Equational abstractions in rewriting logic and Maude

Narciso Martí-Oliet

(joint work with J. Meseguer, M. Palomino, F. Durán, and A. Verdejo)

Facultad de Informática

Universidad Complutense de Madrid, Spain

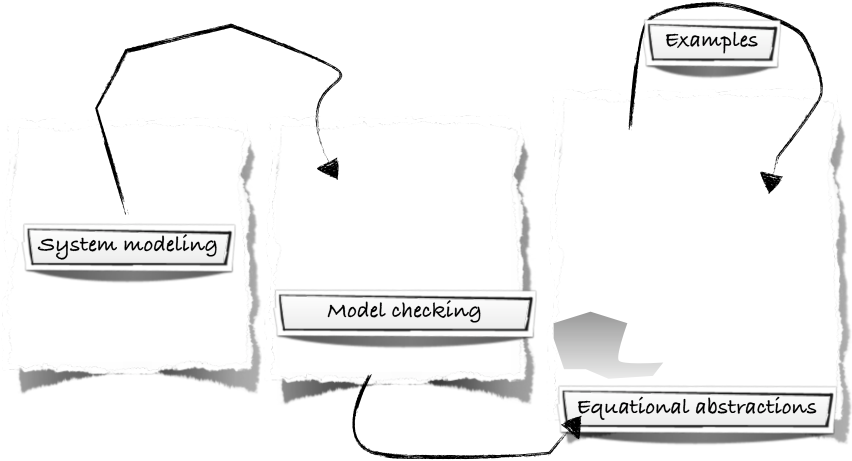
narciso@ucm.es

SBMF 2014, Maceió, Brazil, 30 Sept 2014

Goals

1. To introduce Maude as a framework for modeling systems and model checking their properties.
2. To present a simple method of defining quotient abstractions by means of equations collapsing the set of states.
3. To show how the Maude Formal Environment tools can help in discharging the associated proof obligations.

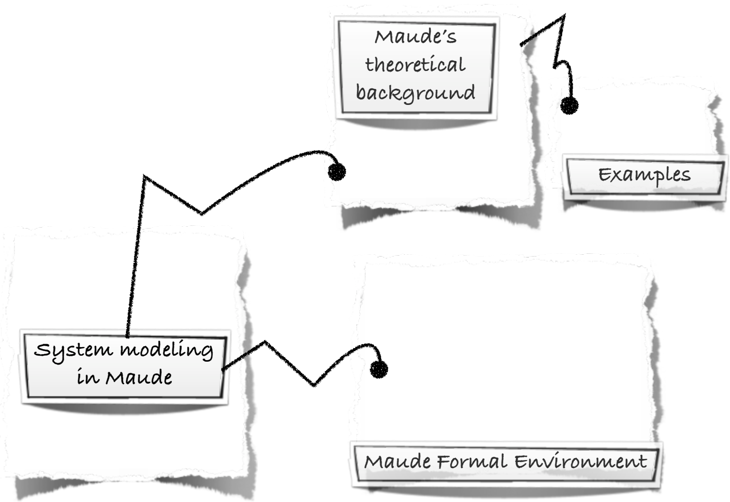
Outline


System modeling
Model checking
Equational abstractions
Examples

Abstraction... what for?

- Abstraction reduces the problem of whether an infinite state system satisfies a temporal logic property to model checking that property on a finite state abstract version.
- Some common abstractions are quotients of the original system.
- We present a simple method of defining quotient abstractions by means of equations collapsing the set of states.
- Our method yields the minimal quotient system together with a set of proof obligations that guarantee its executability and can be discharged with tools such as those in the Maude Formal Environment.

Outline


Maude's theoretical background
Examples
System modeling in Maude
Maude Formal Environment

Ingredients of rewriting logic

- Types (and subtypes).
- Typed operators providing syntax: signature Σ.
- Syntax allows the construction of both static data and states: term algebra T_{Σ}.

- Equations E define functions over static data as well as properties of states.
- Rewrite rules R define transitions between states.
- Deduction in the logic corresponds to computation with those functions and transitions.

- The Maude language is an implementation of (equational and) rewriting logic, allowing the execution of specifications satisfying some admissibility requirements.

So ... who is Maude?

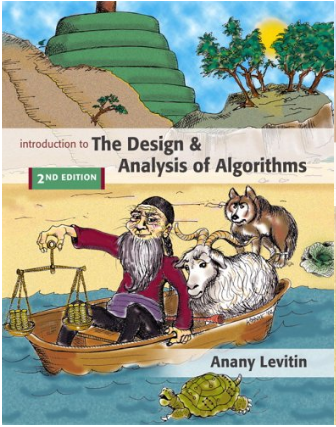
- Maude follows a long tradition of declarative algebraic specification languages in the OBJ family, including
  - OBJ3,
  - CafeOBJ,
  - Elan.
- Computation = Deduction in the appropriate logic.
- Functional modules = (Admissible) specifications in (membership) equational logic.
- System modules = (Admissible) specifications in rewriting logic.
- Operational semantics is based on matching and rewriting.

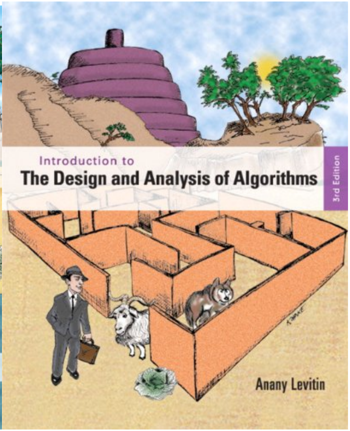
http://maude.cs.uiuc.edu

Example: crossing the river

- A shepherd needs to transport to the other side of a river
  - a wild dog,
  - a lamb, and
  - a cabbage.
- He has only a boat with room for the shepherd himself and another item.
- The problem is that in the absence of the shepherd
  - the wild dog would eat the lamb, and
  - the lamb would eat the cabbage.

Example: crossing the river





Example: crossing the river

- The shepherd and his belongings are represented as objects with only an attribute indicating the side of the river in which each is located.
- The group is put together by means of an associative and commutative juxtaposition.
- Constants `left` and `right` represent the two sides of the river.
- Operation `ch`(ange) is used to modify the corresponding attributes.
- Rules represent the ways of crossing the river that are allowed by the capacity of the boat.

Example: crossing the river

```
mod RIVER-CROSSING is
  sorts Side Group .

  ops left right : -> Side [ctor] .
  op ch : Side -> Side .
  eq ch(left) = right .
  eq ch(right) = left .

  ops s w l c : Side -> Group [ctor] .
  op __ : Group Group -> Group [ctor assoc comm] .

  var S : Side .

  rl [shepherd] : s(S) => s(ch(S)) .
  rl [wdog] : s(S) w(S) => s(ch(S)) w(ch(S)) .
  rl [lamb] : s(S) l(S) => s(ch(S)) l(ch(S)) .
  rl [cabbage] : s(S) c(S) => s(ch(S)) c(ch(S)) .
endm
```

Rewriting and equational simplification

Equational simplification: $t \rightarrow_E t'$

A term t rewrites to a term t' by an equation $l = r$ in E if:

1. there is a subterm $t|_p$ of t at a given position p of t s. t. l matches $t|_p$ via a substitution σ, i.e., $\sigma(l) \equiv t|_p$
2. t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$.

$$t = C[t|_p] = C[\sigma(l)] \rightarrow_E C[\sigma(r)] = t'$$

- We write $t \rightarrow_E^* t'$ to mean either $t = t'$ (0 steps) or $t \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \cdots \rightarrow_E t_n \rightarrow_E t'$ with $n \geq 0$ ($n + 1$ steps).

Confluence and termination

E is confluent (or Church-Rosser)

Any two rewritings of a term $t \in T_\Sigma$ can always be joined by further rewriting: if $t \to_E^* t_1$ and $t \to_E^* t_2$, then there exists a term t' such that $t_1 \to_E^* t'$ and $t_2 \to_E^* t'$.

$$
\begin{tikzcd}
 & t \arrow[dl, "*"', "E"] \arrow[dr, "*", "E"'] & \\
t_1 \arrow[dr, dotted, "*"', "E"] & & t_2 \arrow[dl, dotted, "*", "E"'] \\
 & t' &
\end{tikzcd}
$$

E is terminating

There is no infinite sequence of rewriting steps such as:
$t_0 \to_E t_1 \to_E t_2 \to_E \dots$

Maude functional modules

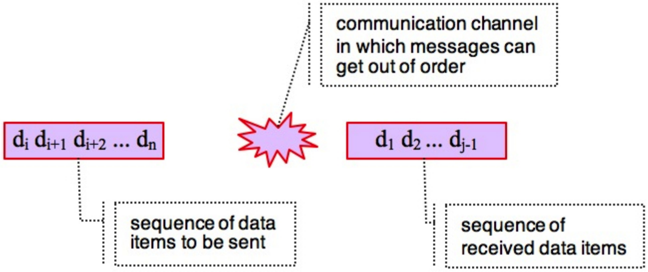
- If E is both confluent and terminating, a term t can be reduced to a unique normal or canonical form $t{\downarrow_E}$, that is, to a term that can no longer be rewritten.
- Checking semantic equality of two terms, $t = t'$, amounts to checking that their respective canonical forms are equal, $t{\downarrow_E} = t'{\downarrow_E}$.
- Functional modules in Maude are assumed to be confluent and terminating, and their operational semantics is equational simplification, that is, rewriting of terms until a canonical form is obtained.

Equational attributes

- Equational attributes are a means of declaring certain axioms in a way that allows Maude to use them efficiently in a built-in way: `assoc`, `comm`, `id`.
- Given an equational theory A, a pattern term t and a subject term u, we say that t matches u modulo A if there is a substitution σ such that $\sigma(t) =_A u$, that is, $\sigma(t)$ and u are equal modulo the equational theory A.
- Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operators, Maude synthesizes a combined matching algorithm for the theory A, and does equational simplification modulo the axioms A.

Example: an unordered communication channel

- Consider a communication channel in which messages can get out of order.
- There is a sender and a receiver.
- The sender is sending a sequence of data items, for example numbers.



In-order communication in an unordered channel

- The receiver is supposed to get the sequence in the exact same order in which they were in the sender's sequence.
- 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.
- The receiver sends back an ack indicating that has received the item.

In-order communication in an unordered channel

- The contents of the unordered channel are modeled as a multiset of messages of sort `Conf`(iguration).
- The entire system state is a 5-tuple of sort `State`, where the components are:
  - a buffer with the items to be sent,
  - a counter for the acknowledged items,
  - the contents of the unordered channel,
  - a buffer with the items received, and
  - a counter for the items received.

```
op {_,_|_|_,_} : List Nats Conf List Nats -> State [ctor] .
```

Example: unordered channel infrastructure

```
fmod UNORDERED-CHANNEL-EQ is
  sorts Nats List Msg Conf State .
  op 0 : -> Nats [ctor] .
  op s : Nats -> Nats [ctor] .
  op nil : -> List [ctor] .
  op _;_ : Nats List -> List [ctor] .   *** list cons
  op _@_ : List List -> List .          *** list append

  op [_,_] : Nats Nats -> Msg [ctor] .
  op ack : Nats -> Msg [ctor] .
  subsort Msg < Conf .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .

  op {_,_|_|_,_} : List Nats Conf List Nats -> State [ctor] .

  vars N : Nats .        vars L P : List .
  eq nil @ L = L .
  eq (N ; L) @ P = N ; (L @ P) .
endfm
```

Rewriting logic

- Rewriting logic was introduced by J. Meseguer in 1990 as a unifying framework for concurrency.
- We arrive at the main idea behind rewriting logic by dropping symmetry and the equational interpretation of rules.
- We interpret a rule $t \to t'$ computationally as a local concurrent transition of a system, and logically as an inference step from formulas of type t to formulas of type t'.
- Rewriting logic is a logic of becoming or change, that allows us to specify the dynamic aspects of systems.

Modeling systems in rewriting logic

- The static part is specified as an equational theory.
- The dynamics is specified by means of possibly conditional rules that rewrite terms, representing parts of the system, into others.
- The rules need only specify the part of the system that actually changes: the frame problem is avoided.

System modules

- System modules in Maude correspond to rewrite theories in rewriting logic.
- A rewrite theory has both rules and equations, so that rewriting is performed modulo such equations.
- The equations are divided into
  - a set A of structural axioms (associativity, commutativity, identity), for which matching algorithms exist in Maude, 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.

Example: unordered channel rules

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

  vars N M J : Nats .
  vars L P : List .
  var  C : Conf .

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

  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
```

Coherence

- The rules R in the module must be coherent with the equations E modulo A, 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.

$$
\begin{tikzcd}
t \arrow[r, "1", "R/A"'] \arrow[dd, "E/A"', "!"] & t' \arrow[dr, dotted, "!", "E/A"'] & \\
 & & w \\
u \arrow[r, dotted, "1", "R/A"'] & u' \arrow[ur, dotted, "!", "E/A"'] &
\end{tikzcd}
$$

- A simple strategy available when coherence holds is to always reduce to canonical form using E before applying any rule in R.

Tools around Maude

Sufficient completeness

SCC

Sort decreasingness

CRC

Termination

MTT

Confluence

Maude

Coherence

ChC

Execution

Theorem proving

ITP

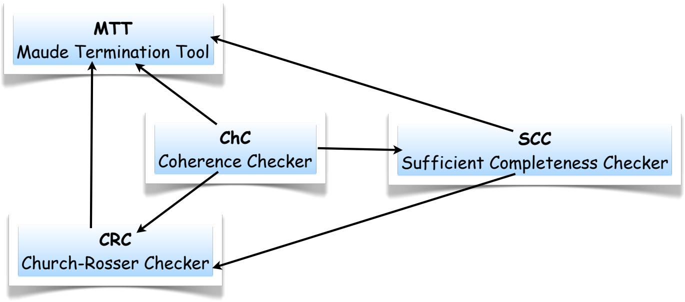
Model checking

Reachability analysis

Maude Formal Environment

- Maude Termination Tool (MTT) to prove termination of system modules by connecting to external termination tools.
- Church-Rosser Checker (CRC) to check the Church-Rosser property of functional modules.
- Sufficient Completeness Checker (SCC) to check that defined functions have been fully defined in terms of constructors.
- Coherence Checker (ChC) to check the coherence of system modules.
- Inductive Theorem Prover (ITP) to verify inductive properties of functional modules.

Maude Formal Environment



Example: termination and confluence

```
Maude> (select tool MTT .)
The MTT has been set as current tool.
Maude> (select external tool aprove .)
aprove is now the current external tool.

Maude> (ct UNORDERED-CHANNEL .)
Success: The module UNORDERED-CHANNEL is terminating.

Maude> (select tool CRC .)
The CRC has been set as current tool.

Maude> (ccr UNORDERED-CHANNEL .)
Church-Rosser check for UNORDERED-CHANNEL
All critical pairs have been joined.
The specification is locally-confluent.
The module is sort-decreasing.

Maude> (submit .)
The termination goal for the functional part of UNORDERED-CHANNEL
    has been submitted to MTT.
The functional part of module UNORDERED-CHANNEL has been checked
    terminating.
Success: The module is therefore Church-Rosser.
Success: The module UNORDERED-CHANNEL is Church-Rosser.
```

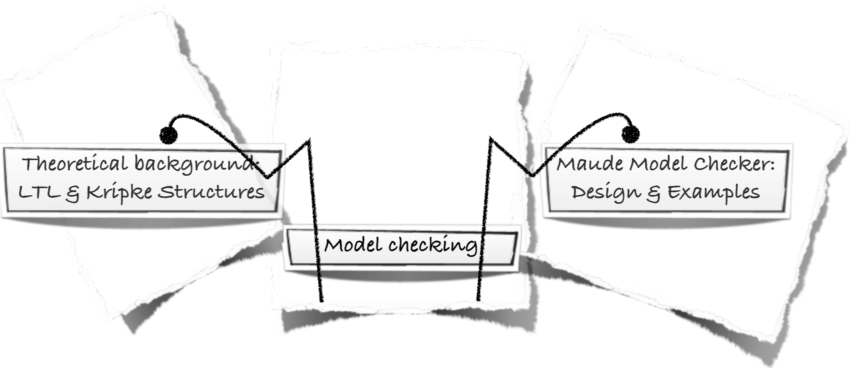
Example: sufficient completeness

```
Maude> (select tool SCC .)
The SCC has been set as current tool.

Maude> (scc UNORDERED-CHANNEL .)
Sufficient completeness check for UNORDERED-CHANNEL
Completeness counter-examples: none were found
Freeness counter-examples: none were found
Analysis: it is complete and it is sound
Ground weak termination: not proved
Ground sort-decreasingness: not proved

Maude> (submit .)
The sort-decreasingness goal for UNORDERED-CHANNEL has been
    submitted to CRC.
The termination goal for the functional part of UNORDERED-CHANNEL
    has been submitted to MTT.
Church-Rosser check for UNORDERED-CHANNEL
    The module is sort-decreasing.
Success: The functional module UNORDERED-CHANNEL is sufficiently
    complete and has free constructors.
```

Example: coherence

```
Maude> (select tool ChC .)
The ChC has been set as current tool.
Maude> (cch UNORDERED-CHANNEL .)
Coherence checking of UNORDERED-CHANNEL
    All critical pairs have been rewritten and no rewrite with
    rules can happen at non-overlapping positions of equations
    left-hand sides.
    The sufficient-completeness, termination and Church-Rosser
    properties must still be checked.

Maude> (submit .)
The Church-Rosser goal for UNORDERED-CHANNEL has been submitted
    to CRC.
The Sufficient-Completeness goal for UNORDERED-CHANNEL has been
    submitted to SCC.
The termination goal for the functional part of UNORDERED-CHANNEL
    has been submitted to MTT.
Sufficient completeness check for UNORDERED-CHANNEL
    [...]
Church-Rosser check for UNORDERED-CHANNEL
    [...]
The functional part of module UNORDERED-CHANNEL has been checked
    terminating.
The module UNORDERED-CHANNEL has been checked Church-Rosser.
Success: The module UNORDERED-CHANNEL is coherent.
```

Outline


Theoretical background:
LTL & Kripke Structures
Model checking
Maude Model Checker:
Design & Examples

Model checking

- Two levels of specification:
  - a system specification level, provided by the rewrite theory specified by a system module,
  - a property specification level, given by some properties that we want to state and prove about our module.
- Temporal logic allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens), related to the possibly infinite global behavior of a system.
- Maude 2 includes a model checker to prove properties expressed in linear temporal logic (LTL).

Linear temporal logic: syntax

- Main connectives:
  - True: $\top$
  - Atomic propositions: $p \in AP$
  - Next: $\bigcirc\varphi$
  - Until: $\varphi\,\mathcal{U}\,\psi$
  - Negation and disjunction: $\neg\varphi$, $\varphi \vee \psi$
- Derived connectives:
  - False: $\bot = \neg\top$
  - Conjunction: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - Implication: $\varphi \to \psi = (\neg\varphi) \vee \psi$
  - Eventually: $\Diamond\varphi = \top\,\mathcal{U}\,\varphi$
  - Henceforth: $\Box\varphi = \neg\Diamond\neg\varphi$

Linear temporal logic: intuition

- $\top$ is a formula that always holds at the current state.
- $\bigcirc\varphi$ holds at the current state if φ holds at the state that follows.
- $\varphi\,\mathcal{U}\,\psi$ holds at the current state if ψ is eventually satisfied at a future state and, until that moment, φ holds at all intermediate states.
- $\Box\varphi$ holds if φ holds at every state from now on.
- $\Diamond\varphi$ holds if φ holds at some state in the future.

Kripke structures

- A Kripke structure is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that
  - A is a set, called the set of states,
  - $\rightarrow_{\mathcal{A}}$ is a total binary relation on A, called the transition relation, and
  - $L : A \longrightarrow \mathcal{P}(AP)$ is a labeling function, associating to each state $a \in A$ the set $L(a)$ of those atomic propositions in AP that hold in a.
- A path in a Kripke structure $\mathcal{A}$ is a function $\pi : \mathbb{N} \longrightarrow A$ with $\pi(i) \rightarrow_{\mathcal{A}} \pi(i+1)$ for every i.
- π^i is the suffix of π starting at $\pi(i)$.

Linear temporal logic: semantics

- Satisfaction relation between a Kripke structure $\mathcal{A}$, a state $a \in A$, and an LTL formula $\varphi \in \mathrm{LTL}(AP)$:

$$\mathcal{A}, a \models \varphi \iff \mathcal{A}, \pi \models \varphi \quad \text{for all paths } \pi \text{ with } \pi(0) = a.$$

- Satisfaction relation for paths $\mathcal{A}, \pi \models \varphi$ defined by structural induction on φ:

$$\begin{array}{lcl}
\mathcal{A}, \pi \models p & \iff & p \in L(\pi(0)) \\
\mathcal{A}, \pi \models \top & \iff & \mathit{true} \\
\mathcal{A}, \pi \models \varphi \vee \psi & \iff & \mathcal{A}, \pi \models \varphi \text{ or } \mathcal{A}, \pi \models \psi \\
\mathcal{A}, \pi \models \neg\varphi & \iff & \mathcal{A}, \pi \not\models \varphi \\
\mathcal{A}, \pi \models \bigcirc\varphi & \iff & \mathcal{A}, \pi^1 \models \varphi \\
\mathcal{A}, \pi \models \varphi\, \mathcal{U}\, \psi & \iff & \text{there exists } n \in \mathbb{N} \text{ such that} \\
 & & \mathcal{A}, \pi^n \models \psi \text{ and,} \\
 & & \text{for all } m < n, \mathcal{A}, \pi^m \models \varphi
\end{array}$$

- The semantics of the remaining operators can be derived from these.

Kripke structures associated to rewrite theories

- Given a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, we
  - choose a type k in `M` as our type of states;
  - define in a module, say `M-PREDS`, protecting `M` some state predicates Π and their semanticsby means of the basic satisfaction operation

    ```
    op _|=_ : State Prop -> Bool .
    ```

- Then we get a Kripke structure (more details later)

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow^{1}_{\mathcal{R}})^{\bullet}, L_{\Pi}).$$

- Under some assumptions on `M` and `M-PREDS`, including that the set of states reachable from t is finite, the relation $\mathcal{K}(\mathcal{R}, k)_{\Pi}, t \models \varphi$ can be model checked.

Crossing the river: transitions

```
mod RIVER-CROSSING is
  sorts Side Group .

  ops left right : -> Side [ctor] .
  op change : Side -> Side .
  eq change(left) = right .
  eq change(right) = left .

  ops s w l c : Side -> Group [ctor] .
  op __ : Group Group -> Group [ctor assoc comm] .

  var S : Side .

  rl [shepherd] : s(S) => s(ch(S)) .
  rl [wdog] : s(S) w(S) => s(ch(S)) w(ch(S)) .
  rl [lamb] : s(S) l(S) => s(ch(S)) l(ch(S)) .
  rl [cabbage] : s(S) c(S) => s(ch(S)) c(ch(S)) .
endm
```

Crossing the river: properties

```
mod RIVER-CROSSING-PROP is
  protecting RIVER-CROSSING .
  including MODEL-CHECKER .
  subsort Group < State .
  op initial : -> Group .
  eq initial = s(left) w(left) l(left) c(left) .

  ops disaster success : -> Prop [ctor] .
  vars S S' S'' : Side .
  ceq (w(S) l(S) s(S') c(S'') |= disaster) = true if S =/= S' .
  ceq (w(S'') l(S) s(S') c(S) |= disaster) = true if S =/= S' .
  eq (s(right) w(right) l(right) c(right) |= success) = true .
  eq G:Group |= P:Prop = false [owise] .
endm
```

- success characterizes the (good) state in which the shepherd and his belongings are all in the other side,
- disaster characterizes the (bad) states in which some eating takes place.

Crossing the river

- The model checker only returns either `true` or paths that are counterexamples of properties.
- To find a safe path we need a formula that expresses the negation of the property we like: a counterexample will then witness a safe path for the shepherd.
- If no safe path exists, then it is true that whenever `success` is reached a disastrous state has been traversed before:

  ```
  <> success -> (<> disaster /\ ((~ success) U disaster))
  ```

  Equivalently

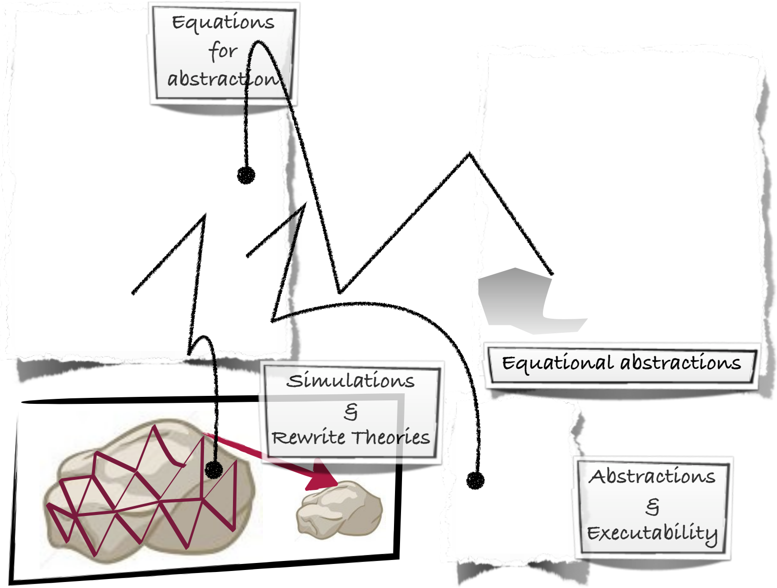
  ```
  <> success -> ((~ success) U disaster)
  ```

- A counterexample to this formula is a safe path, completed so as to have a cycle.

Crossing the river

```
Maude> red modelCheck(initial,
  <> success -> (<> disaster /\ ((~ success) U disaster)))

result ModelCheckResult: counterexample(
    {s(left) w(left) l(left) c(left),'lamb}
    {s(right) w(left) l(right) c(left),'shepherd}
    {s(left) w(left) l(right) c(left),'wdog}
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'cabbage}
    {s(right) w(right) l(left) c(right),'shepherd}
    {s(left) w(right) l(left) c(right),'lamb}
    {s(right) w(right) l(right) c(right),'lamb}
    {s(left) w(right) l(left) c(right),'shepherd}
    {s(right) w(right) l(left) c(right),'wdog}
    {s(left) w(left) l(left) c(right),'lamb}
    {s(right) w(left) l(right) c(right),'cabbage}
    {s(left) w(left) l(right) c(left),'wdog},
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'lamb})
```

Outline


Equations for abstraction
Simulations & Rewrite Theories
Equational abstractions
Abstractions & Executability

The problem

- Given a concurrent system, we want to check whether certain properties hold in it or not.
- If the number of (reachable) states is finite, use model checking.
- If the number of (reachable) states is infinite (or too large) this does not work. Then
  - we can employ deductive methods, or
  - we can calculate an abstract version of the system with a finite number of states to which model checking can be applied.

Our approach to abstraction

- A simple method of defining quotient abstractions is by means of equations collapsing the set of states:
- The concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$.
- Then the quotient is obtained by adding more equations to $\mathcal{R}$, thus getting $\mathcal{R}' = (\Sigma, E \cup E', R)$.
- Such a quotient will be useful for model-checking purposes if
  - the resulting theory is executable, and
  - the state predicates are preserved by the equations.
- These proof obligations can be discharged using the tools in the Maude Formal Environment.

Simulations between Kripke structures

- An *AP-simulation* $H : \mathcal{A} \longrightarrow \mathcal{B}$ between Kripke structures $\mathcal{A}$ and $\mathcal{B}$ over AP is a total relation $H \subseteq A \times B$ such that:
  - $$\begin{array}{ccc} a & \longrightarrow_{\mathcal{A}} & a' \\ H & & H \\ b & \longrightarrow_{\mathcal{B}} & b' \end{array}$$
  - If aHb then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.
- H is strict if the previous inclusion is an equality.
- $H : \mathcal{A} \longrightarrow \mathcal{B}$ reflects the satisfaction of a formula φ if

$$\mathcal{B}, b \models \varphi \text{ and } aHb \text{ implies } \mathcal{A}, a \models \varphi.$$

Theorem

AP-simulations reflect satisfaction of $\mathrm{LTL}^{-}(AP)$ formulas (where $\mathrm{LTL}^{-}(AP)$ is the negation-free fragment of LTL). Strict simulations reflect satisfaction of $\mathrm{LTL}(AP)$ formulas.

Minimal systems

- Often we only have a Kripke structure $\mathcal{M}$ and a surjective function to a set of abstract states $h : M \longrightarrow A$.
- The minimal system $\mathcal{M}^h_{\min}$ (over A) corresponding to $\mathcal{M}$ and h is defined by $(A, \rightarrow_{\mathcal{M}^h_{\min}}, L_{\mathcal{M}^h_{\min}})$, where:
  - $x \rightarrow_{\mathcal{M}^h_{\min}} y \iff \exists a.\exists b.(h(a) = x \wedge h(b) = y \wedge a \rightarrow_{\mathcal{M}} b)$
  - $L_{\mathcal{M}^h_{\min}}(a) = \bigcap_{x \in h^{-1}(a)} L_{\mathcal{M}}(x)$.

Theorem

$h : \mathcal{M} \longrightarrow \mathcal{M}^h_{\min}$ is indeed a simulation.

Minimal systems as quotients

- Minimal systems can also be seen as quotients.
- For a Kripke structure $\mathcal{A}$ and $\sim$ an equivalence relation on A, define $\mathcal{A}/\!\sim\; = (A/\!\sim, \rightarrow_{\mathcal{A}/\sim}, L_{\mathcal{A}/\sim})$, where:
  - $[a_1] \rightarrow_{\mathcal{A}/\sim} [a_2] \iff$
    $\exists a_1' \in [a_1].\ \exists a_2' \in [a_2].\ \ a_1' \rightarrow_{\mathcal{A}} a_2'$
  - $L_{\mathcal{A}/\sim}([a]) = \bigcap_{x \in [a]} L_{\mathcal{A}}(x)$.

Theorem

Given $\mathcal{M}$ and h surjective, the Kripke structures $\mathcal{M}^h_{\min}$ and $\mathcal{M}/\!\sim_h$ are isomorphic, where $x \sim_h y$ iff $h(x) = h(y)$.

Remarks on minimal systems

- The adjective minimal is appropriate since $\mathcal{M}^h_{\min}$ is the most accurate approximation to $\mathcal{M}$ consistent with h.
- It is not always possible to have a computable description of $\mathcal{M}^h_{\min}$.
- The transition relation:

$$x \rightarrow_{\mathcal{M}^h_{\min}} y \iff \exists a.\exists b.(h(a) = x \wedge h(b) = y \wedge a \rightarrow_{\mathcal{M}} b)$$

  is not recursive in general.
- Here we present methods that, when successful, yield a computable description of $\mathcal{M}^h_{\min}$.

The system specification level

- In general, a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with:
  - (Σ, E) an equational theory describing the states;
  - R a set of (conditional) rewrite rules defining the system transitions.
- This determines, for each type k, a transition system

$$(T_{\Sigma/E,k}, (\to^1_{\mathcal{R}})^\bullet)$$

  where
  - $T_{\Sigma/E,k}$ is the set of equivalence classes $[t]$ of terms of type k, modulo the equations E;
  - $(\to^1_{\mathcal{R}})^\bullet$ completes the one-step rewrite relation $\to^1_{\mathcal{R}}$ with an identity pair $([t],[t])$ for each deadlock state $[t]$, to get a total relation.

LTL properties of rewrite theories

- LTL properties are associated to $\mathcal{R}$ and a type k by specifying the basic state predicates Π in an equational theory $(\Sigma', E \cup D)$ extending (Σ, E) conservatively.
- State predicates, possibly parameterized, are constructed with operators $p : s_1 \dots s_n \to Prop$.
- The semantics is defined by means of equations D using the basic “satisfaction operator” $\_ \models \_ : k\ Prop \to Bool$.
- A state predicate $p(u_1, \dots, u_n)$ holds in a state $[t]$ iff

$$E \cup D \vdash \quad t \models p(u_1, \dots, u_n) = true$$

LTL properties of rewrite theories

- The Kripke structure associated to $\mathcal{R}$, k, and Π, with atomic propositions

$$AP_\Pi = \{p(u_1, \ldots, u_n) \text{ ground} \mid p \in \Pi\}$$

is then defined as

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi)$$

where

$$L_\Pi([t]) = \{p(u_1, \ldots, u_n) \mid p(u_1, \ldots, u_n) \text{ holds in } [t]\}$$

- Assuming that the equations $E \cup D$ are Church-Rosser and terminating, and that the rewrite theory $\mathcal{R}$ is executable, the resulting Kripke structure is indeed computable.

Equational abstractions

- We can define an abstraction for $\mathcal{K}(\mathcal{R}, k)_\Pi$ by specifying an equational theory extension

$$(\Sigma, E) \subseteq (\Sigma, E \cup E')$$

- This gives rise to an equivalence relation $\equiv_{E'}$ on $T_{\Sigma/E}$

$$[t]_E \equiv_{E'} [t']_E \iff E \cup E' \vdash t = t' \iff [t]_{E \cup E'} = [t']_{E \cup E'}$$

  and therefore a quotient abstraction $\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'}$.
- Question: Is $\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'}$ the Kripke structure associated to another rewrite theory?

Equational abstractions

- We focus on those rewrite theories $\mathcal{R}$ satisfying the following requirements:
  - $\mathcal{R}$ is k-deadlock free, that is $(\rightarrow^1_{\mathcal{R}})^\bullet = \rightarrow^1_{\mathcal{R}}$ on $T_{\Sigma/E,k}$,
  - $\mathcal{R}$ is k-topmost, so k only appears as the coarity of a certain operator $f : k_1 \dots k_n \longrightarrow k$, and
  - no terms of type k appear in the conditions.
- A rewrite theory $\mathcal{R}$ can often be transformed into an equivalent one satisfying these requirements.
- The unordered channel example satisfies these requirements.

Equational abstractions

- Let us take a closer look at the quotient:

$$\mathcal{K}(\mathcal{R}, k)_\Pi/{\equiv_{E'}} = (T_{\Sigma/E,k}/{\equiv_{E'}}, (\rightarrow^1_{\mathcal{R}})^\bullet/{\equiv_{E'}}, L_{\Pi/\equiv_{E'}}).$$

- $T_{\Sigma/E}/{\equiv_{E'}} \cong T_{\Sigma, E\cup E'}$.
- Under the above assumptions, $\mathcal{R}/E' = (\Sigma, E \cup E', R)$ is k-deadlock free and

$$(\rightarrow^1_{\mathcal{R}/E'})^\bullet = \rightarrow^1_{\mathcal{R}/E'} = (\rightarrow^1_{\mathcal{R}})^\bullet/{\equiv_{E'}}$$

- Therefore, at a purely mathematical level, $\mathcal{R}/E'$ seems to be what we want.

Equational abstractions: executability

- Executability requires that:
  - The equations $E \cup E'$ are ground Church-Rosser and terminating.
  - The rules R are (ground) coherent relative to $E \cup E'$.
- For example, the rules

$$a \longrightarrow c \qquad b \longrightarrow d$$

are not coherent relative to the abstraction

$$a = b\,.$$

- To check and enforce these conditions, and get an executable rewrite theory $\mathcal{R}'$ semantically equivalent to $\mathcal{R}/E'$, we can use some Maude tools.

Equational abstractions: preservation of properties

- What about state predicates? By definition:

$$L_{\Pi/\equiv_{E'}}([t]_{E\cup E'}) = \bigcap_{[x]_E \subseteq [t]_{E\cup E'}} L_\Pi([x]_E).$$

- Coming up with equations D' defining $L_{\Pi/\equiv_{E'}}$ may not be easy.
- It becomes easy if the predicates are preserved by E':

$$[x]_{E\cup E'} = [y]_{E\cup E'} \implies L_\Pi([x]_E) = L_\Pi([y]_E)$$

- In this case we do not need to change the equations D and therefore we have:

$$\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'} \cong \mathcal{K}(\mathcal{R}/E', k)_\Pi.$$

Equational abstractions: preservation of properties

- How can we prove

$$[x]_{E \cup E'} = [y]_{E \cup E'} \implies L_\Pi([x]_E) = L_\Pi([y]_E) \ ?$$

Theorem

If the equations in E' are of the form $t = t'$ if C, with t, t' of type k, and for each such equation

$$E \cup D \ \vdash_{ind} \ \forall \vec{x}.\, \forall \vec{y}.\, C \Rightarrow \\ (t(\vec{x}) \models p(\vec{y}) = true \ \Leftrightarrow \ t'(\vec{x}) \models p(\vec{y}) = true)$$

then the state predicates Π are preserved by E'.

- Instead, we can use tools in the Maude Formal Environment to mechanically discharge these proof obligations.

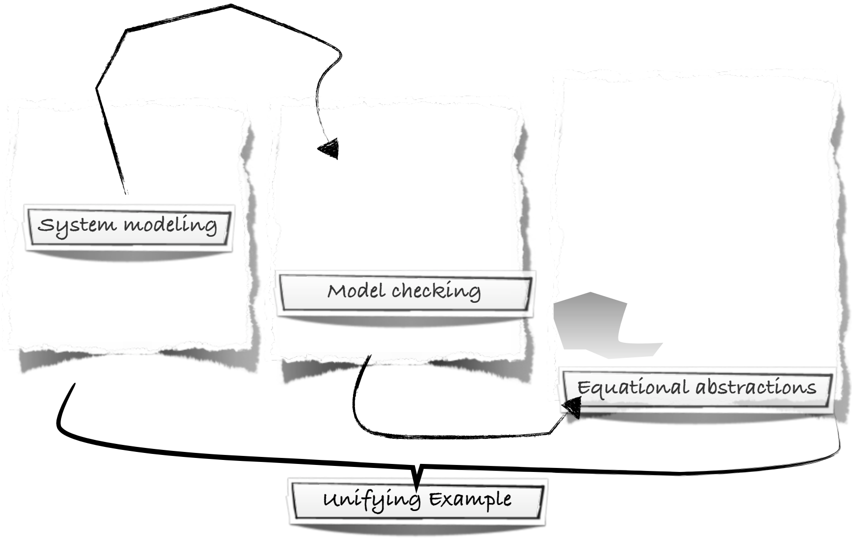
Equational abstractions: all together

- When E, E' and R satisfy the executability requirements described above,
- by construction, the quotient simulation

$$\mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}, E)_\Pi/{\equiv_{E'}} \cong \mathcal{K}(\mathcal{R}/E', k)_\Pi$$

  is strict, so it reflects satisfaction of arbitrary LTL formulas.
- Since $\mathcal{R}/E'$ is executable, for an initial state t having a finite set of reachable states we can use the Maude model checker to check if a property holds.

Outline


System modeling
Model checking
Equational abstractions
Unifying Example

Communication channel's transitions

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

  vars N M J : Nats .
  vars L P : List .
  var  C : Conf .

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

  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
```

Communication channel: abstraction

The channel should not contain repeated copies of sent messages:

```
mod UNORDERED-CHANNEL-ABSTRACTION is
  including UNORDERED-CHANNEL .
  vars M N P K : Nats .
  vars 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
```

Abstraction: termination and confluence

```
Maude> (ct UNORDERED-CHANNEL-ABSTRACTION .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION is terminating.

Maude> (ccr UNORDERED-CHANNEL-ABSTRACTION .)
[...]
Maude> (submit .)
[...]
Success: The module UNORDERED-CHANNEL-ABSTRACTION is Church-Rosser.

Maude> (scc UNORDERED-CHANNEL-ABSTRACTION .)
[...]
Warning: The functional module UNORDERED-CHANNEL-ABSTRACTION is
    sufficiently complete and has free constructors. However‘,
    module UNORDERED-CHANNEL-ABSTRACTION may still not be
    sufficiently complete or not have free constructors.
```

Abstraction: coherence

```
Maude> (select tool ChC .)
The ChC has been set as current tool.

Maude> (cch UNORDERED-CHANNEL-ABSTRACTION .)
Coherence checking of UNORDERED-CHANNEL-ABSTRACTION
The following critical pairs cannot be rewritten:
  cp UNORDERED-CHANNEL-ABSTRACTION2 for A1 and rec
    {L:List,M:Nats | #3:Conf[N:Nats,J:Nats]| P:List,J:Nats}
    =>{L:List,M:Nats | #3:Conf ack(J:Nats)[N:Nats,J:Nats]|
    P:List @ N:Nats ; nil,s(J:Nats)}.
   The sufficient-completeness, termination and Church-Rosser
    properties must still be checked.
```

Abstraction: recovering coherence

In this example, the critical pair indicates that a rule is missing.

```
mod UNORDERED-CHANNEL-ABSTRACTION-2 is
  including UNORDERED-CHANNEL-ABSTRACTION .

  vars M N K : Nats .
  vars L L' : List .
  var C : Conf .

  rl [snd2]: {L, M | [N, K] C | L', K}
         => {L, M | [N, K] ack(K) C | L' @ N ; nil, s(K)} .
endm
```

Abstraction: coherence recovered

```
Maude> (select tool ChC .)
The ChC has been set as current tool.

Maude> (cch UNORDERED-CHANNEL-ABSTRACTION-2 .)
Coherence checking of UNORDERED-CHANNEL-ABSTRACTION-2
    All critical pairs have been rewritten and no rewrite with
    rules can happen at non-overlapping positions of equations
    left-hand sides.
    The sufficient-completeness, termination and Church-Rosser
    properties must still be checked.

Maude> (submit .)
[...]
Success: The module UNORDERED-CHANNEL-ABSTRACTION-2 is coherent.
```

Communication channel: properties

- We assume that all initial states are of the form:

```
{n1 ; ... ; nk ; nil , 0 | null | nil , 0}
```

- The sender's buffer contains a list of numbers

```
n1 ; ... ; nk ; nil
```

  and has the counter set to 0.
- The communication channel initially is empty.
- The receiver's buffer is also empty and the receiver's counter is initially set to 0.
- One essential property is that it achieves in-order communication in spite of the unordered channel.

Communication channel: properties

```
mod UNORDERED-CHANNEL-PREDS is
  protecting BOOLEAN .
  protecting UNORDERED-CHANNEL .

  sort Prop .
  op _~_ : Nats Nats -> Bool .     *** equality predicate
  op _|=_ : State Prop -> Bool [frozen] .  *** satisfaction

  vars M N K P : Nats .    vars L L' L'' : List .
  var  C : Conf .

  eq 0 ~ 0 = true .
  eq 0 ~ s(N) = false .
  eq s(N) ~ 0 = false .
  eq s(N) ~ s(M) = N ~ M .

  op prefix : List -> Prop [ctor] .

  eq [I1]: {L', N | C | K ; L'', P} |= prefix(M ; L)
    = (M ~ K) and  {L', N | C | L'', P} |=  prefix(L) .
  eq [I3]: {L', N | C | nil, K} |=  prefix(L) = true .
  eq [I4]: {L', N | C | M ; L'', K} |=  prefix(nil) = false .
endm
```

Communication channel: properties preservation

```
mod UNORDERED-CHANNEL-ABSTRACTION-CHECK is
    extending UNORDERED-CHANNEL-ABSTRACTION-2 .
    including UNORDERED-CHANNEL-PREDS .
    op init : -> State .
    eq init = {0 ; s(0) ; s(s(0)) ; nil , 0 | null | nil , 0} .
endm
```

- The set of abstract states is finite.
- The equations in both UNORDERED-CHANNEL-PREDS and UNORDERED-CHANNEL-ABSTRACTION-CHECK are Church-Rosser and terminating.
- The equations in both UNORDERED-CHANNEL-PREDS and UNORDERED-CHANNEL-ABSTRACTION-CHECK are sufficiently complete.
- UNORDERED-CHANNEL is deadlock free.

Communication channel: properties preservation

```
Maude> (ct UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-CHECK is
    terminating.

Maude> (ccr UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Maude> (submit .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-CHECK is
    Church-Rosser.

Maude> (scc UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Maude> (submit .)
Warning: The functional module UNORDERED-CHANNEL-ABSTRACTION-CHECK
   is sufficiently complete and has free constructors. However [...]

Maude> (cch UNORDERED-CHANNEL-ABSTRACTION-CHECK .)
Maude> (submit .)
Success: The module UNORDERED-CHANNEL-ABSTRACTION-CHECK is
    coherent.
```

Communication channel: model checking, finally

```
mod UNORDERED-CHANNEL-ABSTRACTION-MODEL-CHECK is
   including UNORDERED-CHANNEL-ABSTRACTION-CHECK .
   including LTL-SIMPLIFIER .   *** optional
   including MODEL-CHECKER .
endm

Maude> reduce in UNORDERED-CHANNEL-ABSTRACTION-MODEL-CHECK :
     modelCheck(init, []prefix(0 ; s(0) ; s(s(0)) ; nil)) .
rewrites: 361 in 41ms cpu (42ms real) (8780 rewrites/second)
result Bool: true
```

Concluding remarks

- The equational abstraction technique is fairly simple and takes advantage of the expressiveness of rewriting logic as well as of the tools available in the Maude Formal Environment.
- Other examples are available in the references, but without using the Maude Formal Environment in its current integrated form.
- Related work: Generalization of the equational theory extension $(\Sigma, E) \subseteq (\Sigma, E \cup E')$ to theory interpretations $(\Sigma, E) \longrightarrow (\Sigma', E'')$ and to (stuttering) simulations.
- Future work: improving the interface of the Maude Formal Environment to make it more user-friendly.
- In particular, the Inductive Theorem Prover (ITP) needs more and better integration with the other tools.

References

- J. Meseguer, M. Palomino, N. Martí-Oliet: Equational abstractions. *TCS* 403(2-3): 239-264 (2008).
- J. Meseguer, M. Palomino, N. Martí-Oliet: Algebraic simulations. *JLAP* 79(2): 103-143 (2010).
- F. Durán, J. Meseguer: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *JLAP* 81(7-8): 816-850 (2012).
- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. *Springer LNCS* 4350, 2007.