ISR: Lecture 7

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

A (Meta-)Logic for Concurrent Systems and Logics

We have seen that confluence abstractly captures the notion of a computational system being deterministic. This means that equational theories $(\Sigma, E \cup B)$ with E confluent modulo B, as in Maude's functional modules, provides an attractive declarative programming language to specify and program deterministic systems.

Is there a similar (meta-)logic to specify and program non-deterministic systems such as: (i) concurrent systems and (ii) logics? I will show in these lectures that rewriting logic is computational logic that generalizes equational logic in a natural way and is very well suited for this purpose.

Rewrite Theories: Preliminary Definition

We give a first, already quite general, definition of rewrite theories. We will further generalize this notion later.

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

- (Σ, E) a (kind-complete) order-sorted equational theory, and
- R a set of labeled rewrite rules of the form $l : t \longrightarrow t' \quad \Leftarrow cond$, 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.

Conditional Rewrite Rules

The most general form of a conditional rewrite rule is:

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

that is, in general, the condition is a conjunction of equations and rewrites, where the variables in all the Σ-terms $t, t', u_i, u'_i, w_j, w'_j$ 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.

Maude System Modules

In Maude, rewrite theories are specified in system modules.

The same way that a functional module has essentially the form, `fmod` (Σ, E) `endfm`, with (Σ, E) an order-sorted equational logic theory, a system module has essentially the form, `mod` (Σ, E, R) `endm`, with (Σ, E, R) a rewrite theory.

We will illustrate the syntax details in examples. In particular, a conditional rewrite rule of the form, $l : t \longrightarrow t' \quad \Leftarrow cond$ is specified in Maude with syntax,

$$\texttt{crl [}l\texttt{] : } t \texttt{ => } t' \texttt{ if } cond \texttt{ .}$$

and an unconditional rule $l : t \longrightarrow t'$ with syntax,

$$\texttt{rl [}l\texttt{] : } t \texttt{ => } t' \texttt{ .}$$

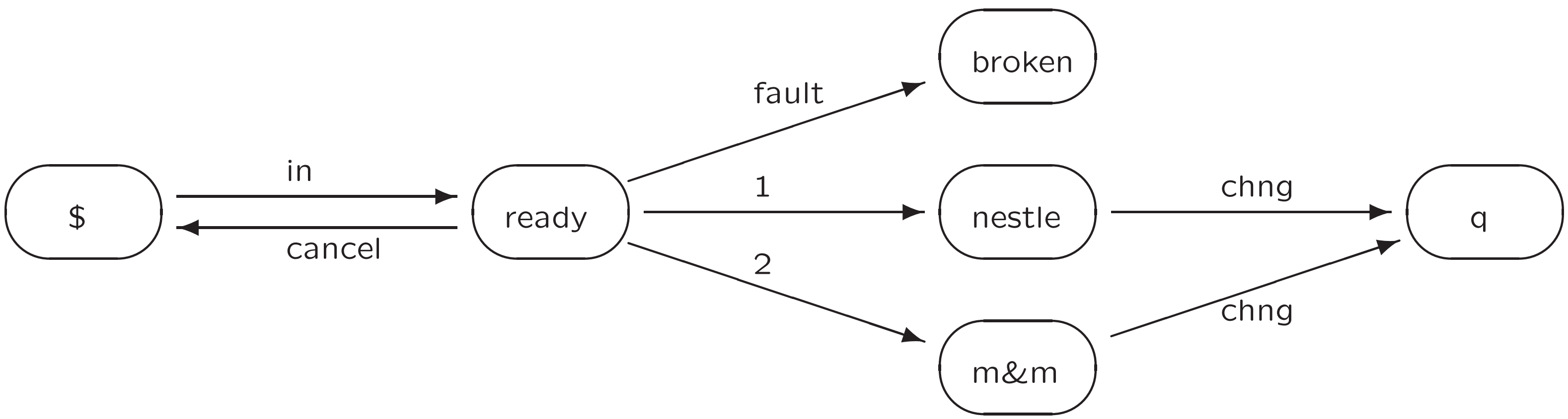
Some Rewriting Logic Examples

To motivate rewriting logic as a formalism to mathematically model and program concurrent systems and logics, we will show how it can be used to naturally specify three important classes of systems, namely:

- automata, also called labeled transition systems,
- Petri nets, one of the simplest concurrency models, and
- object-oriented concurrent systems.

Concurrency vs. Nondeterminism: Automata

We can motivate concurrency by its absence. The point is that we can have systems that are nondeterministic, but are not concurrent. Consider the following faulty automaton to buy candy:



Concurrency vs. Nondeterminism: Automata (II)

Although in the above automaton each labeled transition from each state leads to a single next state, the automaton is nondeterministic in the sense that the automaton's computations are not confluent, and therefore completely different outcomes are possible.

For example, from the `ready` state the transitions `fault` and `1` lead to completely different states that can never be reconciled in a common subsequent state.

Concurrency vs. Nondeterminism: Automata (III)

So, the automaton is in this sense nondeterminisitc, yet it is strictly sequential, in the sense that, although at each state the automaton may be able to take several transitions, it can only take one transition at a time.

Since the intuitive notion of concurrency is that several transitions can happen simultaneously, we can conclude by saying the our automaton, although it exhibits a form of nondeterminism, has no concurrency whatsoever.

Automata as Rewrite Theories

We can specify such an automaton as a system module,

```
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
```

Rewite 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 axioms B, 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
```

The `rewrite` command applies the rules in a fair way (all rules are given a chance) hopefully until termination, and, if it terminates, gives one result.

The `rewrite` Command (II)

In this example, fairness saves us from nontermination, but in general we can esily have nonterminating computations.

For this reason the `rewrite` command can be given a numeric argument stating the maximum number of rewrite steps. Furthermore, using Maude's the `trace` command we can observe such steps. For example,

The `rewrite` Command (III)

```
Maude> set trace on .
Maude> rew [3] $ .
rewrite [3] in CANDY-AUTOMATON : $ .
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
*********** rule
rl [cancel]: ready => $ .
empty substitution
ready ---> $
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result State: ready
```

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 (II)

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

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken

Solution 2 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

We can then inspect the search graph by giving the command,

The `search` Command (III)

```
Maude> show search graph .
state 0, State: $
arc 0 ===> state 1 (rl [in]: $ => ready .)

state 1, State: ready
arc 0 ===> state 0 (rl [cancel]: ready => $ .)
arc 1 ===> state 2 (rl [1]: ready => nestle .)
arc 2 ===> state 3 (rl [2]: ready => m&m .)
arc 3 ===> state 4 (rl [fault]: ready => broken .)

state 2, State: nestle
arc 0 ===> state 5 (rl [chng]: nestle => q .)

state 3, State: m&m
arc 0 ===> state 5 (rl [chng]: m&m => q .)

state 4, State: broken
state 5, State: q
```

The `search` Command (IV)

We can then ask for the shortest path to any state in the state graph (for example, state 5) by giving the command,

```
Maude> show path 5 .
state 0, State: $
===[ rl [in]: $ => ready . ]===>
state 1, State: ready
===[ rl [1]: ready => nestle . ]===>
state 2, State: nestle
===[ rl [chng]: nestle => q . ]===>
state 5, State: q
```

The `search` Command (V)

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

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

The `search` Command (VI)

Furthermore, we can restrict any of those searches by giving an equational condition on the target term. For example, all terminating states reachable from `$` other than `broken` can be found by the command,

```
Maude> search $ =>! X:State such that X:State =/= broken .
search in CANDY-AUTOMATON : $ =>! X:State
such that X:State =/= broken = true .

Solution 1 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

The `search` Command (VII)

Of course, in general there can be an infinite number of solutions to a given search. Therefore, a search can be further restricted by giving as an extra parameter in brackets the number of solutions (i.e., target terms that are instances of the pattern and satisfy the condition) we want:

```
search [1] in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken
```

The `search` Command (VIII)

In our CANDY-AUTOMATON example the number of states is finite, but for a general rewrite theory the number of states reachable from an initial state can be infinite. So, even if we search for a single solution, the search process may not terminate, because no such solution exists. To make search terminating, at least for unconditional rewrite rules, we can add a second parameter, namely, a bound on the length of the paths searched from the initial state.

```
search [1, 1] in CANDY-AUTOMATON : $ =>! X:State .

No solution.
states: 2  rewrites: 2 in 0ms cpu (36ms real) (~ rewrites/second)
```

Labelled Transition Systems

Our `CANDY-AUTOMATON` example is just a special instance of a general concept, namely, that of automaton, also called a labeled transition system (LTS) by which we mean a triple: $\mathcal{A} = (A, L, T)$ with:

- A is a set, called the set of states,
- L is a set called the set of labels, and
- $T \subseteq A \times L \times A$ is called the set of labeled transitions.

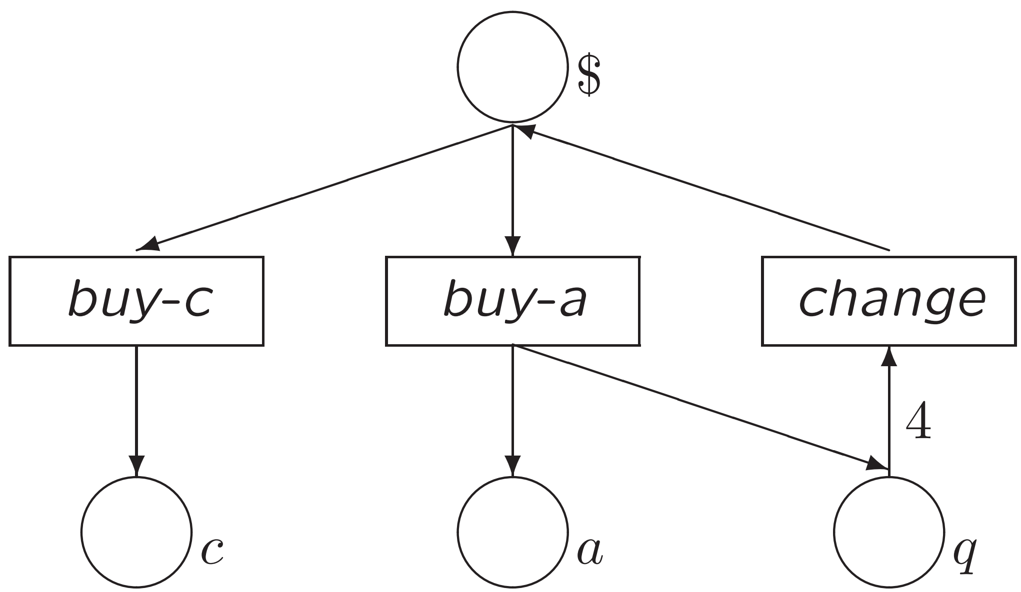
LTS's as Rewrite Theories

Note that we have associated to our candy automaton a rewrite theory (system module) `CANDY-AUTOMATON`.

This is of course just an instance of a general transformation, that assign to a LTS A a rewrite theory $R(A)$ with a single sort A, constants $x \in A$, and for each $(x, l, y) \in T$ a rewrite rule $l : x \longrightarrow y$.

Petri Nets

So far so good, but we have not yet seen any concurrency. The simplest concurrent system examples are probably the concurrent automata called Petri nets. Consider for example the picture,



Petri Nets (II)

The previous picture represents a concurrent machine to buy cakes and apples; a cake costs a dollar and an apple three quarters.

Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar.

The machine is concurrent, because we can push several buttons at once, provided enough resources exist in the corresponding slots, which are called places

Petri Nets (III)

For example, if we have one dollar in the $\$$ place, and four quarters in the q place, we can simultaneously push the *buy-a* and *change* buttons, and the machine returns, also simultaneously, one dollar in $\$$, one apple in a, and one quarter in q.

That is, we can achieve the concurrent computation,

$$\textit{buy-a}\ \ \textit{change} : \$\, q\, q\, q\, q \longrightarrow a\, q\, \$.$$

Petri Nets (IV)

This has a straightforward expression as a rewrite theory (system module) as follows:

```
mod PETRI-MACHINE is
  sort Marking .
  ops null $ c a q : -> Marking .
  op _ _ : Marking Marking -> Marking [assoc comm id: null] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [chng] : q q q q => $ .
endm
```

Petri Nets (V)

That is, we view the distributed state of the system as a multiset of places, called a marking, with identity for multiset union the empty multiset `null`.

We then view a transition as a rewite rule from one (pre-)marking to another (post-)marking.

Petri Nets (VI)

The rewrite rule can be applied modulo associativity, commutativity and identity to the distributed state iff its pre-marking is a submultiset of that state.

Furthermore, if the distributed state contains the union of several such presets, then several transitions can fire concurrently.

For example, from `$ $ $` we can get in one concurrent step to `c c a q` by pushing twice (concurrently!) the `buy-c` button and once the `buy-a` button.

Petri Nets (VII)

We can of course ask and get answers to questions about the behaviors possible in this system. For example, if I have a dollar and three quarters, can I get a cake and an apple?

```
Maude> search $ q q q =>+ c a M:Marking .
search in PETRI-MACHINE : $ q q q =>+ c a M:Marking .

Solution 1 (state 4)
states: 5 in 0ms cpu (0ms real)
M:Marking --> null
```

we can also interrogate the search graph,

Petri Nets (VIII)

```
Maude> show search graph .
state 0, Marking: $ q q q
arc 0 ===> state 1 (rl [buy-c]: $ => c .)
arc 1 ===> state 2 (rl [buy-a]: $ => a q .)

state 1, Marking: c q q q

state 2, Marking: a q q q q
arc 0 ===> state 3 (rl [chng]: q q q q => $ .)

state 3, Marking: $ a
arc 0 ===> state 4 (rl [buy-c]: $ => c .)
arc 1 ===> state 5 (rl [buy-a]: $ => a q .)

state 4, Marking: c a

state 5, Marking: a a q
```

Petri Nets (IX)

```
Maude> show path 4 .
state 0, Marking: $ q q q
===[ rl [buy-a]: $ => a q . ]===>
state 2, Marking: a q q q q
===[ rl [chng]: q q q q => $ . ]===>
state 3, Marking: $ a
===[ rl [buy-c]: $ => c . ]===>
state 4, Marking: c a
```

What is Concurrency?

Why was concurrency impossible in our `CANDY-AUTOMATON` example, but possible in our little `PETRI-MACHINE` example?

The problem with `CANDY-AUTOMATON`, and with any LTS having unstructured states, is that its states are atomic, and, having no smaller pieces, cannot be distributed.

By contrast, a Petri net marking is made out of smaller pieces, namely its constituent places, and therefore can be distributed, so that several transitions can happen simultaneously.

What is Concurrency? (II)

Then what, is concurrency about multisets?

Not necessarily; this is the very common fallacy of taking the part for the whole; for example, "Logic Programming = Prolog," or "Concurrency = Petri Nets".

A more fair and open-minded answer is to give the rewriting logic motto:

Concurrent Structure = Algebraic Structure.

What is Concurrency? (III)

That is, any algebraic structure in the set of states, other than atomic constants, even a single unary operator, will open the possibility for the states to be distributed, and therefore for transitions being concurrent.

Of course that potential for concurrency may be frustrated by the specific transitions of a system forcing a sequential execution, but the potential is there if we use other transitions.

In summary, there are as many possible styles of concurrent systems as there are signatures Σ and equations E. For example: multiset concurrency, tree concurrency, string concurrency, and many, many other possibilities.

Petri Nets in General

I give the Meseguer-Montanari "Petri nets are monoids" definition, instead than the usual, but less enlightening, multigraph definition.

A place-transition Petri net N consists of:

- a set P of places; we then call markings to the elements in the free commutative monoid $M(P)$ of finite multisets of P.

- a labeled transition system $N = (M(P), L, T)$.

Petri Nets in General (II)

The general transformation associating a rewrite theory $R(N)$ to each Petri net N is then obvious. $R(N)$ has:

- a single sort, named, say $M(P)$, or just *Marking*, with constants the elements of P and a *null* constant.

- a binary operator
  $\_\ \_ :$ *Marking Marking* $\longrightarrow$ *Marking* $[assoc\ comm\ id :\ null]$

- for each $(m, l, m') \in T$ a rewrite rule $l : m \longrightarrow m'$.

Petri Net Computations

The computations of a net N are not just paths, since we can now take several concurrent steps at once. They are generated as follows:

- **Reflexivity**.

$$\frac{m \in M(P)}{m \xrightarrow{m} m}$$

- **Basic Transition**.

$$\frac{(m, l, m') \in T}{m \xrightarrow{l} m'}$$

- **Congruence**.

$$\frac{m \xrightarrow{\alpha} m' \qquad u \xrightarrow{\beta} u'}{m\ u \xrightarrow{\alpha\ \beta} m'\ u'}$$

Petri Net Computations (II)

- **Transitivity**.

$$\frac{m \xrightarrow{\alpha} u \qquad u \xrightarrow{\beta} v}{m \xrightarrow{\alpha;\beta} v}$$

We will see later that, when we view Petri nets as rewite theories, the above inference system generating all Petri net computations of a net N coincides with the specialization of the general inference system of rewriting logic to the rewrite theory $R(N)$.

This illustrates a general point, namely, that rewriting logic is a very expressive semantic framework, in which many different concurrency models can be naturally specified.