Program Verification: Lecture 15

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)

$$
\begin{array}{ll}
r: & M_1 \dots M_n \; \langle O_1 : F_1 \mid atts_1 \rangle \dots \langle O_m : F_m \mid atts_m \rangle \\[1em]
 & \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \dots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\[1em]
 & \qquad \langle Q_1 : D_1 \mid atts''_1 \rangle \dots \langle Q_p : D_p \mid atts''_p \rangle \\[1em]
 & \qquad M'_1 \dots M'_q \\[1em]
 & \qquad \mathit{if}\ C
\end{array}
$$

where r is the label, the Ms are message expressions, $i_1, \dots, i_k$ are different numbers among the original $1, \dots, m$, 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).

Σ-Transition Systems

What are the models of rewrite theories? The simplest models are the Σ-transition systems.[a]

For $\Sigma = ((S,<),F,\Sigma)$ a kind-complete order-sorted signature, a Σ-transition system is a pair $\mathcal{A} = (\mathbb{A}, \rightarrow_{\mathcal{A}})$ where:

- $\mathbb{A} = (A, \_{}_{\mathbb{A}})$ is a Σ-algebra, and
- $\rightarrow_{\mathcal{A}} = \{\rightarrow_{\mathcal{A},[s]}\}_{[s] \in S/(\leq \cup \geq)^{+}}$ a $S/(\leq \cup \geq)^{+}$-indexed family of transition relations, with $\rightarrow_{\mathcal{A},[s]} \subseteq A_{[s]} \times A_{[s]}$ for each $[s]$; and such that, if $f : [s_1] \ldots [s_n] \rightarrow [s]$ in F, $a_j \in A_{[s_j]}$, $1 \leq j \leq n$, and $a_i \rightarrow_{\mathcal{A},[s_i]} a_i'$, then (*Transition Σ-closure*),

$$f_{\mathbb{A}}(a_1, \ldots, a_i, \ldots, a_n) \rightarrow_{\mathcal{A},[s]} f_{\mathbb{A}}(a_1, \ldots, a_i', \ldots, a_n).$$

[a] For more general true concurrency models see: J. Meseguer, "20 Years of Rewriting Logic," *JLAMP*, 81: 721–781 (2012).

Satisfaction

By definition, a Σ-transition system $\mathcal{A} = (\mathbb{A}, \to_{\mathcal{A}})$ satisfies (or is a model of) a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, written $\mathcal{A} \models \mathcal{R}$, iff it satisfies each equation $(u = v) \in E$, written $\mathcal{A} \models u = v$, and each rule[a] $(l : t \to t') \in R$, written $\mathcal{A} \models l : t \to t'$. This exactly means:

- $\mathbb{A} \models E$, and
- for each rewrite rule $(l : t \to t') \in R$ and each assignment $a \in [X \to A]$, with t, t' of kind $[s]$, there is a transition:

$$ta \to_{\mathcal{A},[s]} t'a.$$

[a] To simplify the exposition we assume all rules in R are unconditional. All generalizes smoothly to the case of conditional rules.

Soundness and Completeness of Rewriting Logic

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ proves $t \rightarrow^* t'$, with t, t' Σ-terms of same kind $[s]$, written $\mathcal{R} \vdash t \rightarrow^* t'$, iff $t \rightarrow^*_{R/E} t$.

Theorem (Soundness). For each rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and Σ-transition system $\mathcal{A} = (\mathbb{A}, \rightarrow_{\mathcal{A}})$ such that $\mathcal{A} \models \mathcal{R}$ we have:

$$\mathcal{R} \vdash t \rightarrow^* t' \quad \Rightarrow \quad \mathcal{A} \models t \rightarrow^* t'.$$

Rewriting logic is also complete (Bruni and Meseguer, Theoretical Computer Science, 360, 386-414, 2006), that is, we have:

$$\mathcal{R} \models t \rightarrow^* t' \quad \Rightarrow \quad \mathcal{R} \vdash t \rightarrow^* t'.$$

where, by definition, $\mathcal{R} \models t \rightarrow^* t'$ iff for all $\mathcal{A}$ such that $\mathcal{A} \models \mathcal{R}$, $\mathcal{A} \models t \rightarrow^* t'$ holds (i.e., $\forall a \in [X \to A]$, $ta \rightarrow^*_{\mathcal{A},[s]} t'a$).

Simulations Maps of Σ-Transition Systems

Given two Σ-transition systems $\mathcal{A} = (\mathbb{A}, \rightarrow_{\mathcal{A}})$, and $\mathcal{B} = (\mathbb{B}, \rightarrow_{\mathcal{B}})$, a Σ-simulation map $h : \mathcal{A} \rightarrow \mathcal{B}$ is a Σ-homomorphism $h : \mathbb{A} \rightarrow \mathbb{B}$ such that it "preserves transitions," that is, for each $[s] \in S/(\leq \cup \geq)^{+}$, if we have a transition $a \rightarrow_{\mathcal{A},[s]} a'$ in $\mathcal{A}$, then there is a corresponding transition $h_{[s]}(a) \rightarrow_{\mathcal{B},k} h_{[s]}(a')$ in $\mathcal{B}$.

Intuitively, we can think of h as an algebraic (because it preserves the algebraic operations) simulation of system $\mathcal{A}$ by system $\mathcal{B}$, because, via h, $\mathcal{B}$ can mimic or simulate any transition that $\mathcal{A}$ can make.

The Initial Σ-Transition System $\mathcal{T}_{\mathcal{R}}$

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, consider the Σ-transition system $\mathcal{T}_{\mathcal{R}} = (\mathbb{T}_{\Sigma/E}, \rightarrow_{\mathcal{R}})$, where, by definition,

$$[t] \rightarrow_{\mathcal{R}} [t'] \quad \Leftrightarrow \quad \mathcal{R} \vdash t \rightarrow t' \;\; (i.e.,\; t \rightarrow_{R/E} t')$$

This is indeed a Σ-transition system, and $\mathcal{T}_{\mathcal{R}} \models \mathcal{R}$, because:

- $\mathbb{T}_{\Sigma/E} \models E$, and
- for each $(l : u \rightarrow v) \in R$ we trivially have $ua \rightarrow_{\mathcal{R}} va$ for any $a \in [X \rightarrow T_{\Sigma/E}]$ because $u\theta \rightarrow_{R/E} v\theta$ for any θ.

Using the Soundness Theorem and the initiality theorem for order-sorted equational logic it is relatively easy to prove (exercise) that we have:

The Initial Σ-Transition System $\mathcal{T}_{\mathcal{R}}$ (II)

Theorem. (Initiality Theorem). For Σ sensible and kind-complete, $\mathcal{T}_{\mathcal{R}}$ is initial in the class of all Σ-transition systems that satisfy $\mathcal{R}$. That is, if $\mathcal{A} \models \mathcal{R}$, then there is a unique Σ-simulation map:

$$\_{}^{\mathcal{R}}_{\mathcal{A}} : \mathcal{T}_{\mathcal{R}} \to \mathcal{A}.$$

In fact, on the underlying algebras, $\_{}^{\mathcal{R}}_{\mathcal{A}}$ is just the unique Σ-homomorphism: $\_{}^{E}_{\mathbb{A}} : \mathbb{T}_{\Sigma/E} \to \mathbb{A}$.

When reasoning about a concurrent system specified by a rewrite theory $\mathcal{R}$, for example as a system module in Maude, $\mathcal{T}_{\mathcal{R}}$ gives us the standard model specified by $\mathcal{R}$. In other words, the initial algebra semantics of equational logic generalizes in a natural way to an initial Σ-transition system semantics for rewriting logic.

Executability of Rewrite Theories: Coherence

When is a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ executable? $(\Sigma, E \cup B)$ should be sort-decreasing, terminating, confluent, and sufficiently complete modulo B. But this is not enough, because $\rightarrow_{R/E \cup B}$ can be undecidable. We can reduce it to $\rightarrow_{R/B}$ by requiring coherence of R w.r.t. E modulo B:

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

Maude's Coherence Checker checks this property.

The Canonical Σ-Transition System $\mathbb{C}_{\mathcal{R}}$

Given a system module `mod` $\mathcal{R}$ `endm`, with, say, $\mathcal{R} = (\Sigma, E \cup B, R)$, Maude assumes the following executability conditions: (i) the usual ones for $(\Sigma, E \cup B)$, and (ii) the (ground) coherence of R with respect to E modulo B.

Assuming (i)–(ii), we can define the canonical Σ-transition system $\mathcal{C}_{\mathcal{R}} = (\mathbb{C}_{\Sigma/E,B}, \rightarrow_{\mathcal{C}_{\mathcal{R}}})$, were $\mathbb{C}_{\Sigma/E,B}$ is the canonical term algebra modulo B, and given $[u], [v] \in C_{\Sigma/E,B,[s]}$, $[u] \rightarrow_{\mathcal{C}_{\mathcal{R}}} [v]$ holds iff there exists v' such that $u \rightarrow_{R/B} v'$ and $[v] = [v'!_{E/B}]$. I.e., states are elements of $\mathbb{C}_{\Sigma/E,B}$; transitions from $[u]$ are the normal forms $[v'!_{E/B}]$ of rewrites $u \rightarrow_{R/B} v'$.

Theorem. $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{C}_{\mathcal{R}}$ are isomorphic Σ-transition systems, and therefore both are initial among all $\mathcal{A}$ such that $\mathcal{A} \models \mathcal{R}$.