Foundations of System Development

Martin Wirsing

in cooperation with
Axel Rauschmayer

WS 05/06

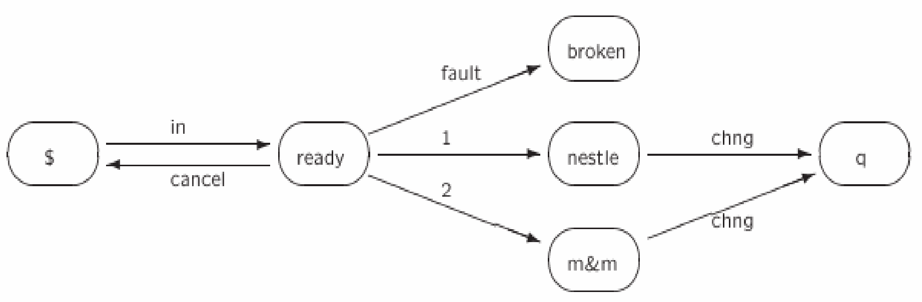
Goals

- Learn how to specify
  state-based concurrent systems in Maude:
  - transition systems
  - (concurrent) object-oriented systems.
- Understand the differences between
  Rewriting Logic and Equational Logic.
- Understand the
  computational and logical interpretations
  of Rewriting Logic.

Automata as Rewrite Systems

Example:

Consider the following possibly faulty candy automaton:



Automata as Rewrite Systems

The candy automaton in Maude:

```
mod CANDY-AUTOMATON is
  sort State .
  ops $ ready broken nestle m&m q : -> State .
  rl [in] : $ => ready .
  rl [cancel] : ready => $ .
  rl [1] : ready => nestle .
  rl [2] : ready => m&m .
  rl [fault] : ready => broken .
  rl [chng] : nestle => q .
  rl [chng] : m&m => q .
endm
```

Rewrite Theories

A rewrite theory $\mathcal{R}$ is a triple $\mathcal{R} = (\Sigma, E, R)$, with:

- (Σ, E) a equational theory, and
- R a set of labeled rewrite rules of the form $l : t \longrightarrow t' \Leftarrow cond$, with l a label, $t, t' \in T_\Sigma(X)_k$ for some kind k, and $cond$ a condition (involving the same variables X) as explained below.

Rewrite Theories

The most general form of a conditional rewrite rule is:

$$l : t \longrightarrow t' \Leftarrow (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w'_k),$$

that is, in general, the condition is a conjunction of equations, memberships, and rewrites, where the variables in all the Σ-terms $t, t', u_i, u'_i, v_j, w_k, w'_k$ are contained in a common set X. There is no requirement that $vars(t) = X$, and no assumptions of confluence or termination. The rule is called unconditional if the condition is empty.

Rewrite Theories in Maude

- Rewrite theories are specified by system modules of the form
  `mod (Σ,E, R) endm`
- With conditional rewrite rules of the form:
  `crl [l] t => t' if cond .`
- A labelled transition system (Z, A, δ) is represented in Maude as follows:
  - The set of states Z is represented by the sort `State` .
  - Any transition `s -a-> s1` is represented by a rewrite rule
    `rl [a] : s => s1 .`

Rewrite Rules as Transitions

Note that rewrite rules do not have an equational interpretation. They are not understood as equations, but as transitions, that in general cannot be reversed.

This is why, in a rewite theory (Σ, E, R) the equations in E are totally different from the rules R, since equations and rules have a totally different semantics.

However, operationally Maude will assume that the equations in E are confluent, terminating, and sort decreasing modulo some $A \subseteq E$, and will compute with such equations and also with the rules in R by rewriting, yet distinguishing equation simplification (the `reduce` command) from rewriting with rules (the `rewrite` command).

The `rewrite` Command

Maude can execute rewrite theories with the **rewrite** command (can be abbreviated to **rew**). For example,

```
Maude> rew $ .
rewrite in CANDY-AUTOMATON : $ .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result State: q
```

- In general:

  `rewrite {[ bound ]} {in module :} term .`

  Causes the specified `term` to be rewritten using the rules, equations, and membership axioms in the given `module`. The default interpreter for rules applies them using a top-down (lazy) strategy and stops when the number of rule applications reaches the given `bound`.

The `search` Command

Of course, since we are in a nondeterministic situation, the **rewrite** command gives us one possible behavior among many.

To systematically explore all behaviors from an initial state we can use the **search** command, which takes two terms: a ground term which is our initial state, and a term, possibly with variables, which describes our desired target state.

Maude then does a breadth first search to try to reach the desired target state. For example, to find the terminating states from the $ state we can give the command (where the "!" in =>! specifies that the target state must be a terminating state),

The search Command

```
Maude> search in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 2)
states: 5  rewrites: 5 in 267757978123ms cpu (0ms
   real) (0 rewrites/second)
X:State --> broken

Solution 2 (state 5)
states: 6  rewrites: 7 in 267757978123ms cpu (9ms
   real) (0 rewrites/second)
X:State --> q

No more solutions.
states: 6  rewrites: 7 in 267757978123ms cpu (13ms
   real) (0 rewrites/second)
```


The search Command

Similarly, we can search for target terms reachable by one rewrite step, one or more, or zero or more steps by typing (respectively):

- `search` t `=>1` t' .
- `search` t `=>+` t' .
- `search` t `=>*` t' .

The `search` Command

```
Maude> search in CANDY-AUTOMATON : $ =>* broken .

Solution 1 (state 2)
states: 3  rewrites: 3 in 267758005139ms cpu (1ms
  real) (0 rewrites/second)
empty substitution

No more solutions.
states: 6  rewrites: 7 in 267758005139ms cpu (2ms
  real) (0 rewrites/second)
```


A Children's problem

"Crossing the river" :

A shepherd needs to transport to the other side of a river a wolf, a goat, and a cabbage:

- The boat has only room for himself and another item.
- In the absence of the shepherd, the wolf would eat the goat.
- In the absence of the shepherd, the goat would eat the cabbage.

Crossing the River in Maude

- Representation of the two sides of the river:

```
sort Side .
ops left right : -> Side .
```

- The shepherd and his belongings are objects with an attribute indicating their location.

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

- The outcome of crossing the river.

```
op change : Side -> Side .
eq change(left) = right .
eq change(right) = left .
```


Crossing the River in Maude

- Equations specify that the wolf eats the goat, and the goat the cabbage, if the shepherd leaves them alone:

```
ceq w(S) g(S) s(S') = w(S) s(S') if S =/= S' .
ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S')
                                      if S =/= S' .
```

- Transitions specify crossing the river as expected.

```
rl [shepherd-alone] : s(S) => s(change(S)) .
rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
rl [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
```

Crossing the River in Maude

- There exists a correct solution:

```
Maude> search in CROSSING-RIVER :
   s(left) w(left) g(left) c(left) =>*
               s(right) w(right) g(right) c(right) .

Solution 1 (state 27)
states: 28  rewrites: 154 in 267758336123ms cpu (27ms real)
   (0 rewrites/second)
empty substitution

No more solutions.
states: 28  rewrites: 170 in 267758336123ms cpu (41ms real)
   (0 rewrites/second)
```


Crossing the River in Maude

- Show the search graph:

```
Maude> show search graph .
state 0, Group: s(left) w(left) g(left) c(left)
   arc 0 ===> state 1 (rl s(S) w(S) => s(change(S))
        w(change(S)) [label wolf] .)
   arc 1 ===> state 2 (rl s(S) g(S) => s(change(S))
        g(change(S)) [label goat] .)
   arc 2 ===> state 3 (rl s(S) c(S) => s(change(S))
        c(change(S)) [label cabbage] .)
   arc 3 ===> state 4 (rl s(S) => s(change(S)) [label
        shepherd-alone] .)
. . .

state 27, Group: s(right) w(right) g(right) c(right)
. . .
```

Crossing the River in Maude

```
Maude> show path 27 . ***Show shortest path to state 27
state 0, Group: s(left) w(left) g(left) c(left)
===[ rl s(S) g(S) => s(change(S)) g(change(S)) [label goat] .
   ]===>
state 2, Group: s(right) w(left) g(right) c(left)
===[ rl s(S) => s(change(S)) [label shepherd-alone] . ]===>
state 7, Group: s(left) w(left) g(right) c(left)
===[ rl s(S) w(S) => s(change(S)) w(change(S)) [label wolf] .
   ]===>
state 13, Group: s(right) w(right) g(right) c(left)
===[ rl s(S) g(S) => s(change(S)) g(change(S)) [label goat] .
   ]===>
state 20, Group: s(left) w(right) g(left) c(left)
===[ rl s(S) c(S) => s(change(S)) c(change(S)) [label cabbage] .
   ]===>
state 25, Group: s(right) w(right) g(left) c(right)
===[ rl s(S) => s(change(S)) [label shepherd-alone] . ]===>
state 26, Group: s(left) w(right) g(left) c(right)
===[ rl s(S) g(S) => s(change(S)) g(change(S)) [label goat] .
   ]===>
state 27, Group: s(right) w(right) g(right) c(right)
```


(Concurrent) Object-based Systems

One of the most useful and important classes of concurrent systems is that of concurrent object systems, made out of concurrent objects, which encapsulate their own local state and can interact with other objects in a variety of ways, including both synchronous interaction, and asynchronous communication by message passing.

Concurrent Object Systems in Maude

It is of course possible to represent a concurrent object system as a rewrite theory with somewhat different modeling styles and adopting different notational conventions.

What follows is a particular style of representation that has proved useful and expressive in practice, and that is supported by Full Maude's object-oriented modules.

It is also possible to define object-oriented modules in Core Maude using the `conf` attribute to specify an associative commutative multiset union operators as a constructor of configurations of objects and messages; the **`frewrite`** command then ensures object and message fair executions (see the Maude 2.0 manual).

Concurrent Object Systems in Maude

To model a concurrent object system as a rewrite theory, we have to explain two things:

- how the distributed states of such a system are equationally axiomatized and modeled by the initial algebra of an equational theory (Σ, E), and
- how the concurrent interactions between objects are axiomatized by rewrite rules.

Concurrent Object Systems in Maude

Distributed Object States

- The concurrent state of an object-oriented system, often called a configuration,
  has typically the structure of a multiset made up, of objects and messages.

```
    sorts Conf Object Msg .
    subsort Object Msg < Conf .

*** multiset union
    op _ _ : Conf Conf -> Conf  [assoc comm id: null] .
```

Objects

An object in a given state is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's attribute identifiers, and the v_i's are the corresponding values.

The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator _ , _ which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial.

Configurations

The value of each attribute shouldn't be arbitrary: it should have an appropriate sort, dictated by the nature of the attribute. Therefore, in Full Maude object classes can be declared in class declarations of the form,

$$\texttt{class}\ C \mid a_1 : s_1, \ldots, a_n : s_n\ .$$

where C is the class name, and s_i is the sort required for attribute a_i.

In Core Maude classes are formalized similarly as in FOOSE:

```
op C :  -> Cid .
op a1 :_ : s1 -> Attribute .
. . .
```

Example: Simple Asynchronous Communication

- Consider a system consisting of 3 objects:
  a buffer, a sender and a receiver.
- A buffer stores a list of integers in its q attribute. Lists of integers are built using an associative list concatenation operator, _ . _ with identity nil, and integers are regarded as lists of length one. The name of the object reading from the buffer is stored in its reader attribute; such names belong to a sort Oid of object identifiers. Therefore, the class declaration for buffers is,

```
class Buffer | q : IntList, reader: Oid .
```


Example: Simple Asynchronous Communication

The sender and receiver objects store an integer in a cell attribute that can also be empty (mt) and have also a counter (cnt) attribute. The sender stores also the name of the receiver in an additional attribute.

```
class Sender | cell: Int?, cnt: Int, receiver: Oid .
class Receiver | cell: Int?, cnt: Int .
```

where Int? is a supersort of Int having a new constant mt.

Messages

The messages sent by a sender object have the form,

```
(to Z : E from (Y,N))
```

where Z is the name of the receiver, E is the number sent, Y is the name of the sender, and N is the value of its counter at the time of the sending.

The syntax of messages is user-definable; it can be declared in Full Maude by message operator declarations. In our example by:

```
msg (to _ : _ from (_,_)) : Oid Int Oid Int -> Msg .
```


"Soup of Objects and Messages"

The associativity and commutativity of a configuration's multiset structure make it very fluid. We can think of it as "soup" in which objects and messages float, so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind.

In general, the rewrite rules in R describing the dynamics of an object-oriented system can have the form,

Object Rewrite Rules

$$
\begin{aligned}
r: \quad & M_1 \ldots M_n \ \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\
& \qquad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
& \qquad M'_1 \ldots M'_q \\
& \qquad \mathit{if}\ C
\end{aligned}
$$

where r is the label, the Ms are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition.

Object Rewrite Rules

That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created.

If two or more objects appear in the lefthand side, we call the rule synchronous, because it forces those objects to jointly participate in the transition. If there is only one object and at most one message in the lefthand side, we call the rule asynchronous.

Example: Simple Asynchronous Communication

```
rl [read] : < X : Buffer | q: L . E, reader: Y >
            < Y : Sender | cell: mt, cnt: N >
         => < X : Buffer | q: L, reader: Y >
            < Y : Sender | cell: E, cnt: N + 1 >

rl [send] : < Y : Sender | cell: E, cnt: N, receiver: Z >
   => < Y : Sender | cell: mt, cnt: N > (to Z : E from (Y,N))

rl [receive] : < Z : Receiver | cell: mt, cnt: N >
               (to Z : E from (Y,N))
             => < Z : Receiver | cell: E, cnt: N + 1 >
```

where `E` and `N` range over `Int`, `L` over `IntList`, `X, Y, Z` over `Oid`, and `L.E` is a list with last element `E`.

Example: Simple Asynchronous Communication

Notice that the `read` rule is synchronous and the `send` and `receive` rules asynchronous.

Of course, these rules are applied modulo the associativity and commutativity of the multiset union operator, and therefore allow both object synchronization and message sending and receiving events anywhere in the configuration, regardless of the position of the objects and messages.

We can then consider the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizing the object system with these three object classes, with R the three rules above (and perhaps other rules, such as one for the receiver to write its contents into another buffer object, that are omitted).

Rewriting Logic in General

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, the sentences that it proves are universally quantified rewrites of the form, $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_{\Sigma,E}(X)_k$, for some kind k, which are obtained by finite application of the following rules of deduction:

- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$

Rewriting Logic in General

- **Equality**.
$$\frac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X) u = u' \quad E \vdash (\forall X) v = v'}{(\forall X)\ u' \longrightarrow v'}$$

- **Congruence**. For each $f : k_1 \dots k_n \longrightarrow k$ in Σ, with $\{1, \dots, n\} - \phi(f) = \{j_1, \dots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,
$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \dots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \dots, t_{j_1}, \dots, t_{j_m}, \dots, t_n) \longrightarrow f(t_1, \dots, t'_{j_1}, \dots, t'_{j_m}, \dots, t_n)}$$

- **Transitivity**
$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

Rewriting Logic in General

- **Replacement**. For each finite substitution $\theta : X \longrightarrow T_\Sigma(Y)$, with, say, $X = \{x_1, \ldots, x_n\}$, and $\theta(x_l) = p_l$, $1 \leq l \leq n$, and for each rule in R of the form,

$$l : (\forall X)\ t \longrightarrow t' \ \Leftarrow\ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w'_k)$$

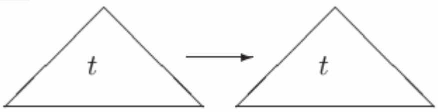
with $Z = \{x_{j_1}, \ldots, x_{j_m}\}$, the set of unfrozen variables in t and t', then,

$$\frac{(\bigwedge_r (\forall Y)\ p_{j_r} \longrightarrow p'_{j_r}) \qquad (\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\ \theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\ \theta(w_k) \longrightarrow \theta(w'_k))}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$

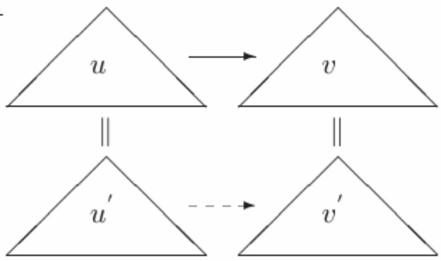
where for $x \in X - Z$, $\theta'(x) = \theta(x)$, and for $x_{j_r} \in Z$, $\theta'(x_{j_r}) = p'_{j_r}$, $1 \leq r \leq m$.

Rewriting Logic in General

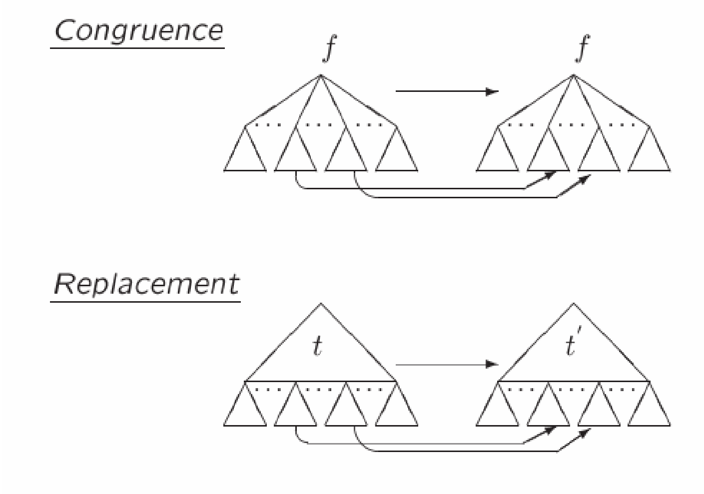
Reflexivity



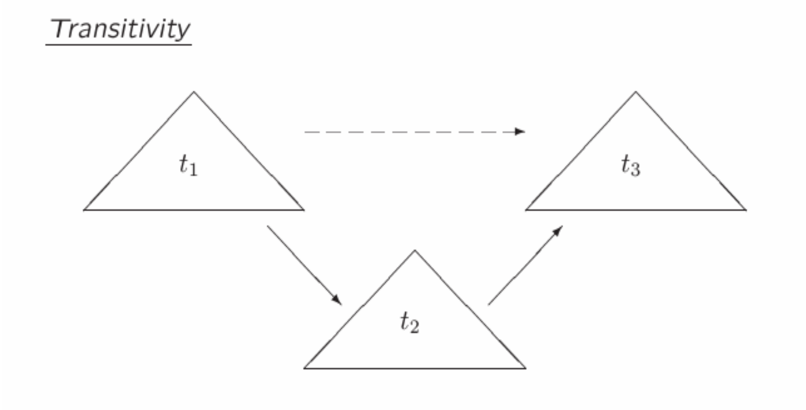
Equality



Rewriting Logic in General



Rewriting Logic in General



Computational Meaning of Rewrite Rules

Rewriting logic is a computational logic to specify concurrent systems. Its inference system allows us to infer all the possible finitary concurrent computations of a system specified as a rewrite theory $\mathcal{R}$ as follows:

- **Reflexivity** is just the possibility of having idle transitions
- **Equality** means that states are equal modulo E
- **Congruence** is a general form of sideways parallelism
- **Replacement** combines an atomic transition at the top using a rule with nested concurrency in the substitution
- **Transitivity** is sequential composition.

Computational and Logical Readings

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has two closely related, yet different, readings, one computational, and another logical.

Computationally, a rewrite theory specifies a concurrent system, whose set of states is (a kind in) the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies a parameterized family of concurrent transitions in the system.

Logically, a rewrite theory specifies a logic, whose set of formulas is (a kind in) the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies an inference rule in the logic.

Example: Implicational Logic in Maude

```
mod MINIMALR is sorts SentConstant Formula Configuration .
subsorts SentConstant < Formula < Configuration .
op _→_ : Formula Formula -> Formula .
op empty : -> Configuration .
op __ : Configuration Configuration -> Configuration [assoc comm id: empty] .
vars A B C : Formula .
rl [ax.K] :        empty
           => ----------------
                A → (B → A) .
rl [ax.S] :                        empty
           => --------------------------------------------
              (A → B) → ((A →(B → C)) → (A → C)) .
rl [mp] :     (A → B)    A
           => ------------------
                     B .
endm
```


Computational and Logical Readings

The point is that we have the following equivalences between these two readings:

State	$\longleftrightarrow$	*Term*	$\longleftrightarrow$	*Formula*
Computation	$\longleftrightarrow$	*Rewriting*	$\longleftrightarrow$	*Proof*
Distributed Structure	$\longleftrightarrow$	*Algebraic Structure*	$\longleftrightarrow$	*Logical Structure*

Summary

- Rewriting Logic provides a mathematical basis for modelling concurrent distributed systems including object-oriented systems.
- Dynamic behaviour and, in particular, concurrent transitions are defined by rewrite rules.
- Distributed configurations are represented by terms

  where the distribution structure is algebraically defined by initial algebras of equational theories.