ISR: Lecture 8

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Concurrent Objects in Rewriting Logic

Rewriting logic can model very naturally many different kinds of concurrent systems. We have, for example, seen that Petri nets can be naturally formalized as rewrite theories. The same is true for many other models of concurrency such as CCS, the π-calculus, dataflow, real-time models, and so on.

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 Objects in Rewriting Logic (II)

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 All About Maude book).

Concurrent Objects in Rewriting Logic (III)

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.

We first explain how the distributed states are equationally axiomatized.

Configurations

Let us consider the key state-building operations in Σ and the equations E axiomatizing the distributed states of concurrent object systems. 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.

Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax (i.e. juxtaposition) as,

$$\_\ \_ : \mathbf{Conf} \times \mathbf{Conf} \longrightarrow \mathbf{Conf}.$$

Configurations (II)

The operator _ _ is declared to satisfy the structural laws of associativity and commutativity and to have identity `null`. Objects and messages are singleton multiset configurations, and belong to subsorts

$$\mathbf{Object}\ \mathbf{Msg} < \mathbf{Conf},$$

so that more complex configurations are generated out of them by multiset union.

Configurations (III)

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 (IV)

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.

We can illustrate such class declarations by considering three classes of objects, `Buffer`, `Sender`, and `Receiver`.

Configurations (IV)

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 .
```

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.

Configurations (V)

The counter attribute is used to ensure that messages are received by the receiver in the same order as they are sent by the sender, even though communication between the two parties is asynchronous.

Each time the sender gets a new value from the buffer, it increments its counter. It later uses the current value of the counter to tag the message sent with that value to the receiver.

The receiver only accepts a message whose tag is its current counter. It then increments its counter indicating that it is ready for the next message.

Configurations (VI)

The class declarations are:

```
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`.

In Full Maude one can also give subclass declarations, with `subclass` syntax (similar to that of `subsort`) so that all the attributes and rewrite rules of a superclass are inherited by a subclass, which can have additional attributes and rules of its own.

Configurations (VII)

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 .
```

Object Rewrite Rules

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 (II)

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

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 (III)

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.

Object Rewrite Rules (IV)

Three typical rewrite rules involving objects in the `Buffer`, `Sender`, and `Receiver` classes are,

```
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`.

Object Rewrite Rules (V)

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).

Rewrite Theories in General

It is clear that rewriting logic has the underlying equational logic as a parameter: the more general the equational logic, the more general the resulting rewite theories. For example, we have seen that for order-sorted equational logic rules can have the general form,

$$l : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_{i} u_i = u'_i) \wedge (\bigwedge_{j} w_j \longrightarrow w'_j).$$

It has also become increasingly clear that frozen operators, that restrict the rewrites allowed below them, are also very useful in practice.

Rewrite Theories in General (II)

We can illustrate frozen operators with the following nondeterministic choice example (in Maude syntax):

```
mod CHOICE is
  protecting INT .
  sorts Elt MSet .
  subsorts Elt < MSet .
  ops a b c d e f g : -> Elt .
  op __ : MSet MSet -> MSet [assoc comm] .
  op card : MSet -> Int [frozen] .
  eq card(X:Elt) = 1 .
  eq card(X:Elt M:MSet) = 1 + card(M:MSet) .
  rl [choice] : X:MSet Y:MSet => Y:MSet .
endm
```

Rewrite Theories in General (III)

It does not make much sense to rewrite below the cardinality function `card`, because then the multiset whose cardinality we wish to determine becomes a moving target.

If `card` had not been declared `frozen`, then the rewrites, `a b c` $\longrightarrow$ `b c` $\longrightarrow$ `c` would induce rewrites, $3 \longrightarrow 2 \longrightarrow 1$, which seems bizarre.

The point is that we think of the kind `[MSet]` as the state kind in this example, whereas `[Int]` is the data kind. By declaring `card` frozen, we restrict rewrites to the state kind, where they belong.

Rewrite Theories in General (IV)

This leads to the following general definition of a rewrite theory on order-sorted equational logic:

A rewrite theory is a 4-tuple, $\mathcal{R} = (\Sigma, E, \phi, R)$, where:

- (Σ, E) is a kind-complete order-sorted equational theory, with, say, kinds K, sorts S, and operations Σ
- $\phi : \Sigma \longrightarrow \mathcal{P}_{fin}(\mathbb{N})$ is a $K^* \times K$-indexed family of functions assigning to each $f : k_1 \ldots k_n \longrightarrow k$ in Σ the finite set $\phi(f) \subseteq \{1, \ldots, n\}$ of its frozen argument positions
- R is a set of (universally quantified) labeled conditional rewrite rules of the form (with t, t' and the w_k, w'_k pairs of terms of same kind)

$$l : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j).$$

Rewrite Theories in General (V)

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, and given a Σ-term $t \in T_\Sigma(X)$, we call a variable $x \in vars(t)$ frozen in t iff there is a nonvariable position $\alpha \in \mathbb{N}^*$ such that
$t/\alpha = f(u_1, \ldots, u_i, \ldots, u_n)$, with $i \in \phi(f)$, and $x \in vars(u_i)$.
Otherwise, we call $x \in X$ unfrozen.

Similarly, given Σ-terms $t, t' \in T_\Sigma(X)$, we call a variable $x \in X$ unfrozen in t and t' iff it is unfrozen in both t and t'.

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 $t \in T_{\Sigma}(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$

- **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 $f : k_1 \ldots k_n \longrightarrow k$ in Σ, with $\{1, \ldots, n\} - \phi(f) = \{j_1, \ldots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,

$$\frac{(\forall X)\, t_{j_1} \longrightarrow t'_{j_1} \quad \ldots \quad (\forall X)\, t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\, f(t_1, \ldots, t_{j_1}, \ldots, t_{j_m}, \ldots, t_n) \longrightarrow f(t_1, \ldots, t'_{j_1}, \ldots, t'_{j_m}, \ldots, t_n)}$$

- **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,

$$l : (\forall X)\, t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j)$$

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

$$(\bigwedge_r (\forall Y)\, p_{j_r} \longrightarrow p'_{j_r})$$

$$\frac{(\bigwedge_i (\forall Y)\, \theta(u_i) = \theta(u'_i)) \quad \wedge \quad (\bigwedge_j (\forall Y)\, \theta(w_j) \longrightarrow \theta(w'_j))}{(\forall Y)\, \theta(t) \longrightarrow \theta'(t')}$$

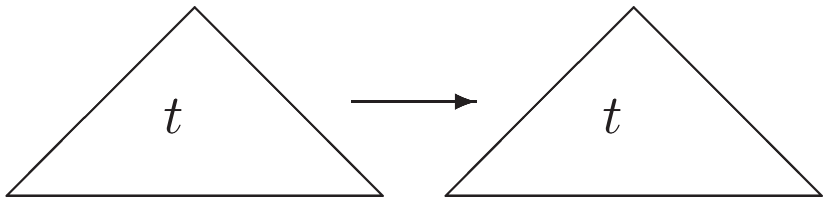
where for $x \in X - Z$, $\theta'(x) = \theta(x)$, and for $x_{j_r} \in Z$, $\theta'(x_{j_r}) = p'_{j_r}$, $1 \leq r \leq m$.

- **Transitivity**

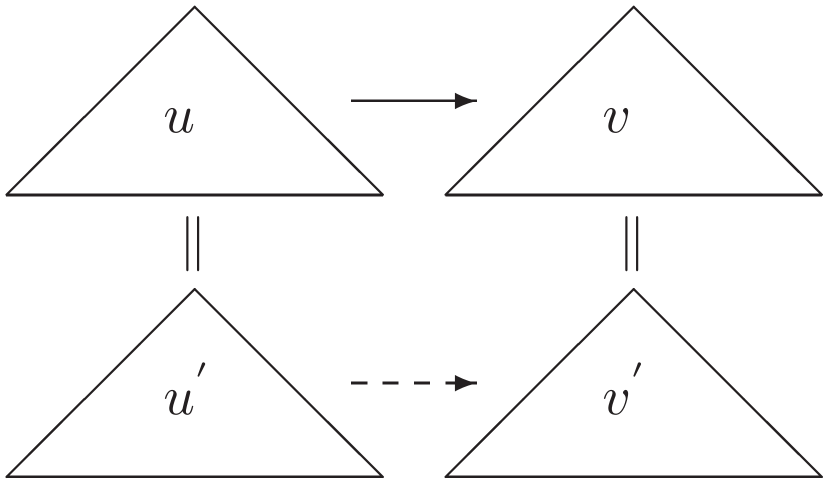
$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

Rewriting Logic in Pictures

Reflexivity

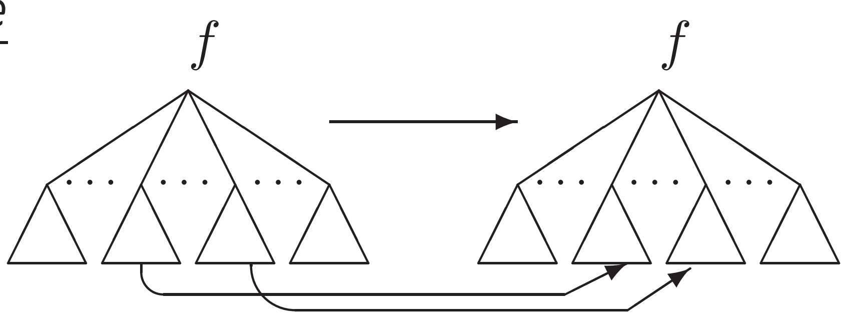


Equality

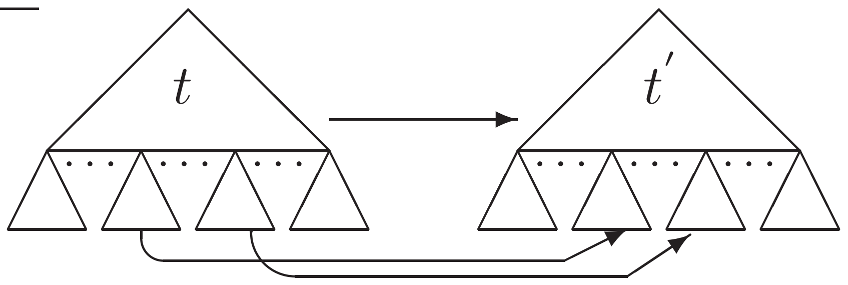


Rewriting Logic in Pictures (II)

Congruence

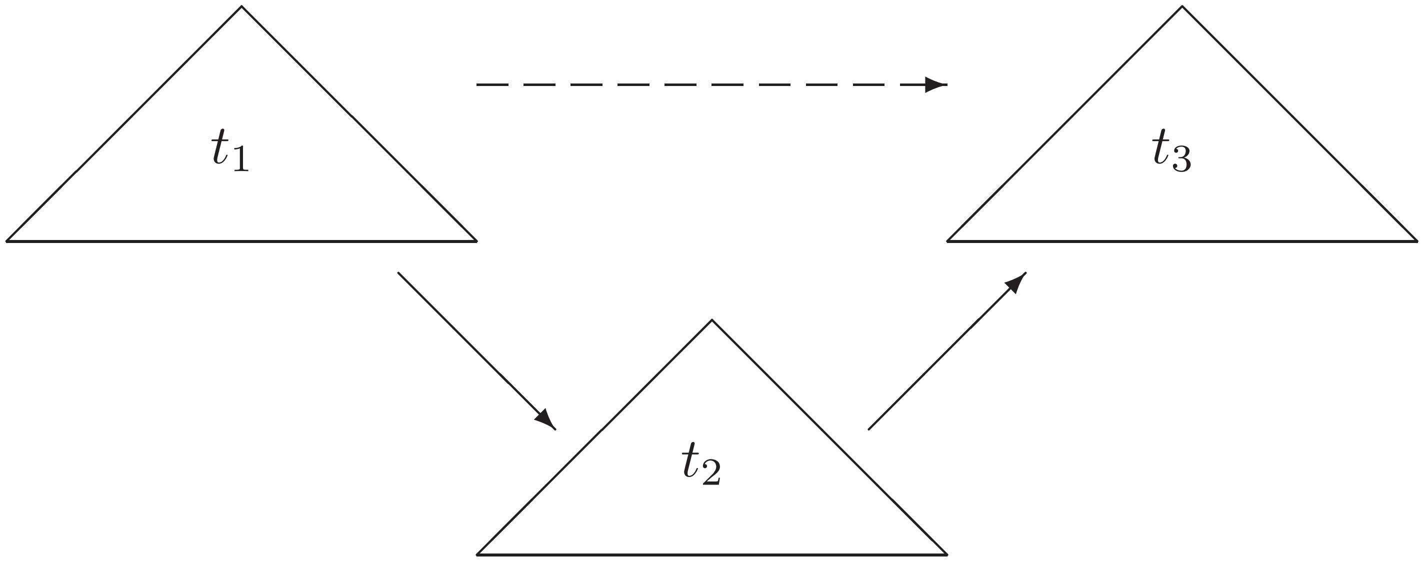


Replacement



Rewriting Logic in Pictures (III)

Transitivity



Computational Meaning of the Inference 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 of concurrent transitions.

Logical and Computational 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.

Logical and Computational Readings (II)

For example, the logic of implication is:

```
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
```

Logical and Computational Readings (III)

The computational and logical readings are not mutually exclusive. Rather, the same theory can be regarded computationally, or logically, or both!, depending on one's point of view.

For example, logically, our `PETRI-MACHINE` example is a linear logic theory in disguise. It is just a matter of a slight change of syntax, replacing the empty syntax, $\_\ \_$ by that of linear logic's multiplicative conjunction operator $\_ \otimes \_$.

Logical and Computational Readings (IV)

The operator $\_ \otimes \_$ can be viewed as a form or resource-conscious non-idempotent conjunction. Then, the state $a \otimes q \otimes q$ corresponds to having an apple *and* a quarter *and* a quarter, which is a strictly better situation than having an apple *and* a quarter (non-idempotence of $\otimes$).

Then, in order to get the tensor theory corresponding to `PETRI-MACHINE`, it is enough to change the arrows into turnstiles, getting the following axioms:

$$\begin{array}{ll} \textit{buy-c}: & \$ \vdash c \\ \textit{buy-a}: & \$ \vdash a \otimes q \\ \textit{change}: & q \otimes q \otimes q \otimes q \vdash \$ \end{array}$$

Logical and Computational Readings (V)

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*

In particular, concurrent computations in our Petri net example coincide with linear logic proofs.

Exercises

Ex.8.1. Show in detail that, given a Petri net N, the inference system given in pages 39–40 of Lecture 7 to generate all its concurrent computations is equivalent to the specialization to the theory $R(N)$ of the general inference system for rewriting logic.