A logical approach to the verification of concurrent systems

(joint work with many colleagues)

by

Narciso Martí-Oliet (UCM)

August 2017

Introduction

Goals

1. To introduce Maude as a framework for modeling concurrent 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 illustrate this method with several detailed examples.
4. To comment recent developments introducing new features like narrowing and SMT constraints.

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.

Maude in a nutshell

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, or executability, requirements.

So … who is Maude?

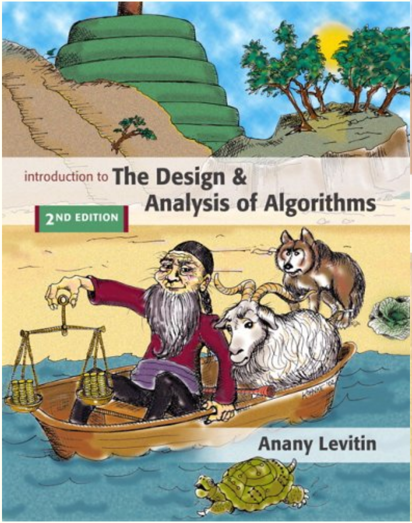
- Maude follows a long tradition of declarative algebraic specification languages in the OBJ family, including OBJ3, CafeOBJ, and Elan.
- Computation = Deduction in the appropriate logic.
- Functional modules = Admissible specifications in (membership) equational logic.
- System modules = Admissible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

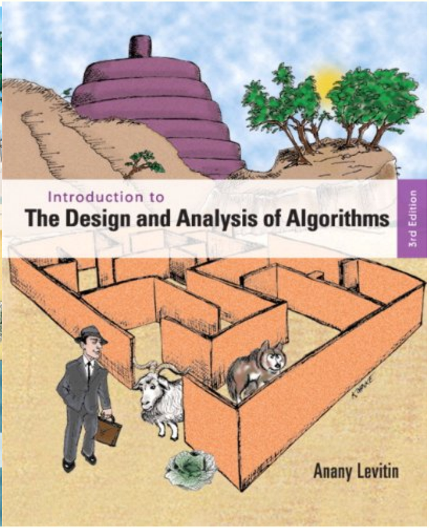
`http://maude.cs.uiuc.edu`

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
```

Example: mutual exclusion

```
mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: none] .

  ops a b : -> Name [ctor] .
  ops wait critical : -> Mode [ctor] .
  op [_,_] : Name Mode -> Proc [ctor] .
  ops * $ : -> Token [ctor] .

  rl [a-enter] : $ [a, wait] => [a, critical] .
  rl [b-enter] : * [b, wait] => [b, critical] .
  rl [a-exit] : [a, critical] => [a, wait] * .
  rl [b-exit] : [b, critical] => [b, wait] $ .
endm
```

Example: readers and writers

```
mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  --- natural numbers in Peano notation

  sort State .
  op <_,_> : Nat Nat -> State [ctor] .
             --- readers/writers

  vars R W : Nat .
  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm
```

Equational simplification

- A term t rewrites to a term t' (denoted $t \rightarrow_E t'$) 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 $\sigma(r)$.
- That is,

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

- We write $t \rightarrow_E^* t'$ to mean that 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

- A set of equations E is confluent (or Church-Rosser) when any two rewritings of a term can always be joined by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.

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

- A set of equations E is terminating when there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \ldots$

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.

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.

Maude 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.

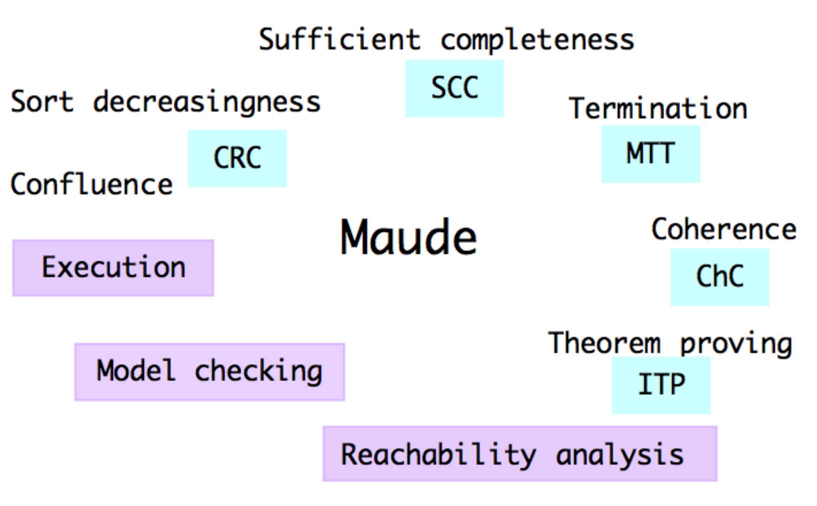
Coherence

- Rules *R* in the module must be coherent wrt. equations *E* modulo *A*, allowing us to intermix rewriting with rules and rewriting with equations without losing computations.

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

- 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



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.

Model checking

Model checking

- Two levels of specification:
  - a system specification level, provided by the rewrite theory specified by that system module, and
  - 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 infinite 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: $\neg\varphi, \varphi \vee \psi$

Linear temporal logic: syntax

- Derived connectives:
  - False: $\bot = \neg\top$
  - Conjunction: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - Implication: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$
  - 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 $\varphi \in LTL(AP)$:

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

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

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

Kripke structs for 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 semantics by means of the basic satisfaction operation

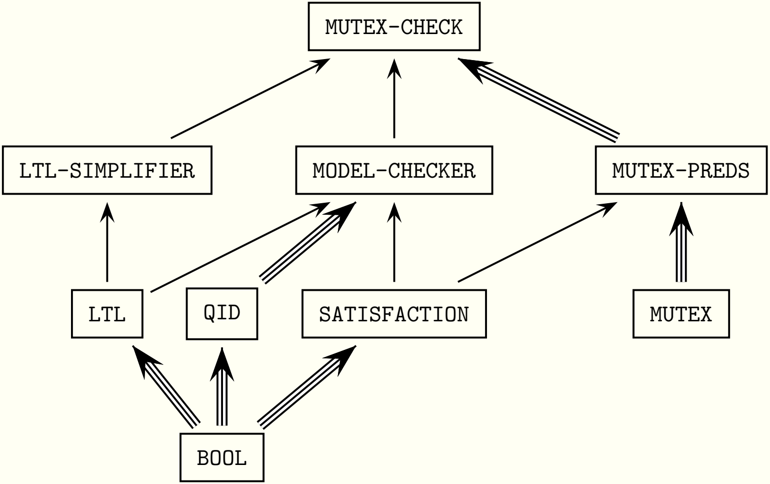
    ```
    op _|=_ : State Prop -> Bool .
    ```

- 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.

Model-checking modules



Mutual exclusion: processes

```
mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: none] .

  ops a b : -> Name [ctor] .
  ops wait critical : -> Mode [ctor] .
  op [_,_] : Name Mode -> Proc [ctor] .
  ops * $ : -> Token [ctor] .

  rl [a-enter] : $ [a, wait] => [a, critical] .
  rl [b-enter] : * [b, wait] => [b, critical] .
  rl [a-exit] : [a, critical] => [a, wait] * .
  rl [b-exit] : [b, critical] => [b, wait] $ .
endm
```

Mutual exclusion: properties

```
mod MUTEX-PREDS is
  protecting MUTEX .
  including SATISFACTION .

  subsort Conf < State .

  ops crit wait : Name -> Prop [ctor] .

  var N : Name .   var C : Conf .   var P : Prop .

  eq [N, critical] C |= crit(N) = true .
  eq [N, wait] C |= wait(N) = true .
  eq C |= P = false [owise] .
endm
```

Model checking mutual exclusion

```
mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a, wait] [b, wait] .
  eq initial2 = * [a, wait] [b, wait] .
endm

Maude> red modelCheck(initial1, [] ~(crit(a) /\ crit(b))) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true
```

Counterexamples

- If we check whether, beginning in the state `initial1`, process b will always be waiting, we get a counterexample:

```
Maude> red modelCheck(initial1, [] wait(b)) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 4 system states.

result ModelCheckResult:
  counterexample({$ [a, wait] [b, wait], 'a-enter}
                 {[a, critical] [b, wait], 'a-exit}
                 {* [a, wait] [b, wait], 'b-enter} ,
                 {[a, wait] [b, critical], 'b-exit}
                 {$ [a, wait] [b, wait], 'a-enter}
                 {[a, critical] [b, wait], 'a-exit}
                 {* [a, wait] [b, wait], 'b-enter})
```

Crossing the river: transitions

```
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
```

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 can take 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))
```

- 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})
```

Equational abstractions

Readers and writers: transitions

```
mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  sort State .
  op <_,_> : Nat Nat -> State [ctor] .
           --- readers/writers

  vars R W : Nat .
  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .  --- infinite system
  rl < s(R), W > => < R, W > .
endm
```

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.

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 rewrite rules defining the system transitions.
- This determines, for each type k, a transition system

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

where

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

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 $p : s_1 \ldots s_n \to Prop$.
- The semantics is defined by means of equations D using the basic "satisfaction operator" $\_ \models \_ : k\, Prop \to Bool$.
- A state predicate $p(u_1, \ldots, u_n)$ holds in a state $[t]$ iff

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

LTL properties of rewrite theories

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

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

where

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

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

- Assuming that the equations $E \cup D$ are Church-Rosser and terminating, and that the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is executable, the resulting Kripke structure is 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 then a quotient abstraction $\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'}$.
- In what follows, we assume that $\mathcal{R}$ is k-deadlock free and k-topmost. In particular, $(\rightarrow^1_{\mathcal{R}})^\bullet = \rightarrow^1_{\mathcal{R}}$.

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, $\mathcal{R}' = (\Sigma, E \cup E', R)$ is also k-deadlock free and

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

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

Equational abstractions:

- 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 at all.
- It becomes much easier if the predicates are preserved by E':

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

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

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

Equational abstractions: all together

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

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

  is strict and so it reflects satisfaction of arbitrary LTL formulas.
- Moreover, since $\mathcal{R}'$ 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.

Readers and writers

Readers and writers: transitions

```
mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  sort State .
  op <_,_> : Nat Nat -> State [ctor] .
             --- readers/writers

  vars R W : Nat .
  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .  --- infinite system
  rl < s(R), W > => < R, W > .
endm
```

Readers and writers: properties

```
mod READERS-WRITERS-PREDS is
  protecting READERS-WRITERS .
  including SATISFACTION .
  ops mutex one-writer : -> Prop [ctor] .
  eq < s(N:Nat), s(M:Nat) > |= mutex = false .
  eq < 0, N:Nat > |= mutex = true .
  eq < N:Nat, 0 > |= mutex  = true .
  eq < N:Nat, s(s(M:Nat)) > |= one-writer = false .
  eq < N:Nat, 0 > |= one-writer = true .
  eq < N:Nat, s(0) > |= one-writer  = true .
endm
```

- mutual exclusion: readers and writers never access the resource simultaneously: only readers or only writers can do so at any given time.
- one writer: at most one writer will be able to access the resource at any given time.

Readers and writers: abstraction

```
mod READERS-WRITERS-ABS is
  including READERS-WRITERS-PREDS .
  including READERS-WRITERS .
  eq < s(s(N:Nat)), 0 > = < s(0), 0 > .
endm
```

- The exact number of readers is unimportant.
- We are only interested in whether there is at least a reader or not.

Readers and writers: abstraction

- For the executability and the property-preservation properties of this abstraction, we need to check:
  1. that the equations in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are (ground) Church-Rosser and terminating;
  2. that the equations in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are sufficiently complete (this is equivalent to requiring that properties are preserved); and
  3. that the rules in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are ground coherent with respect to their equations.

Readers and writers: Church-Rosser

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

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

Readers and writers: completeness

```
Maude> (scc READERS-WRITERS-PREDS .)
Sufficient completeness check for READERS-WRITERS-PREDS ...
     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> (scc READERS-WRITERS-ABS .)
Sufficient completeness check for READERS-WRITERS-ABS ...
     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
```

Readers and writers: coherence

```
Maude> (cch READERS-WRITERS-PREDS .)
Coherence checking of READERS-WRITERS-PREDS
Coherence checking solution:
  All critical pairs have been rewritten and all equations
    are non-constructor.
  The specification is coherent.

Maude> (check coherence READERS-WRITERS-ABS .)
Coherence checking of READERS-WRITERS-ABS
Coherence checking solution:
  The following critical pairs cannot be rewritten:
  cp < s(0), 0 > => < s(N:Nat), 0 > .
```

- A simple argument by cases shows that this critical pair can be joined for each instantiation of N by considering the two cases for natural numbers N = `0` and N = `s(M)`, thus proving ground coherence.

Readers and writers: finally

```
mod READERS-WRITERS-ABS-CHECK is
  protecting READERS-WRITERS-ABS .
  including MODEL-CHECKER .
endm
```

```
Maude> reduce in READERS-WRITERS-ABS-CHECK :
         modelCheck(< 0,0 >, []mutex) .
rewrites: 15 in 0ms cpu (0ms real) (28790 rewrites/second)
result Bool: true
```

```
Maude> reduce in READERS-WRITERS-ABS-CHECK :
         modelCheck(< 0,0 >, []one-writer) .
rewrites: 15 in 0ms cpu (0ms real) (76142 rewrites/second)
result Bool: true
```

Readers and writers: search

```
Maude> search in READERS-WRITERS-ABS :
         < 0, 0 > =>* C:State
         such that C:State |= mutex = false .

No solution.
states: 3
rewrites: 9 in 0ms cpu (0ms real) (80357 rewrites/second)


Maude> search in READERS-WRITERS-ABS :
         < 0, 0 > =>* C:State
         such that C:State |= one-writer = false .

No solution.
states: 3
rewrites: 9 in 0ms cpu (0ms real) (94736 rewrites/second)
```

Bakery protocol

A bakery protocol example

- It is an infinite-state protocol that achieves mutual exclusion between processes by the usual method in bakeries and deli shops: there is a number dispenser and customers are served according to the number they hold.
- Consider a simple Maude specification for the case of two processes, where a state is represented by a tuple

```
op <_,_,_,_> : Mode Nat Mode Nat -> State .
```

Bakery protocol: processes

```
mod BAKERY is
  protecting NAT .
  sorts Mode State .
  ops sleep wait crit : -> Mode [ctor] .
  op <_,_,_,_> : Mode Nat Mode Nat -> State [ctor] .
  op initial : -> State .
  vars P Q : Mode .     vars X Y : Nat .
  eq initial = < sleep, 0, sleep, 0 > .
  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  crl [p1_wait] : < wait, X, Q, Y > => < crit, X, Q, Y >
                  if not (Y < X) .
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .
  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, s X > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
  crl [p2_wait] : < P, X, wait, Y > => < P, X, crit, Y >
                  if Y < X .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm
```

Bakery protocol: properties

- mutual exclusion: the two processes are never simultaneously in their critical section.

  ```
  []~ (1crit /\ 2crit )
  ```

- liveness: whenever a process enters the waiting mode, it will eventually enter its critical section.

  ```
  (1wait |-> 1crit) /\ (2wait |-> 2crit)
  ```

Bakery protocol: properties

```
mod BAKERY-PREDS is
  protecting BAKERY .
  including SATISFACTION .
  ops 1wait 2wait 1crit 2crit : -> Prop [ctor] .
  vars P Q : Mode .
  vars X Y : Nat .
  eq < wait, X, Q, Y > |= 1wait = true .
  eq < sleep, X, Q, Y > |= 1wait = false .
  eq < crit, X, Q, Y > |= 1wait = false .
  eq < P, X, wait, Y > |= 2wait = true .
  eq < P, X, sleep, Y > |= 2wait = false .
  eq < P, X, crit, Y > |= 2wait = false .
  eq < crit, X, Q, Y > |= 1crit = true .
  eq < sleep, X, Q, Y > |= 1crit = false .
  eq < wait, X, Q, Y > |= 1crit = false .
  eq < P, X, crit, Y > |= 2crit = true .
  eq < P, X, sleep, Y > |= 2crit = false .
  eq < P, X, wait, Y > |= 2crit = false .
endm
```

Bakery protocol: abstraction

- We can define an abstraction by:

```
abs(< P, X, Q, Y >)
        =
< P, Q, X == 0, Y == 0, Y < X >
```

- Equivalently:

$$\langle P, N, Q, M\rangle \equiv \langle P', N', Q', M'\rangle$$

iff

  - $P = P'$ and $Q = Q'$,
  - $N = 0$ iff $N' = 0$,
  - $M = 0$ iff $M' = 0$,
  - $M < N$ iff $M' < N'$.

Bakery protocol: abstraction

```
mod ABSTRACT-BAKERY is
  including BAKERY  .
  vars P Q : Mode .
  vars X Y : Nat .

  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
  eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
  eq < P, s s s X, Q, s s s Y > = < P, s s X, Q, s s Y > .
  eq < P, s s s X, Q, s s 0 > = < P, s s 0, Q, s 0 > .
  eq < P, s s s X, Q, s 0 > = < P, s s 0, Q, s 0 > .
  eq < P, s s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
  eq < P, s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
endm
```

Bakery protocol: checking

- The set of abstract states is finite.
- The equations in both ABSTRACT−BAKERY and BAKERY−PREDS are (ground) Church-Rosser and terminating.
- The equations in both ABSTRACT−BAKERY and BAKERY−PREDS are sufficiently complete (this guarantees that NAT and BOOL are really protected).
- The rules (coming from BAKERY) and the equations in ABSTRACT−BAKERY are ground coherent.

Bakery protocol: all together

```
mod ABSTRACT-BAKERY-PREDS is
  protecting ABSTRACT-BAKERY .
  including BAKERY-PREDS .
endm
```

- The equations in ABSTRACT−BAKERY−PREDS are (ground) Church-Rosser and terminating.
- The equations in ABSTRACT−BAKERY−PREDS are sufficiently complete.
- This guarantees the required preservation of properties.

Bakery protocol: deadlock freedom

```
eq enabled(< sleep, X, Q, Y >) = true .
eq enabled(< wait, X, Q, 0 >) = true .
ceq enabled(< wait, X, Q, Y >) = true if not (Y < X) .
eq enabled(< crit, X, Q, Y >) = true .
eq enabled(< P, X, sleep, Y >) = true .
eq enabled(< P, 0, wait, Y >) = true .
ceq enabled(< P, X, wait, Y >) = true if Y < X .
eq enabled(< P, X, crit, Y >) = true .
```

- The equations for the `enabled` predicate are sufficiently complete (although the current version of the SCC tool does not help in proving this).
- This implies that the system is deadlock-free.

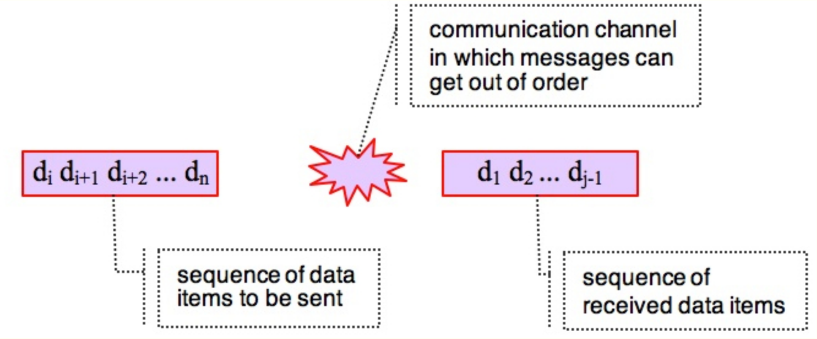
Bakery protocol: finally

```
mod ABSTRACT-BAKERY-CHECK is
  protecting ABSTRACT-BAKERY-PREDS .
  including MODEL-CHECKER .
endm

Maude> reduce in ABSTRACT-BAKERY-CHECK :
         modelCheck(initial, []~ (1crit /\ 2crit)) .

result Bool: true

Maude> reduce in ABSTRACT-BAKERY-CHECK :
         modelCheck(initial, (1wait |-> 1crit)
                          /\ (2wait |-> 2crit)) .

result Bool: true
```

Unordered communication channel

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

- The receiver is supposed to get the sequence in the exact same order in which items 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.

Unordered channel infrastructure

- 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] .
```

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
```

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
```

Unordered channel: Church-Rosser

```
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.
```

Unordered channel: completeness

```
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.
```

Unordered channel: coherence

```
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. [...]

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.
```

Unordered channel: 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
```

Unordered 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
```

Channel abstraction: coherence

```
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> (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-Rosse
     properties must still be checked.

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

Unordered 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.

Unordered 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
```

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.

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-CI
   is sufficiently complete and has free constructors. [...]

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

Unordered channel: 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
```

More features

Unification

- Given terms t and u, we say that t and u are unifiable if there is a substitution σ such that $\sigma(t) \equiv \sigma(u)$.
- Given an equational theory A and terms t and u, we say that t and u are unifiable modulo A if there is a substitution σ such that $\sigma(t) \equiv_A \sigma(u)$.
- Maude supports order-sorted equational unification modulo many combinations of equational attributes such as `assoc`, `comm id` (as well as variant-based equational unification).

Narrowing

- Narrowing generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification instead of matching in order to (non–deterministically) reduce a term.
- A term t narrows to a term t' using a rule $l \Rightarrow r$ in R and a substitution σ if
  1. there is a subterm $t|_p$ of t at a nonvariable position p of t such that l and $t|_p$ are unifiable via σ, and
  2. $t' = \sigma(t[r]_p)$ is obtained from $\sigma(t)$ by replacing the subterm $\sigma(t|_p) \equiv \sigma(l)$ with the term $\sigma(r)$.

Narrowing

- Narrowing can also be defined modulo an equational theory.
- Narrowing with R modulo E requires E-unification at each narrowing step.
- Maude supports a version of narrowing modulo with simplification, where each narrowing step with a rule is followed by simplification with the equations.
- There are some restrictions on the allowed rules; for example, they cannot be conditional.

Narrowing reachability analysis

- Narrowing can be used as a general deductive procedure for solving reachability problems of the form

$$(\exists \vec{x})\ t_1(\vec{x}) \to t_1'(\vec{x})\ \wedge \cdots \wedge\ t_n(\vec{x}) \to t_n'(\vec{x})$$

  in a given rewrite theory.
  - he terms t_i and t_i' denote sets of states.
  - For what subset of states denoted by t_i are the states denoted by t_i' reachable?
  - No finiteness assumptions about the state space.
  - Sound and complete for topmost rewrite theories.
- Narrowing can be also used for logical model checking.

Maude-NPA

- Maude-NPA (NRL Protocol Analyzer) is a tool to find or prove the absence of attacks using backwards search in possibly infinite state systems.
- Uses rewriting logic as general theoretical framework:
  - protocols and intruder rules are specified as rewrite rules,
  - crypto properties as oriented equational properties and axioms.
- Uses narrowing modulo equational theories in two ways:
  - as a symbolic reachability analysis method,
  - as an extensible equational unification method.
- Combines with state reduction techniques of NRL Protocol Analyzer: grammars, optimizations, etc.

Maude-NPA

- Maude-NPA supports as equations the algebraic properties of the cryptographic functions:
  - explicit encryption and decryption,
  - exclusive or,
  - modular exponentiation,
  - homomorphism.
- Reasoning modulo such algebraic properties is very important.
- Some protocols that can be proved secure when cryptographic functions are treated as a “black box” can actually be broken by an attacker making clever use of the algebraic properties of cryptographic functions.

Rewriting modulo SMT

- SMT rewrite rules are conditional rewrite rules of the form

$$t(\vec{x}) \to t'(\vec{x}, \vec{y}) \;\; \mathit{if} \;\; \phi$$

  where
  - $\vec{y}$ are SMT variables, and
  - ϕ is an SMT constraint.
- Rewriting is extended to SMT rewriting between constrained terms

$$u \mid \varphi \;\; \rightsquigarrow \;\; v \mid \psi$$

- Applying a rule requires satisfaction of its condition by SMT solving.

Rewriting modulo SMT

- Maude has been interconnected to SMT solvers, such as Z3, CVC4, Yices2.
- Rewriting modulo SMT has been used to develop executable semantics for NASA's PLEXIL language for distributed programming of robot tasks, used in several projects.
- Also to analyze the CASH scheduling algorithm, which attempts to maximize system performance while guaranteeing that critical tasks are executed in a timely manner.
- This algorithm uses an unbounded queue of unused execution budgets, and thus it cannot be analyzed by means of timed-automata formalisms.

Conclusion

Algebraic simulations

- The equational abstraction technique is simple and takes advantage of the expressiveness of rewriting logic and the tools available in the Maude formal environment.
- Other examples, such as the bakery protocol for an arbitrary number of processes and the bounded retransmission protocol, are available in the references.
- These ideas can be generalized to an arbitrary theory interpretation $H : (\Sigma, E) \longrightarrow (\Sigma', E'')$, and to (stuttering) simulations between different sets AP and AP' of state predicates.

Symbolic methods combinations

- Many colleagues are working on the combination of
  - SMT solving,
  - rewriting- and narrowing-based analysis, and
  - (automata-based) model checking;
- and new formal tools for applying these symbolic techniques all together.
- Much work remains to be done.

References

All About Maude – A High-Performance Logical Framework
This monograph gives a comprehensive account of Maude, a language and system based on rewriting logic. 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. Maude is used in many institutions around the world for teaching, research, and formal modeling and analysis of concurrent and distributed systems.
Many examples are used throughout the book to illustrate the main ideas, features, and uses of Maude. The book comes with a CD-ROM containing the complete Maude 2.3 software distribution (including source code), a pdf version of this monograph, and the executable Maude code for all the examples in the book.
In parallel to the printed book, each new volume is published electronically in LNCS Online.
Detailed information on LNCS can be found at
www.springer.com/lncs
Proposals for publication should be sent to
LNCS Editorial, Tiergartenstr. 17, 69121 Heidelberg, Germany
E-mail: lncs@springer.com
ISSN 0302-9743
ISBN 978-3-540-71940-3
Lecture Notes in Computer Science
LNCS
LNAI
LNBI
springer.com
Clavel et al.
Tutorial
LNCS 4350
LNCS
4350
All About Maude –
A High-Performance Logical Framework
with CD-ROM
Manuel Clavel Francisco Durán
Steven Eker Patrick Lincoln
Narciso Martí-Oliet José Meseguer
Carolyn Talcott
All About Maude –
A High-Performance
Logical Framework
How to Specify, Program and Verify
Systems in Rewriting Logic
MAUDE CT
W MAUDE AVE
900
Springer
with CD-ROM

References

- José Meseguer, Miguel Palomino, Narciso Martí-Oliet: Equational abstractions. *Theoretical Computer Science* 403(2-3): 239-264 (2008).
- José Meseguer, Miguel Palomino, Narciso Martí-Oliet: Algebraic simulations. *Journal of Logic and Algebraic Programming* 79(2): 103-143 (2010).
- Francisco Durán, José Meseguer: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming* 81(7-8): 816-850 (2012).

References

- Kyungmin Bae, Santiago Escobar, José Meseguer: Abstract logical model checking of infinite-state systems using narrowing. *Rewriting Techniques and Applications (RTA)*, LIPIcs 21: 81-96 (2013).
- M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott: Maude Manual (Version 2.7.1), 507 pages, July 2016.
- Camilo Rocha, José Meseguer, César A. Muñoz: Rewriting modulo SMT and open system analysis. *J. Log. Algebr. Meth. Program.* 86(1): 269-297 (2017).
- Luis Aguirre, Narciso Martí-Oliet, Miguel Palomino, Isabel Pita: Conditional narrowing modulo SMT and axioms. *Principles and Practice of Declarative Programming (PPDP)*. ACM (2017), to appear.