
Maude> (fmod CNAT is
          sort Natural .
          op 0 : -> Natural [ctor] .
          op s_ : Natural -> Natural [ctor] .
          op _+_ : Natural Natural -> Natural .
          vars N M : Natural .
          eq N + 0 = N .
          eq N + s M = s(N + M) .
        endfm)

Maude> (check Church-Rosser .)

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

Ground confluent but not confluent

- Quite often, a module will not pass the check, not because there is any real problem with its equations, but simply because it is ground confluent but not confluent.
- In such a case, the tool will return a set of critical pairs as proof obligations. Such critical pairs are equations $(\forall X)\ t = t'$ such that:
  - $E \vdash (\forall X)\ t = t'$,
  - the tool failed to establish $t \downarrow_E t'$.
- Furthermore, as we shall see, they are sufficient, as proof obligations, to establish ground confluence. That is, if we can show $\overline{\theta}(t) \downarrow_E \overline{\theta}(t')$ for each ground substitution θ, then E is indeed ground confluent (assuming termination).

A ground-confluent but not confluent specification

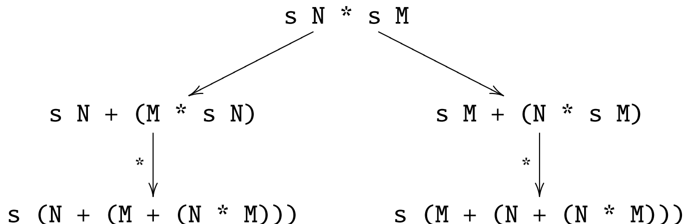
```
Maude> (fmod CNAT-2 is
          sorts Zero Natural .
          subsort Zero < Natural .
          op 0 : -> Zero .
          op s_ : Natural -> Natural .
          ops _+_ _*_ : Natural Natural -> Natural [comm] .
          vars N M : Natural .
          eq [nat01] : 0 + N = N .
          eq [nat02] : s N + M = s (N + M) .
          eq [nat03] : 0 * N = 0 .
          eq [nat04] : s N * M = M + (N * M) .
        endfm)
Introduced module CNAT-2

Maude> (check Church-Rosser .)

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

A ground-confluent but not confluent specification (II)

- This critical pair comes from applying the equation

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

modulo commutativity to the term `s N * s M` in two different ways yielding terms that, after further simplification, cannot be further simplified, and therefore cannot be joined, showing that the equations are not confluent.



- However, every ground instance can be joined.

A ground-confluent but not confluent specification (II)

- What to do?
  1. use the critical pair as useful information to transform the equations into equivalent equations that are confluent; or
  2. attempt an automatic transformation into confluent equations using Maude's Knuth-Bendix Completion tool; or
  3. prove an inductive theorem about the rewriting relation $\longrightarrow_E$ itself, not about equality!, showing that for each ground instance the pair can be joined; or
  4. find a conterexample disproving ground confluence.

A ground-confluent but not confluent specification (III)

In our example, alternative (1) yields a transformed module, by realizing that the equation `nat04` is in a sense too general.

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

Maude> (fmod CNAT-3 is
          sorts Zero Natural .
          subsort Zero < Natural .
          op 0 : -> Zero .
          op s_ : Natural -> Natural .
          ops _+_ _*_ : Natural Natural -> Natural [comm] .
          vars N M : Natural .
          eq [nat01] : 0 + N = N .
          eq [nat02] : s N + M = s (N + M) .
          eq [nat03] : 0 * N = 0 .
          eq [nat04']: s N * s M = s((N + M) + (N * M)) .
        endfm)
Maude> (check Church-Rosser CNAT-3 .)
Church-Rosser checking of CNAT-3
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

A ground-confluent but not confluent specification (IV)

Alternatively, we can join the critical pair by making _*_ associative.

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

```
Maude> (fmod CNAT-4 is
          sorts Nat Zero .
          subsorts Zero < Nat .
          op 0 : -> Zero .
          op s_ : Nat -> Nat .
          op _+_ : Nat Nat -> Nat [comm assoc] .
          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 CNAT-4 .)
```

```
Church-Rosser checking of CNAT-4
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

Descent

- Given a specification of natural numbers and integers with the typical operations and definitions, and in particular a `square` operation defined as

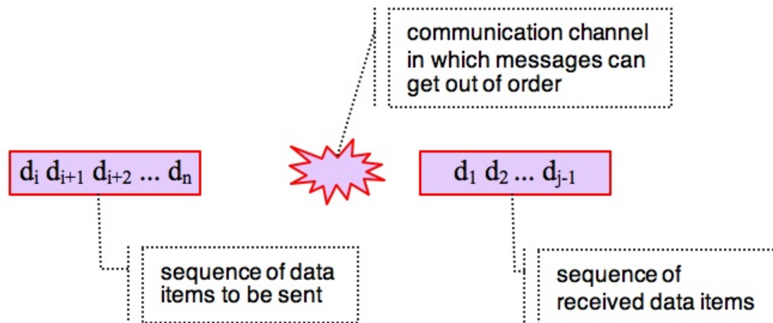
```
op square : Int -> Nat .
eq square(I:Int) = I:Int * I:Int .
```

  this equation gives rise to a membership assertion, because the least sort of the term `square(I:Int)` is `Nat`, but it is `Int` for the term in the righthand side.
- The proof obligation generated by the tool is

```
mb I:Int * I:Int : Nat .
```

- This membership assertion must be proved inductively. 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 using the ITP.

An unordered communication channel



In-order communication in an unordered channel

To achieve this in-order communication in spite of the unordered nature of the channel, the sender sends each data item in a message together with a sequence number; and the receiver sends back an ack message indicating that has received the item.

Communication channel's structure

```
fmod UNORDERED-CHANNEL-EQ is
  sorts Natural List Msg Conf State .
  subsort Msg < Conf .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op nil : -> List [ctor] .
  op _;_ : Natural List -> List [ctor] .    *** list constructor
  op _@_ : List List -> List .              *** list append
  op [_,_] : Natural Natural -> Msg [ctor] .
  op ack : Natural -> Msg [ctor] .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
  op {_,_|_|_,_} : List Natural Conf List Natural -> State [ctor] .
  vars N M J K : Natural . vars L P Q : List . var  C : Conf .
  eq nil @ L = L .
  eq (N ; L) @ P = N ; (L @ P) .
endfm
```

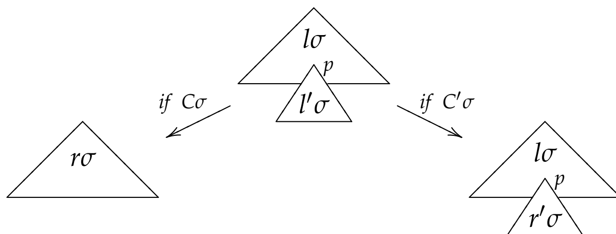
Communication channel's dynamics

```
mod UNORDERED-CHANNEL is
  pr UNORDERED-CHANNEL-EQ .

  vars N M J K : Natural .
  vars L P Q : List .
  var  C : Conf .

  rl [snd]: {N ; L, M | C | P, K} => {N ; L, M | [N, M] C | P, K} .

  rl [rec]: {L, M | [N, J] C | P, J}
    => {L, M | ack(J) C | P @ (N ; nil), s(J)}  .

  rl [rec-ack]: {N ; L, J | ack(J) C | P, M}  => {L, s(J) | C | P, M} .
endm
```

Church-Rosser property

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

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

Conditional critical pairs



Definition

Given an OS equational specification $\mathcal{R} = (\Sigma, R \cup A)$, with Σ A-preregular and R A-coherent, and given rewrite rules $l \to r$ `if` C and $l' \to r'$ `if` C' in R such that $vars(l, r, C) \cap vars(l', r', C') = \emptyset$ and $l|_p\sigma =_A l'\sigma$, for some nonvariable position $p \in \mathcal{P}(l)$ and A-unifier σ, then the triple

$$C\sigma \wedge C'\sigma \Rightarrow l\sigma[r'\sigma]_p = r\sigma$$

is called a (conditional) critical pair.

A sufficient condition for confluence

Definition

Let $\mathcal{R} = (\Sigma, A, R)$ be a deterministic rewrite theory that is quasi-reductive w.r.t. an A-compatible well-founded relation $\succ$, and let $C \Rightarrow s = t$ be a critical pair resulting from $l_i \to r_i$ *if* C_i for $i = 1, 2$, and $\sigma \in \mathit{Unif}_A(l_1|_p, l_2)$. We call $C \Rightarrow s = t$ unfeasible if there is some $u \to v$ in C such that $\overline{u} \to_{R\cup\overline{C},A} \overline{w}_1$, $\overline{u} \to_{R\cup\overline{C},A} \overline{w}_2$, and $\mathit{Unif}_A(w_1, w_2) = \emptyset$ and w_1 and w_2 are strongly irreducible with R modulo A. We call $C \Rightarrow s = t$ context-joinable if $\bar{s}\downarrow_{R\cup\overline{C}}\bar{t}$.

Theorem

Let $\mathcal{R} = (\Sigma, A, R)$ be a strongly deterministic rewrite theory that is quasi-reductive w.r.t. an A-compatible well-founded relation $\succ$. If every critical pair $C \Rightarrow s = t$ of $\mathcal{R}$ is either unfeasible or context-joinable, then $\mathcal{R}$ is confluent.

MTT: a Termination Tool for Maude Specifications

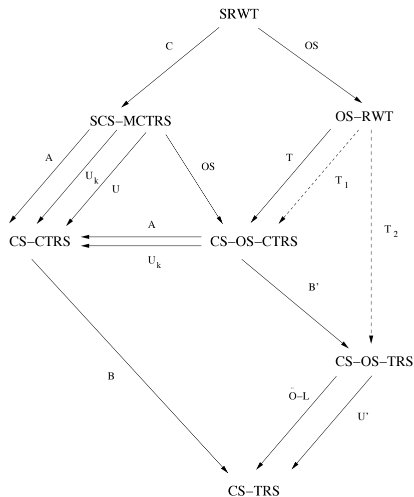
- A remarkable development in the theory of termination
  - AProVE,
  - TTT,
  - MU-TERM,
  - ...
  - . . .
- They consider restrictive specifications
  - untyped, unconditional, ...
- High-level (equational) languages
  - ASF-SDF, CafeOBJ, Maude, ...

  with advanced features
  - conditional equations and rules,
  - types and subtypes,
  - (possibly programmable) strategies for controlling execution,
  - matching modulo axioms,
  - ...

Termination tools for high-level languages

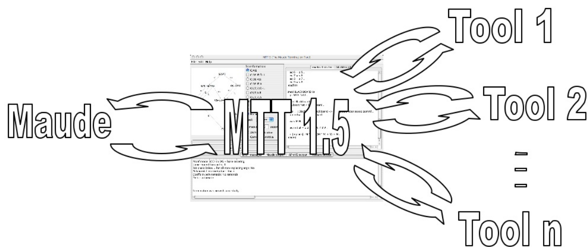
- based on rewriting and membership equational logics
  - with equations, rules, and membership axioms
  - with types, subtypes, and kinds
  - with matching modulo A, C, U, and its combinations
  - with local strategies
- Haskell

A transformational aproach



On the implementation of the tool

- transformations written in Maude
- Java GUI
- the termination competition
  - the TPDB syntax
  - interaction with the tools
- interaction with the tools
  - local
  - remote via web services



UNORDERED-CHANNEL-EQ terminating

MTT 5 (The Maude Termination Tool)

File Edit Help

SRWT
C
OS
SCS_MCTRS
OS_RWT
A
Uk
U
OS
T
CS_CTRS
Uk
U
CS_OS_CTRS
B'
B
CS_OS_TRS
O-L
Uk
U
CS_TRS

Tool: AProVE

Timeout: 30 seconds

AND optimization

Context-sensitive

Check

Automatic check

unordered-channel-eq-no-ids.fm

```
op 0 : -> Natural
  [ctor] .
op _;_ : Natural List -> List
  [ctor] .
op __ : Conf Conf -> Conf
  [assoc comm ctor] .
op `[_`,_`] : Natural Natural -> Msg
  [ctor] .
op `{_`,_|_|_`,_`} : List Natural Conf List Natural -> State
  [ctor] .
op ack : Natural -> Msg
  [ctor] .
op nil : -> List
  [ctor] .
op null : -> Conf
  [ctor] .
op s : Natural -> Natural
  [ctor] .
eq nil @ L:List
  = L:List .
eq (N:Natural ; L:List)@ P:List
  = N:Natural ;(L:List @ P:List) .
eq null X:[Conf]
  = X:[Conf] [label idEq1] .
endfm)
```

AProVE

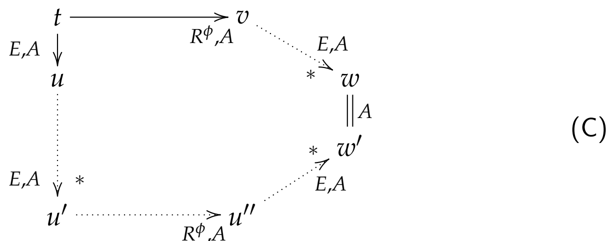
C;U;B false false

Termination of R successfully shown.

Duration:
0:00 minutes

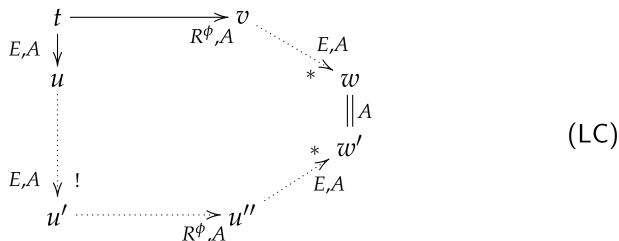
Coherence

- Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$, and assuming E confluent (resp. ground confluent), sort-decreasing and terminating modulo A, we say that the rules R are *coherent* (resp. ground coherent) with E modulo A relative to ϕ if for each Σ-term t (resp. ground Σ-term t) such that $t \rightarrow_{E,A} u$, and $t \rightarrow_{R^\phi,A} v$



Local coherence

$\mathcal{R}$ is called *locally coherent* (resp. *ground locally coherent*) iff for each Σ-term t (resp. ground Σ-term t) such that $t \rightarrow_{E,A} u$, and $t \rightarrow_{R^\phi,A} v$



(LC)

Theorem

$\mathcal{R}$ is coherent (resp. ground coherent) iff $\mathcal{R}$ is locally coherent (resp. locally ground coherent).

Context of use

- The user has already developed an *executable specification* with an initial model semantics which has already been
  - checked to have confluent and terminating equations and
  - 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.
- The tool can only guarantee success when the user's specification is unconditional and coherent, and not just ground-coherent: not generating any proof obligations is only a *sufficient* condition.

Use of the tool

- User interaction will typically be quite essential, coherence completion is not attempted.
- The specification may be *ground* coherent, but not coherent, so that a collection of critical pairs will be returned by the tool as proof obligations.
- The feedback of the tool should instead be used as a guide for careful analysis about one's specification.
    - The Maude ITP can be enlisted to prove some of these proof obligations.
    - The user may in fact have to modify the original specification by carefully considering the information conveyed by the proof obligations.

```
fmod UNORDERED-CHANNEL-EQ is
  sorts Natural List Msg Conf State .
  subsort Msg < Conf .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op nil : -> List [ctor] .
  op _;_ : Natural List -> List [ctor] .    *** list constructor
  op _@_ : List List -> List .              *** list append
  op [_,_] : Natural Natural -> Msg [ctor] .
  op ack : Natural -> Msg [ctor] .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
  op {_,_|_|_,_} : List Natural Conf List Natural -> State [ctor] .
  vars N M J K : Natural . vars L P Q : List . var  C : Conf .
  eq nil @ L = L .
  eq (N ; L) @ P = N ; (L @ P) .
endfm

mod UNORDERED-CHANNEL is
  pr UNORDERED-CHANNEL-EQ .
  vars N M J K : Natural . vars L P Q : List . var  C : Conf .
  rl [snd]: {N ; L, M | C | P, K} => {N ; L, M | [N, M] C | P, K} .
  rl [rec]: {L, M | [N, J] C | P, J}
    => {L, M | ack(J) C | P @ (N ; nil), s(J)}  .
  rl [rec-ack]: {N ; L, J | ack(J) C | P, M}  => {L, s(J) | C | P, M} .
endm
```

Ground coherence of UNORDERED-CHANNEL

```
Maude> (check ground coherence UNORDERED-CHANNEL .)

Coherence checking of UNORDERED-CHANNEL
Coherence checking solution:
All critical pairs have been rewritten and all equations are non-constructor.
The specification is ground coherent.
```

UNORDERED-CHANNEL-ABSTRACTION

```
mod UNORDERED-CHANNEL-ABSTRACTION-EQ is
  including UNORDERED-CHANNEL-EQ .
  vars M N P K : Natural .
  vars L L' L'' : List .
  var C : Conf .
  eq [A1]: {L, M | [N, P] [N, P] C | L', K} = {L, M | [N, P] C | L', K} .
endm

mod UNORDERED-CHANNEL-ABSTRACTION is
  including UNORDERED-CHANNEL-ABSTRACTION-EQ .
  including UNORDERED-CHANNEL .
endm
```

Coherence of UNORDERED-CHANNEL-ABSTRACTION

```
Maude> (check ground coherence UNORDERED-CHANNEL-ABSTRACTION .)

Coherence checking of UNORDERED-CHANNEL-ABSTRACTION
Coherence checking solution:
The following critical pairs cannot be rewritten:
  cp for A1 and rec
    {L:List, M:Natural | #3:Conf [N:Natural, J:Natural] | P:List, J:Natural}
    => {L:List, M:Natural |
          #3:Conf ack(J:Natural) [N:Natural, J:Natural] |
          P:List @ N:Natural ; nil, s(J:Natural)}.
  cp for A1 and rec
    {L:List, M:Natural | [N:Natural, J:Natural] | P:List, J:Natural}
    => {L:List, M:Natural |
          ack(J:Natural) [N:Natural, J:Natural] |
          P:List @ N:Natural ; nil, s(J:Natural)}.
```

UNORDERED-CHANNEL-ABSTRACTION-2

These critical pairs indicate that a rule is missing.

```
mod UNORDERED-CHANNEL-ABSTRACTION-2-EQ is
  extending UNORDERED-CHANNEL-ABSTRACTION-EQ .
endm

mod UNORDERED-CHANNEL-ABSTRACTION-2 is
  extending UNORDERED-CHANNEL-ABSTRACTION-2-EQ .
  extending UNORDERED-CHANNEL-ABSTRACTION .
  vars M N P K : Natural .
  vars L L' L'' : List .
  var  C : Conf .
  rl [rec2]: {L, M | [N, K] C | L', K}
    => {L, M | [N, K] ack(K) C | L' @ N ; nil, s(K)} .
endm
```

Coherence of UNORDERED-CHANNEL-ABSTRACTION-2

```
Maude> (check ground coherence UNORDERED-CHANNEL-ABSTRACTION-2 .)

Coherence checking of UNORDERED-CHANNEL-ABSTRACTION-2
Coherence checking solution:
All critical pairs have been rewritten, and no rule can be applied
below non-frozen and non-linear variables of equations.
The specification is ground coherent.
```

Church-Rosser property of UNORDERED-CHANNEL-ABSTRACTION-2

```
Maude> (check Church-Rosser UNORDERED-CHANNEL-ABSTRACTION-2 .)

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

UNORDERED-CHANNEL-ABSTRACTION-2-EQ terminating

MTT 5 (The Maude Termination Tool)

File Edit Help

SRWT, SCS_MCTRS, OS_RWT, CS_CTRS, CS_OS_CTRS, CS_OS_TRS, CS_TRS

Tool: AProVE

Timeout: 30 seconds

AND optimization

Context-sensitive

Check

Automatic check

unordered-channel-abstraction-2-eq-no-ids.fm

```
(set include BOOL off .)
(set include TRUTH-VALUE off .)

(mod UNORDERED-CHANNEL-ABSTRACTION-2-EQ is
  sorts Conf List Msg Natural State .
  subsort Msg < Conf .
  op _@_ : List List -> List .
  op 0 : -> Natural
    [ctor] .
  op _;_ : Natural List -> List
    [ctor] .
  op __ : Conf Conf -> Conf
    [assoc comm ctor] .
  op `[_`,_`] : Natural Natural -> Msg
    [ctor] .
  op `{_`,_|_|_`,_`} : List Natural Conf List Natural -> State
    [ctor] .
  op ack : Natural -> Msg
    [ctor] .
  op nil : -> List
    [ctor] .
  op null : -> Conf
    [ctor] .
  op s : Natural -> Natural
    [ctor] .
```

AProVE

C;U;B false false

Termination of R successfully shown.

Duration:
0:00 minutes

Conditional critical pairs modulo A

Given rewrite rules with disjoint variables $l \to r \ if \ C$ in R and $l' \to r' \ if \ C'$ in E, their set of conditional critical pairs modulo A is defined as usual: Either we find a non-variable position p in l such that $\alpha \in Unif_A(l|_p, l')$

$$\alpha(C) \wedge \alpha(C') \quad \Rightarrow \quad \begin{array}{ccccc} \alpha(l[l']_p) & \underset{A}{=\!=\!=} & \alpha(l) & \underset{R}{\longrightarrow} & \alpha(r) \\ {\scriptstyle E}\big\downarrow & & & & \\ \alpha(l[r']_p) & & & & \end{array} \qquad \text{(I)}$$

or a non-variable and *nonfrozen* position p' in l' with $\alpha \in Unif_A(l'|_{p'}, l)$

$$\alpha(C) \wedge \alpha(C') \quad \Rightarrow \quad \begin{array}{ccccc} \alpha(l') & \underset{A}{=\!=\!=} & \alpha(l'[l]_{p'}) & \underset{R}{\longrightarrow} & \alpha(l'[r]_{p'}) \\ {\scriptstyle E}\big\downarrow & & & & \\ \alpha(r') & & & & \end{array} \qquad \text{(II)}$$

Sufficient condition for coherence

Theorem

Given $\mathcal{R}$ as above, then if:

(i) *all conditional critical pairs are joinable and*

(ii) *for any equation $l' \to r'$ if C' in E, for each $x \in$ vars(l') such that x is non-frozen in l', then either*

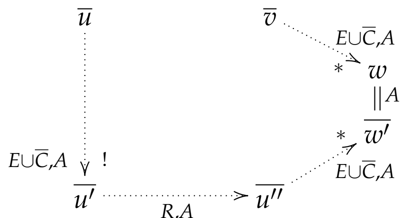
  (a) *x is such that $x \notin$ vars(C'), x is also non-frozen in r', and x is linear in both l' and r', or*

  (b) *the sort s of x is such that no rewriting with $\to_{R,A}$ is possible for terms of such sort s,*

then $\mathcal{R}$ is coherent.

Context joinability of conditional critical pairs

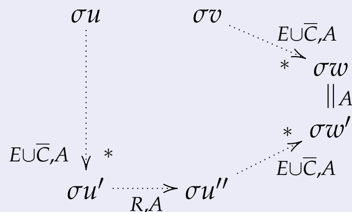
Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$, a non-joinable conditional critical pair $C \Rightarrow u \to v$ is *context-joinable* if and only if in the extended rewrite theory $\mathcal{R}_C = (\Sigma \cup \overline{X}, E \cup \overline{C} \cup A, R)$ we have:



Context joinability of conditional critical pairs

Lemma

If the conditional critical pair $C \Rightarrow u \to v$ is context joinable, then for all substitutions σ such that σC holds we have



and therefore, the coherence property holds for the conditional critical pair $C \Rightarrow {}_{E,A}u \swarrow t \searrow v_{R,A}$.

The ITP: an Inductive Theorem Prover

- The Maude Inductive Theorem Prover tool (ITP) is a theorem-proving assistant.
- It can be used to interactively verify inductive properties of membership equational specifications, or, more precisely, for proving properties of the initial algebra $T_{\mathcal{E}}$ of a MEL specification $\mathcal{E}$ written in Maude.
- It supports proofs by
  - structural induction, and
  - coverset induction.
- It is a program written in Maude by M. Clavel and J. Hendrix in which one can:
  - load in Maude the functional module or modules one wants to reason about,
  - enter named goals to be proved by the ITP, and
  - give commands, corresponding to proof steps, to prove that property.

Mathematical Proof of Associativity of Addition

- We want to prove that the addition operation in the module

  ```
  fmod NATURAL is
    sort Natural .
    op 0 : -> Natural [ctor] .
    op s : Natural -> Natural [ctor] .
    op _+_ : Natural Natural -> Natural .
    vars N M : Natural .
    eq N + 0 = N .
    eq N + s(M) = s(N + M) .
  endfm
  ```

  satisfies the associativity property,

  $$(\forall \mathtt{N,M,L})\ \mathtt{N + (M + L)} = \mathtt{(N + M) + L}.$$

- We can prove the property by induction on L. That is, we prove it for $\mathtt{L} = \mathtt{0}$ (base case) and then assuming that it holds for L, we prove it for s(L) (induction step).

- **Base Case:** We need to show,

$$(\forall \texttt{N,M})\ \texttt{N + (M + 0)} = \texttt{(N + M) + 0}.$$

We can do this trivially, by simplification with the equation

```
eq N + 0 = N .
```

- **Induction Step:** We think of `L` as a generic constant and assume that the associativity equation (induction hypothesis (IH))

$$(\forall \mathtt{N,M})\ \mathtt{N + (M + L)} = \mathtt{(N + M) + L}$$

holds for that constant. Then we try to prove the equation,

$$(\forall \mathtt{N,M})\ \mathtt{N + (M + s(L))} = \mathtt{(N + M) + s(L)}$$

using the induction hypothesis. Again, we can do this by simplification with the equations E in `NAT`, and the induction hypothesis IH equation, since we have,

$$\begin{array}{rcl} \mathtt{N + (M + s(L))} & \longrightarrow_E & \mathtt{N + s(M + L)} \\ & \longrightarrow_E & \mathtt{s(N + (M + L))} \\ & \longrightarrow_{IH} & \mathtt{s((N + M) + L)} \end{array}$$

and

$$\mathtt{(N + M) + s(L)} \longrightarrow_E \mathtt{s((N + M) + L)}.$$

q.e.d

Machine-assisted proof of associativity of addition

We first load into Maude the module, say,

```
fmod NATURAL is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

We then enter our associativity goal

```
Maude> (goal assoc : NATURAL |- A{N:Natural ; M:Natural ; L:Natural}
             ((N + (M + L)) = ((N + M) + L)) .)

=================================
label-sel: assoc@0
=================================
A{N:Natural ; M:Natural ; L:Natural}
N:Natural + (M:Natural + L:Natural) = (N:Natural + M:Natural) + L:Natural
+++++++++++++++++++++++++++++++++
```

We can then try to prove goal assoc@0 by induction on L:Natural:

```
Maude> (ind on L:Natural .)

=================================
label-sel: assoc@1.0
=================================
A{N:Natural ; M:Natural}
N:Natural + (M:Natural + 0) = (N:Natural + M:Natural) + 0

=================================
label: assoc@2.0
=================================
A{V0#0:Natural}
(A{N:Natural ; M:Natural}
N:Natural + (M:Natural + V0#0:Natural)
   = (N:Natural + M:Natural) + V0#0:Natural)
 ==>
(A{N:Natural ; M:Natural}
N:Natural + (M:Natural + s(V0#0:Natural))
   = (N:Natural + M:Natural) + s(V0#0:Natural))
+++++++++++++++++++++++++++++++++
```

We can then try prove the above "base case" subgoal by using the ITP's auto tactic that—after turning the variables into constants by the constants lemma (more on this later) and doing implication elimination if necessary—tries to simplify the goal by applying equations in the module, until hopefully reaching an identity.

```
Maude> (auto .)

=================================
label-sel: assoc@2.0
=================================
A{V0#0:Natural}
(A{N:Natural ; M:Natural}
N:Natural + (M:Natural + V0#0:Natural)
   = (N:Natural + M:Natural) + V0#0:Natural)
 ==>
(A{N:Natural ; M:Natural}
N:Natural + (M:Natural + s(V0#0:Natural))
   = (N:Natural + M:Natural) + s(V0#0:Natural))
+++++++++++++++++++++++++++++++++
```

We can likewise apply the `auto` tactic to the second goal, thus proving the associativity theorem.

```
Maude> (auto .)

q.e.d

+++++++++++++++++++++++++++++++++
```

Note that, in this case, both the constants lemma and implication elimination had to be invoked by `auto` before being able to simplify both sides of the conclusion using the induction hypothesis.

Using Lemmas

- Often, attempts at simplification using the `auto` tactic do not succeed.
- However, they suggest lemmas to be proved.
- Consider the following goal of proving commutativity of addition in our `NATURAL` module:

```
Maude> (goal comm : NATURAL |- A{N:Natural ; M:Natural}
        ((N + M) = (M + N)) .)

=================================
label-sel: comm@0
=================================
A N:Natural ; M:Natural N:Natural + M:Natural = M:Natural + N:Natural
+++++++++++++++++++++++++++++++++
```

We can try to prove it by induction on `M:Nat`

```
Maude> (ind on M:Natural .)

=================================
label-sel: comm@1.0
=================================
A {N:Natural} N:Natural + 0 = 0 + N:Natural

=================================
label: comm@2.0
=================================
A {V0#0:Natural}
(A {N:Natural}
 N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
 ==>
(A {N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural) + N:Natural)
+++++++++++++++++++++++++++++++++
```

When we apply the auto tactic to this first goal we get,

```
Maude> (auto .)

=================================
label-sel: comm@1.0
=================================
N*Natural = 0 + N*Natural
+++++++++++++++++++++++++++++++++
```

What we can do is to assume the unsimplified equation yielded by `auto` as a lemma in the proof of our main goal.

```
Maude> (lem 0-comm : A N:Natural ((0 + N) = (N)) .)

=================================
label-sel: 0-comm@0
=================================
A {N:Natural} 0 + N:Natural = N:Natural

=================================
label: comm@1.0
=================================
N*Natural = 0 + N*Natural

=================================
label: comm@2.0
=================================
A {V0#0:Natural}
(A {N:Natural} N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A {N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural) + N:Natural)
+++++++++++++++++++++++++++++++++
```

```
Maude> (ind on N:Natural .)

=================================
label-sel: 0-comm@1.0
=================================
0 + 0 = 0

=================================
label: 0-comm@2.0
=================================
A {V1#0:Natural}
0 + V1#0:Natural = V1#0:Natural ==> 0 + s(V1#0:Natural) = s(V1#0:Natural)

=================================
label: comm@1.0
=================================
N*Natural = 0 + N*Natural

=================================
label: comm@2.0
=================================
A{V0#0:Natural}
(A{N:Natural} N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A{N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural)+ N:Natural)
+++++++++++++++++++++++++++++++++
```

```
Maude> (auto .)

=================================
label-sel: 0-comm@2.0
=================================
A{V1#0:Natural}
0 + V1#0:Natural = V1#0:Natural
==>
0 + s(V1#0:Natural) = s(V1#0:Natural)
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
label-sel: comm@1.0
=================================
N*Natural = 0 + N*Natural
+++++++++++++++++++++++++++++++++
```

Proving now our first original subgoal becomes automatic (because of the lemma) but we are then faced with the second original subgoal:

```
Maude> (auto .)

=================================
label-sel: comm@2.0
=================================
A {V0#0:Natural}
(A {N:Natural} N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A {N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural)+ N:Natural)
+++++++++++++++++++++++++++++++++
```

We can apply also the auto tactic to the remaining goal comm@2.0, but, again, we get an unproved equality that we can use as a suggestion for a new lemma.

```
Maude> (auto .)

=================================
label-sel: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural)+ N*Natural
+++++++++++++++++++++++++++++++++

Maude> (lem s-comm : AN:Natural ; M:Natural ((s(M) + N) = (s(M + N))) .)

=================================
label: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural)+ N*Natural

=================================
label-sel: s-comm@0
=================================
A{N:Natural ; M:Natural} s(M:Natural) + N:Natural = s(M:Natural + N:Natural)
+++++++++++++++++++++++++++++++++
```

We can again enter and prove this lemma by induction on `N:Natural` and two applications of the `auto` tactic, which brings us back to our last unproved subgoal, which we can discharge with a last `auto` command.

```
Maude> (ind on N:Natural .)

=================================
label: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural) + N*Natural

=================================
label-sel: s-comm@1.0
=================================
A{M:Natural} s(M:Natural) + 0 = s(M:Natural + 0)

=================================
label: s-comm@2.0
=================================
A {V1#0:Natural}
(A {M:Natural} s(M:Natural) + V1#0:Natural = s(M:Natural + V1#0:Natural))
==>
(A {M:Natural} s(M:Natural) + s(V1#0:Natural) = s(M:Natural + s(V1#0:Natural)))
+++++++++++++++++++++++++++++++++
```

```
Maude> (auto .)

=================================
label-sel: s-comm@2.0
=================================
A{V1#0:Natural}
(A{M:Natural} s(M:Natural)+ V1#0:Natural = s(M:Natural + V1#0:Natural))
==>
(A{M:Natural} s(M:Natural)+ s(V1#0:Natural) = s(M:Natural + s(V1#0:Natural)))
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
label-sel: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural) + N*Natural
+++++++++++++++++++++++++++++++++

Maude> (auto .)

q.e.d

+++++++++++++++++++++++++++++++++
```

The ITP inference rules

- In the ITP we reason backwards, replacing the main goal G we want to prove by subgoals, $G_1, \ldots, G_n$, such that if we prove each of the subgoals, then we have proved the main goal.

$$\frac{G}{G_1 \quad \ldots \quad G_n}$$

The cns inference rule

- The lemma of constants converts universally quantified variables in a goal into constants.
- The fact that this is a semantically valid inference is based on the Constants Lemma, which states the equivalence between satisfiability of a quantified equation, and of the same equation with the variables transformed into generic constants,

$$E \models_{\Sigma} (\forall X)\ t = t' \quad \Leftrightarrow \quad E \models_{\Sigma(X)} (\forall \emptyset)\ t = t'.$$

- Thanks to the completeness of equational reasoning, this is expressed in the ITP as the cns rule,

$$\frac{E \vdash_{\Sigma} (\forall X)\ t = t'}{E \vdash_{\Sigma(X)} (\forall \emptyset)\ t = t'}$$

Reasoning by cases: the `split` rule

- Given an unquantified goal without variables and given a Boolean-valued expression involving some of those generic constants, the `split` rule splits a given goal into two:
  - one assuming the expression `true`, and
  - another assuming it `false`.
- Applications of `split` should protect `BOOL`.

Structural induction: the `ind` rule

Given a specification with a subsignature Ω of constructors, proving an inductive property of the form $(\forall x : s)\ P(x)$ consists in proving:

- **Base Case**. For any constant $a : \longrightarrow s'$ in Ω with $s' \leq s$, the subgoal $P(x/a)$.
- **Induction Step**. For each constructor $f : s_1 \ldots s_n \longrightarrow s'$ in Ω with $s' \leq s$, where the sorts $s_{i_1}, \ldots, s_{i_k}$ are those among the $s_1 \ldots s_n$ such that $s_{i_j} \leq s$, $1 \leq j \leq k$, the subgoal,

$$(\forall x_1 : s_1, \ldots, x_n : s_n)\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f(x_1, \ldots, x_n)).$$

This becomes an inductive inference rule of the form,

$$\frac{(\forall x : s)\ P(x)}{\bigwedge_i P(x/a_i) \wedge \bigwedge_j (\forall \overline{x} : \overline{s})\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f_j(x_1, \ldots, x_{n_j}))}$$

where the a_i and the f_j include all the constructor constants and operators meeting the properties specified above, and where $(\forall \overline{x} : \overline{s})$ abbreviates $(\forall x_1 : s_1, \ldots, x_{n_j} : s_{n_j})$.

Need to check sufficient completeness

- The declared subsignature of constructors must be correct.
- We need to check that it is sufficiently complete, ... for example using the SCC tool.

SCC: a sufficient completeness checker for Maude specs

- We need methods to check that an equational theory (Σ, E) is sufficiently complete.
- For arbitrary equational theories sufficient completeness is in general undecidable.
- We may have to do some inductive theorem proving.
- Sufficient completeness is decidable for a very broad class of order-sorted theories, namely, unconditional theories of the form $(\Sigma, E \cup A)$ with A a set of axioms for operators allowing any combination of associativity/commutativity/identity, except associativity alone, or associativity and identity alone, and E:
  1. left-linear;
  2. ground confluent and sort-decreasing; and
  3. weakly terminating.
- If (1)-(3) are satisfied, sufficient completeness becomes decidable in practice if A includes operators that are only A, or AU, for many specifications of interest using specialized heuristic algorithms.

The Maude SCC tool

- The Maude Sufficient Completeness Checker (SCC) is a tool developed by Joseph Hendrix at UIUC.
- It uses a library of tree automata modulo A operations also developed by him, called CETA, and reduces the sufficient completeness problem of specification $(\Sigma, E \cup A)$ satisfying conditions (1)–(3) to the emptiness problem for the tree automaton $\mathcal{A}_{D_s - (Red \cup C_s)}$ for each sort s in Σ.
- It outputs either "success" or a set of counterexample terms.

An example of use

```
Maude> fmod NATURAL is
         sort Nat .
         op 0 : -> Nat [ctor] .
         op s : Nat -> Nat [ctor] .
         op _+_ : Nat Nat -> Nat .
         vars X Y : Nat .
         eq X + 0 = X .
         eq X + s(Y) = s(X + Y) .
       endfm

Maude> select SCC-LOOP .
Maude> loop init-scc .

Maude> (scc NATURAL .)

Checking sufficient completeness of NATURAL ...
Success: NATURAL is sufficiently complete under the
    assumption that it is weakly-normalizing, confluent,
    and sort-decreasing.
```

Counterexample in case of failure

```
Maude> fmod MY-LIST is
         protecting NAT .
         sorts NzList List .
         subsorts Nat < NzList < List .
         op _;_ : List List -> List [assoc] .
         op _;_ : NzList NzList -> NzList [assoc ctor] .
         op nil : -> List [ctor] .
         op rev : List -> List .
         eq rev(nil) = nil .
         eq rev(N:Nat) = N:Nat .
         eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
       endfm

Maude> select SCC-LOOP .
Maude> loop init-scc .

Maude> (scc MY-LIST .)

Checking sufficient completeness of MY-LIST ...
Failure: The term 0 ; nil is a counterexample as it is a
    irreducible term with sort List in MY-LIST that does
    not have sort List in the constructor subsignature.
```

A revised version of MY-LIST

```
Maude> fmod MY-LIST2 is
         protecting NAT .
         sorts NzList List .
         subsorts Nat < NzList < List .
         op _;_ : List List -> List [assoc] .
         op _;_ : NzList NzList -> NzList [assoc ctor] .
         op nil : -> List [ctor] .
         op rev : List -> List .
         eq rev(nil) = nil .
         eq rev(N:Nat) = N:Nat .
         eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
         eq nil ; L:List = L:List .
         eq L:List ; nil = L:List .
       endfm

Maude> select SCC-LOOP .
Maude> loop init-scc .

Maude> (scc MY-LIST2 .)

Checking sufficient completeness of MY-LIST2 ...
Success: MY-LIST2 is sufficiently complete under the assumption
    that it is weakly-normalizing, confluent, and sort-decreasing.
```