Equational abstractions in rewriting logic and Maude

Narciso Martí-Oliet
(joint work with J. Meseguer, M. Palomino and F. Durán)

Facultad de Informática
Universidad Complutense de Madrid

narciso@ucm.es

Sinaia, Sept. 2, 2014

Abstract

- 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.
- The most 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

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

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

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.

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

- The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located.
- Constants `left` and `right` represent the two sides of the river.
- Operation `change` 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 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(change(S)) .
  rl [wdog] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

Example: mutual exclusion between two 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
```

Example: readers and writers

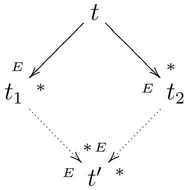
```
mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  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 > .
  rl < s(R), W > => < R, W > .
endm
```

Rewriting and equational simplification

- In a Σ-equation $l = r$ all variables in the righthand side r must appear among the variables of the lefthand side l.
- A term t rewrites to a term t' using such an equation in E if
  1. there is a subterm $t|_p$ of t at a given position p of t such that l matches $t|_p$ via a substitution σ, i.e., $\sigma(l) \equiv t|_p$,and
  2. t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$.
- We denote this step of equational simplification by $t \rightarrow_E t'$.
- 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

- A set of equations E is confluent (or Church-Rosser) when any two rewritings of a term can always be unified 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 $t_2 \to_E^* t'$.



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

Confluence and termination

- 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.
- Therefore, in order to check semantic equality of two terms $t = t'$, it is enough to check that their respective canonical forms are equal, $t{\downarrow_E} = t'{\downarrow_E}$, but, since canonical forms cannot be rewritten anymore, the last equality is just syntactic coincidence: $t{\downarrow_E} \equiv t'{\downarrow_E}$.
- 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.

Matching and simplification modulo

- In the Maude implementation, rewriting modulo A is accomplished by using a matching modulo A algorithm.
- More precisely, given an equational theory A, a term t (corresponding to the lefthand side of an equation) 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

- 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.
- Representation of 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.

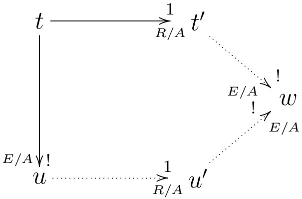
System modules

- 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.

System modules

- 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.



- A simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R.
- In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A.

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

- Main connectives:
  - True: $\top \in \mathrm{LTL}(AP)$.
  - Atomic propositions: If $p \in AP$, then $p \in \mathrm{LTL}(AP)$.
  - Next operator: If $\varphi \in \mathrm{LTL}(AP)$, then $\bigcirc\varphi \in \mathrm{LTL}(AP)$.
  - Until operator: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then $\varphi\, \mathcal{U}\, \psi \in \mathrm{LTL}(AP)$.
  - Boolean connectives: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then the formulas $\neg\varphi$, and $\varphi \vee \psi$ are in LTL(AP).
- Other Boolean connectives:
  - False: $\bot = \neg\top$
  - Conjunction: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - Implication: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.

Linear temporal logic

- Other temporal operators:
  - Eventually: $\Diamond\varphi = \top\,\mathcal{U}\,\varphi$
  - Henceforth: $\Box\varphi = \neg\Diamond\neg\varphi$
  - Release: $\varphi\,\mathcal{R}\,\psi = \neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$
  - Unless: $\varphi\,\mathcal{W}\,\psi = (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$
  - Leads-to: $\varphi \leadsto \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
  - Strong implication: $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
  - Strong equivalence: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$.
- Sometimes it is useful to work with the negation-free fragment of LTL, that we denote LTL^-. Negation is removed, and the duals of the basic operators are added.

Linear temporal logic

- Before considering their formal meaning, let us note that the intuition behind the main temporal connectives is the following:
  - $\top$ 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 $\mathcal{A} = (A, \to_{\mathcal{A}}, L)$ such that
  - A is a set, called the set of states,
  - $\to_{\mathcal{A}}$ is a total binary relation on A, called the transition relation, and
  - $L : A \longrightarrow \mathcal{P}(AP)$ is a function, called the labeling function, associating to each state $a \in A$ the set $L(a)$ of those atomic propositions in AP that hold in the state a.
- A path in a Kripke structure $\mathcal{A}$ is a function $\pi : \mathbb{N} \longrightarrow A$ with $\pi(i) \to_{\mathcal{A}} \pi(i+1)$ for every i.
- We use π^i to refer to the suffix of π starting at $\pi(i)$.

Kripke structures: semantics

- The semantics of the temporal logic LTL is defined by means of a satisfaction relation between a Kripke structure $\mathcal{A}$, a state $a \in A$, and an LTL formula $\varphi \in \mathrm{LTL}(AP)$:

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

- The satisfaction relation $\mathcal{A}, \pi \models \varphi$ is 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 Boolean and temporal operators (e.g., $\bot$, $\wedge$, $\rightarrow$, $\Box$, $\Diamond$, $\mathcal{R}$, and $\leadsto$) 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 some state predicates Π and their semantics in a module, say `M-PREDS`, protecting `M` by means of the operation

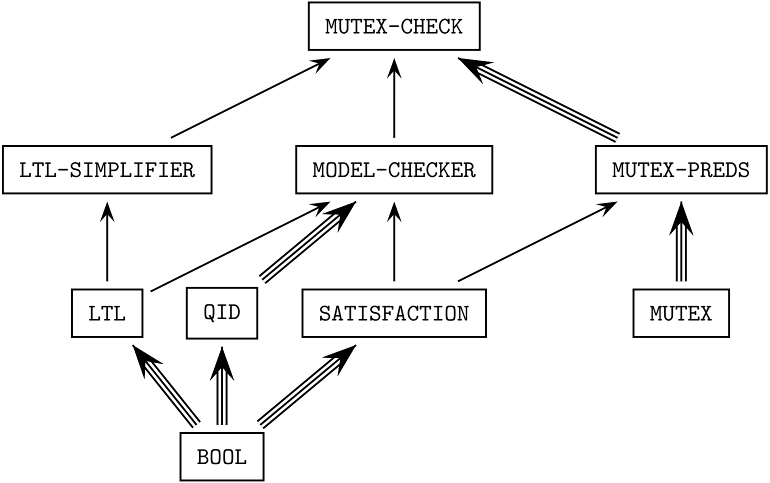
    ```
    op _|=_ : State Prop -> Bool .
    ```

    coming from the predefined `SATISFACTION` module.
- 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 $\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$ becomes decidable.

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: basic 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
```

Model checking a strong liveness property

If a process waits infinitely often, then it is in its critical section infinitely often.

```
Maude> red modelCheck(initial1, ([]<> wait(a)) -> ([]<> crit(a))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial1, ([]<> wait(b)) -> ([]<> crit(b))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial2, ([]<> wait(a)) -> ([]<> crit(a))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial2, ([]<> wait(b)) -> ([]<> crit(b))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true
```

Counterexamples

- A counterexample is a pair consisting of two lists of transitions, where the first corresponds to a finite path beginning in the initial state, and the second describes a loop.
- 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 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(change(S)) .
  rl [wdog] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(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 .
  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 in the other side,
- disaster characterizes the (bad) states in which some eating takes place.

Crossing the river

- The model checker only returns paths that are counterexamples of properties.
- To find a safe path we need to find a formula that somehow expresses the negation of the property we are interested in: 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))
  ```

  Note that this formula is equivalent to the simpler one

  ```
  <> 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})
```

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 (corresponding either to a piece of hardware or software), 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 $\mathcal{R}$.
- Such a quotient is 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 with the help of some Maude tools.

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 $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.
- If the previous inclusion is an equality in all cases, we call H strict.
- $H : \mathcal{A} \longrightarrow \mathcal{B}$ reflects the satisfaction of a formula φ if

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

- Main Theorem. *AP*-simulations reflect satisfaction of $\mathrm{LTL}^{-}(AP)$ formulas. Strict simulations reflect satisfaction of $\mathrm{LTL}(AP)$ formulas.

Minimal systems

- It is often the case that we just 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:
  - $x \rightarrow_{\mathcal{M}^h_{\min}} y \iff \exists a \exists b.(h(a) = x \wedge h(b) = y \wedge a \rightarrow_{\mathcal{M}} b)$
  - $L_{\mathcal{M}^h_{\min}}(a) = \bigcap_{x \in h^{-1}(a)} L_{\mathcal{M}}(x)$.
- Proposition. $h : \mathcal{M} \longrightarrow \mathcal{M}^h_{\min}$ 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 $\mathcal{A}/\!\sim\; = (A/\!\sim, \to_{\mathcal{A}/\sim}, L_{\mathcal{A}/\sim})$, where:
  - $[a_1] \to_{\mathcal{A}/\sim} [a_2] \iff (\exists a_1' \in [a_1])\,(\exists a_2' \in [a_2])\;\; a_1' \to_{\mathcal{A}} a_2'$
  - $L_{\mathcal{A}/\sim}([a]) = \bigcap_{x \in [a]} L_{\mathcal{A}}(x)$.
- Proposition. Given $\mathcal{M}$ and h surjective, the Kripke structures $\mathcal{M}^h_{\mathrm{min}}$ 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 $\mathcal{M}^h_{\mathrm{min}}$ is the most accurate approximation to $\mathcal{M}$ consistent with h.
- It is not always possible to have a computable description of $\mathcal{M}^h_{\mathrm{min}}$.
- The transition relation:

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

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

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

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

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 "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, with atomic propositions $AP_\Pi = \{p(u_1, \ldots, u_n) \text{ ground} \mid p \in \Pi\}$, is

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

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

- 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 $f : k_1 \ldots k_n \longrightarrow k$, 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 readers-and-writers example satisfies these requirements, as well as others we will see later.

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 much easier 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

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

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

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

  then the state predicates Π are preserved by E'.

- We can also use some Maude tools to mechanically discharge these proof obligations.

Equational abstractions: all together

- 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.

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.

Abstraction by adding equations

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

In order to check both 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, since we have no equations with either `true` or `false` in their lefthand side); 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 checker

```
Maude> (check Church-Rosser READERS-WRITERS-PREDS .)
Church-Rosser checking of READERS-WRITERS-PREDS
Checking solution:
  All critical pairs have been joined. The specification is
    locally-confluent.
The specification is sort-decreasing.

Maude> (check Church-Rosser READERS-WRITERS-ABS .)
Church-Rosser checking of READERS-WRITERS-ABS
Checking solution:
  All critical pairs have been joined. The specification is
    locally-confluent.
The specification is sort-decreasing.
```

Readers and writers: sufficient completeness checker

```
Maude> (scc READERS-WRITERS-PREDS .)
Checking sufficient completeness of READERS-WRITERS-PREDS ...
Success: READERS-WRITERS-PREDS is sufficiently complete under the
  assumption that it is weakly-normalizing, confluent, and
  sort-decreasing.

Maude> (scc READERS-WRITERS-ABS .)
Checking sufficient completeness of READERS-WRITERS-ABS ...
Success: READERS-WRITERS-ABS is sufficiently complete under the
  assumption that it is weakly-normalizing, confluent, and
  sort-decreasing.
```

Readers and writers: coherence checker

```
Maude> (check coherence 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: model checking, 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: checking by 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)
```

Concluding remarks

- The 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, such as the bakery protocol for an arbitrary number of processes and the bounded retransmission protocol, are available in the references.
- Related work: Generalization of the equational theory extension $(\Sigma, E) \subseteq (\Sigma, E \cup E')$ to an arbitrary theory interpretation $H : (\Sigma, E) \longrightarrow (\Sigma', E'')$, and to (stuttering) simulations between different sets AP and AP' of state predicates.

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).
- Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn L. Talcott: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. *Lecture Notes in Computer Science* 4350, Springer, 2007.