Program Verification: Lecture 17

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Programming Concurrent Systems with Rewrite Theories

Up to now we have consider Maude's sublanguage of functional modules in equational logic. Maude's full language uses system modules in rewriting logic to program concurrent systems.

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

- (Σ, E) an order-sorted equational theory, and
- R a set of (possibly conditional) labeled rewrite rules of the form $l : t \longrightarrow t'$ *if* *cond*, with l a label, t, t' Σ-terms, and *cond* a condition or guard.

Maude System Modules

In Maude, rewrite theories are specified in system modules of the form:

$$\texttt{mod } (\Sigma, E, R) \texttt{ endm}$$

with (Σ, E, R) a rewrite theory.

A conditional rewrite rule of the form, $l : t \longrightarrow t' \;\; \mathit{if} \; \mathit{cond}$ is specified in Maude with syntax,

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

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

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

In both cases the rule's label $[l]$ may be omitted.

Rewriting Logic is a Semantic Framework for Concurrency

Rewriting logic naturally expresses concurrent computation as concurrent rewriting, and can model, for example,

1. Petri Nets
2. Process Calculi like CCS and the π-Calculus
3. Grammars and Tree Automata
4. Data Flow Networks
5. Concurrent Object Systems

very naturally and without any encodings.

To illustrate the ideas, we will focus on Concurrent Object Systems, which are the most common and natural way to model and program distributed systems.

Concurrent Objects in Rewriting Logic

In Concurrent object systems, objects interact with other objects, typically by asynchronous message passing.

A distributed state, called a configuration, is a multiset or “soup” of objects and messages, built up by an *ACU* union operator with empty syntax (i.e. juxtaposition) as:

```
subsorts Object Msg < Configuration .

op none : ->  Configuration .
op __ : Configuration Configuration -> Configuration
                     [ctor config assoc comm id: none] .
```

Objects and Messages

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 name, the a_i's are the names of the object's attribute identifiers, and the v_i's are the corresponding values, declared in Maude as:

```
op <_:_|_> : Oid Class Atts -> Object [ctor] .
op _,_ : Atts Atts -> Atts [ctor assoc comm id: null] .
```

The user can choose any syntax for messages (will see an example).

A Communication Protocol Example

Consider `Sender` and `Receiver` classes, where a `Sender` (resp. a `Receiver`) sends (resp. receives) elements from an AU-list of numbers (with constructors `nil` and `_;_`) and has the form:

```
vars N M : Nat . var L : List .  vars A B : Oid .  var TV : Bool .
```

```
< A : Sender | buff: L, rec: B, cnt: M, ack-w: TV >
```

```
< B : Receiver | buff: L, snd: A, cnt: M >
```

They use respective messages of the form:

```
msg to_from_val_cnt_ : Oid Oid Nat Nat -> Msg [ctor] .
```

```
msg to_from_ack_ : Oid Oid Nat -> Msg [ctor] .
```

Their communication protocol is defined by the rules:

A Communication Protocol Example (II)

```
rl [snd] : < A : Sender | buff: (N ; L),rec: B,cnt: M,ack-w: false >
=> (to B from A val N cnt M)
    < A : Sender | buff: L,rec: B,cnt: M,ack-w: true > .

rl [rec] : (to B from A val N cnt M)
            < B : Receiver | buff: L,snd: A,cnt: M >
=> < B : Receiver | buff: (L ; N),snd: A,cnt: s(M) >
   (to A from B ack M) .

rl [ack-rec] : (to A from B ack M)
               < A : Sender | buff: L, rec: B, cnt: M, ack-w: true >
=> < A : Sender | buff: L, rec: B, cnt: s(M), ack-w: false > .
```

Since communication is asynchronous, counters and acknowledgements are used to ensure in-order communication.

The `rewrite` Command

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

```
Maude> rew
< 'a : Sender | buff: (1 ; 2 ; 3 ; 4 ; 5),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 > .

result Configuration:
< 'a : Sender | buff: nil,rec: 'b,cnt: 5,ack-w: false >
< 'b : Receiver | buff: (1 ; 2 ; 3 ; 4 ; 5),snd: 'a,cnt: 5 >
```

The `rewrite` command applies the rules in a fair way (all rules are given a chance); and for object systems the `frewrite` command does so in an object- and message-fair manner. Rules are applied until termination, and, if it terminates, a result is given.

The `rewrite` Command (II)

In this example, the rules always terminate, but in general we can easily 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 `trace` command we can observe each of these steps. For example,

The **rewrite** Command (III)

```
Maude> set trace on .
Maude> rew [3] < 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 > .
*********** rule
rl < A : Sender | buff: (N ; L),rec: B,cnt: M,ack-w: false > =>
< A : Sender | buff: L, rec: B,cnt: M,ack-w: true > to B from A val N cnt M
[label snd] .

< 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
--->
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
to 'b from 'a val 1 cnt 0
*********** rule
rl < B : Receiver | buff: L,snd: A,cnt: M > to B from A val N cnt M =>
< B : Receiver | buff: (L ; N),snd: A,cnt: s M > to A from B ack M [label rec] .

< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 > to 'b from 'a val 1 cnt 0
```

```
--->
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
< 'b : Receiver | buff: (nil ; 1),snd: 'a,cnt: 1 > to 'a from 'b ack 0
*********** rule
rl < A : Sender | buff: L,rec: B,cnt: M,ack-w: true > to A from B ack
M => < A : Sender | buff: L,rec: B,cnt: s M,ack-w: false > [label ack-rec] .

< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 > to 'a from 'b ack 0
--->
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 >
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 1, ack-w: false >

result Configuration:
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 1,ack-w: false >
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 >
```

The `search` Command

Concurrent systems can be nondeterministic. The `rewrite` command gives us one possible behavior among many.

To explore all behaviors from an initial state we can use the `search` command, which takes two terms: a ground term which the chosen initial state, and a constructor term, possibly with variables, which specifies a class of target states as term instances.

Maude then does a breadth first search for target states. For example, to find all terminating states from state `< 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false > < 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >` we can give the command (where the "`!`" in `=>!` specifies that the target state must be a terminating state),

The `search` Command (II)

```
Maude> search < 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 > =>! C:Configuration .

Solution 1 (state 9)
states: 10  rewrites: 9 in 7ms cpu (34ms real) (1268 rewrites/second)
C --> < 'a : Sender | buff: nil,rec: 'b,cnt: 3,ack-w: false >
< 'b : Receiver | buff: (1 ; 2 ; 3),snd: 'a,cnt: 3 >
No more solutions.
```

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

The search Command (III)

```
Maude> show search graph .
state 0, Configuration:
< 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >
arc 0 ===> state 1 (rl < A : Sender | buff: (N ; L),rec: B,cnt: M,ack-w: false >
=> < A : Sender | buff: L,rec: B,cnt: M,ack-w: true > to B from A val N cnt M
[label snd] .)

state 1, Configuration:
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >
to 'b from 'a val 1 cnt 0
arc 0 ===> state 2 (rl < B : Receiver | buff: L,snd: A,cnt: M >
                    to B from A val N cnt M =>
< B : Receiver | buff: (L ; N),snd: A,cnt: s M > to A from B ack M [label rec] .

state 2, Configuration:
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
```

```
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 > to 'a from 'b ack 0
arc 0 ===> state 3 (rl < A : Sender | buff: L,rec: B,cnt: M,ack-w: true >
                   to A from B ack M =>
< A : Sender | buff: L,rec: B,cnt: s M,ack-w: false > [label ack-rec] .)

state 3, Configuration:
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 1,ack-w: false >
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 >
arc 0 ===> state 4 (rl < A : Sender | buff: (N ; L),rec: B,cnt: M,ack-w: false >
=> < A : Sender | buff: L,rec: B,cnt: M,ack-w: true >
   to B from A val N cnt M [label snd] .)

state 4, Configuration:
< 'a : Sender | buff: 3,rec: 'b,cnt: 1,ack-w: true >
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 > to 'b from 'a val 2 cnt 1
arc 0 ===> state 5 (rl < B : Receiver | buff: L,snd: A,cnt: M >
                       to B from A val N cnt M =>
< B : Receiver | buff: (L ; N),snd: A,cnt: s M > to A from B ack M [label rec] .

state 5, Configuration:
< 'a : Sender | buff: 3,rec: 'b,cnt: 1,ack-w: true >
```

```
< 'b : Receiver | buff: (1 ; 2),snd: 'a,cnt: 2 > to 'a from 'b ack 1
arc 0 ===> state 6 (rl < A : Sender | buff: L,rec: B,cnt: M,ack-w: true >
                       to A from B ack M =>
< A : Sender | buff: L,rec: B,cnt: s M,ack-w: false > [label ack-rec] .)

state 6, Configuration:
< 'a : Sender | buff: 3,rec: 'b,cnt: 2,ack-w: false >
< 'b : Receiver | buff: (1 ; 2),snd: 'a,cnt: 2 >
arc 0 ===> state 7 (rl < A : Sender | buff: (N ; L),rec: B,cnt: M,ack-w: false >
                    => < A : Sender | buff: L,rec: B,cnt: M,ack-w: true >
                       to B from A val N cnt M [label snd] .)

state 7, Configuration:
< 'a : Sender | buff: nil,rec: 'b,cnt: 2,ack-w: true >
< 'b : Receiver | buff: (1 ; 2),snd: 'a,cnt: 2 > to 'b from 'a val 3 cnt 2
arc 0 ===> state 8 (rl < B : Receiver | buff: L,snd: A,cnt: M >
                       to B from A val N cnt M =>
 < B : Receiver | buff: (L ; N),snd: A,cnt: s M > to A from B ack M [label rec]

state 8, Configuration:
< 'a : Sender | buff: nil,rec: 'b,cnt: 2,ack-w: true >
```

```
< 'b : Receiver | buff: (1 ; 2 ; 3),snd: 'a,cnt: 3 > to 'a from 'b ack 2
arc 0 ===> state 9 (rl < A : Sender | buff: L,rec: B,cnt: M,ack-w: true >
                       to A from B ack M =>
         < A : Sender | buff: L,rec: B,cnt: s M,ack-w: false > [label ack-rec] .

state 9, Configuration:
< 'a : Sender | buff: nil,rec: 'b,cnt: 3,ack-w: false >
< 'b : Receiver | buff: (1 ; 2 ; 3),snd: 'a,cnt: 3 >
```

The **search** Command (IV)

We can then ask for the shortest path to any state in the state graph (for example, state 3) by giving the command,

```
Maude> show path 3 .
state 0, Configuration:
< 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >
===[ rl < A : Sender | buff: (N ; L),rec: B,cnt: M,ack-w: false > =>
< A : Sender | buff: L,rec: B,cnt: M,ack-w: true > to B from A val N cnt M
[label snd] . ]===>
state 1, Configuration:
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 > to 'b from 'a val 1 cnt 0
===[ rl < B : Receiver | buff: L,snd: A,cnt: M > to B from A val N cnt M =>
< B : Receiver | buff: (L ; N),snd: A,cnt: s M > to A from B ack M
[label rec] . ]===>
state 2, Configuration:
```

```
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 0,ack-w: true >
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 > to 'a from 'b ack 0
===[ rl < A : Sender | buff: L,rec: B,cnt: M,ack-w: true > to A from B ack M =>
< A : Sender | buff: L,rec: B,cnt: s M,ack-w: false > [label ack-rec] . ]===>
state 3, Configuration:
< 'a : Sender | buff: (2 ; 3),rec: 'b,cnt: 1,ack-w: false >
< 'b : Receiver | buff: 1,snd: 'a,cnt: 1 >
```

The `search` Command (V)

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

- `search` t `=>1` t' .
- `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 states reachable from `< 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false > < 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >` such that the value in the sender's counter is different from the value in the receiver's counter can be found by the command,

```
Maude> search < 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >  =>*
< 'a : Sender | buff: L,rec: 'b,cnt: N,ack-w: TV > C:Configuration
< 'b : Receiver | buff: Q,snd: 'a,cnt: M > such that N =/= M .

Solution 1 (state 2)
C:Configuration --> to 'a from 'b ack 0
L --> 2 ; 3
```

```
N --> 0
TV --> true
Q --> 1
M --> 1

Solution 2 (state 5)
C:Configuration --> to 'a from 'b ack 1
L --> 3
N --> 1
TV --> true
Q --> 1 ; 2
M --> 2

Solution 3 (state 8)
C:Configuration --> to 'a from 'b ack 2
L --> nil
N --> 2
TV --> true
Q --> 1 ; 2 ; 3
M --> 3
No more solutions.
```

The **search** Command (VII)

A search can be further restricted by giving as an extra parameter in brackets the number of solutions we want:

```
Maude> search [1] < 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >  =>*
< 'a : Sender | buff: L,rec: 'b,cnt: N,ack-w: TV > C:Configuration
< 'b : Receiver | buff: Q,snd: 'a,cnt: M > such that N =/= M .

Solution 1 (state 2)
C:Configuration --> to 'a from 'b ack 0
L --> 2 ; 3
N --> 0
TV --> true
Q --> 1
M --> 1
```

The **search** Command (VIII)

In our communication protocol example the number of reachable states for an initial state was 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, we can add as a second parameter a bound on the length of the paths searched from the initial state.

```
search [1, 1]  < 'a : Sender | buff: (1 ; 2 ; 3),rec: 'b,cnt: 0,ack-w: false >
< 'b : Receiver | buff: nil,snd: 'a,cnt: 0 >  =>*
< 'a : Sender | buff: L,rec: 'b,cnt: N,ack-w: TV > C:Configuration
< 'b : Receiver | buff: Q,snd: 'a,cnt: M > such that N =/= M .

No solution.
```

Why is Rewriting Intrinsically Concurrent?

Up to now our description of rewriting computations has been sequential: either a single step rewrite $u \to_{R/B} v$, or a sequence of 0, 1, or more rewrites $u \to^{*}_{R/B} v$, So, where is the concurrency? In what sense rewrite theories provide a semantic framework for concurrency?

This question can be answered at an intuitive level by observing that rules rewrite a local fragment of the distributed state. For example, the rules `snd`, `rec`, and `ack-rec` in our communication protocol only affect the addresse object and the corresponding message. In a configuration with thousands of sender and receiver objects, many rewrites with these rules can happen concurrently, that is, simultaneously and independently of each other.

Why is Rewriting Intrinsically Concurrent? (II)

At the logical level, the same question has been answered by rewriting logic, a logic where:

- Concurrent computation is logical deduction, and
- Programming concurrent systems is mathematical modeling.

A proof in rewriting logic directly expresses a concurrent computation. Such a proof can have many sequential descriptions (called interleavings) of the form $u \rightarrow^{*}_{R/B} v$; but such interleavings hide the actual concurrency.

An Appendix to this talk contains a link to an early paper on rewriting logic[a] explaining both the inference rules and the models.

[a] J. Meseguer, "Rewriting as a Unified Model of Concurrency" in Proc. CONCUR 1990, Springer LNCS 458, 384–400, 1990.