Program Verification: Lecture 18

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

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 . vars L Q : 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.

Using Maude's `rewrite` Command

Recall from Lecture 17 that we can execute rewrite theories with Maude's `rewrite` command (abbreviated: `rew`). E.g.,

```
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), cnt : 5, snd : 'a >
```

Recall also that 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. E.g.,

Using Maude's **rewrite** Command (II)

```
Maude> set trace on .
Maude> rew [3] < 'a : Sender | buff : (1 ; 2 ; 3 ; 4 ; 5), rec : 'b, cnt : 0,
ack-w : false >
< 'b : Receiver | buff : nil, snd : 'a, cnt : 0 > .
********** rule
rl < A : V:Sender | Atts:AttributeSet, buff : (N ; L), rec : B, cnt :
M, ack-w : false > =>
< A : V:Sender | Atts:AttributeSet, 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 ; 4 ; 5), rec : 'b, cnt : 0, ack-w : false >
--->
< 'a : Sender | none, buff : (2 ; 3 ; 4 ; 5), rec : 'b, cnt : 0, ack-w : true >
to 'b from 'a val 1 cnt 0
********** rule
rl < B : V:Receiver | Atts:AttributeSet, buff : L, cnt : M, snd : A >
to B from A val N cnt M =>
< B : V:Receiver | Atts:AttributeSet, buff : (L ; N), cnt : s M, snd : A >
```

```
to A from B ack M [label rec] .

< 'a : Sender | buff : (2 ; 3 ; 4 ; 5), rec : 'b, cnt : 0, ack-w : true >
< 'b : Receiver | buff : nil, cnt : 0, snd : 'a > to 'b from 'a val 1 cnt 0
--->
< 'a : Sender | buff : (2 ; 3 ; 4 ; 5), rec : 'b, cnt : 0, ack-w : true >
< 'b : Receiver | none, buff : (nil ; 1), cnt : 1, snd : 'a >
to 'a from 'b ack 0
********** rule
rl < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M,
ack-w : true > to A from B ack M =>
< A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : s M,
ack-w : false > [label ack-rec] .

< 'a : Sender | buff : (2 ; 3 ; 4 ; 5), rec : 'b, cnt : 0, ack-w : true >
< 'b : Receiver | buff : 1, cnt : 1, snd : 'a > to 'a from 'b ack 0
--->
< 'b : Receiver | buff : 1, cnt : 1, snd : 'a >
< 'a : Sender | none, buff : (2 ; 3 ; 4 ; 5), rec : 'b, cnt : 1, ack-w : false >
```

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

Using Maude's `search` Command

Further recall from Lecture 17 that we can explore the states reachable from an initial states and search for specific kinds of reachable states with Maude's `search` command. E.g., we can ask for the set of all terminating reachable states as follows:

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

No more solutions.
```

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

Using Maude's **search** Command (II)

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

```
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, cnt : 1, snd : 'a > to 'a from 'b ack 0
arc 0 ===> state 3
(rl < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M,
ack-w : true > to A from B ack M =>
< A : V:Sender | Atts:AttributeSet, 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, cnt : 1, snd : 'a >
arc 0 ===> state 4
(rl < A : V:Sender | Atts:AttributeSet, buff : (N ; L), rec : B, cnt : M,
ack-w : false > =>
< A : V:Sender | Atts:AttributeSet, 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, cnt : 1, snd : 'a >
to 'b from 'a val 2 cnt 1
arc 0 ===> state 5
(rl < B : V:Receiver | Atts:AttributeSet, buff : L, cnt : M, snd : A >
to B from A val N cnt M =>
< B : V:Receiver | Atts:AttributeSet, buff : (L ; N), cnt : s M, snd : A >
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), cnt : 2, snd : 'a >
to 'a from 'b ack 1
arc 0 ===> state 6
(rl < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M,
    ack-w : true > to A from B ack M =>
< A : V:Sender | Atts:AttributeSet, 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), cnt : 2, snd : 'a >
arc 0 ===> state 7
(rl < A : V:Sender | Atts:AttributeSet, buff : (N ; L), rec : B, cnt : M,
ack-w : false > =>
< A : V:Sender | Atts:AttributeSet, 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), cnt : 2, snd : 'a >
to 'b from 'a val 3 cnt 2
arc 0 ===> state 8
(rl < B : V:Receiver | Atts:AttributeSet, buff : L, cnt : M, snd : A >
to B from A val N cnt M =>
< B : V:Receiver | Atts:AttributeSet, buff : (L ; N), cnt : s M, snd : A >
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), cnt : 3, snd : 'a >
to 'a from 'b ack 2
```

```
arc 0 ===> state 9
(rl < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M,
    ack-w : true > to A from B ack M =>
< A : V:Sender | Atts:AttributeSet, 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), cnt : 3, snd : 'a >
```

Using Maude's **search** Command (III)

And 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, cnt : 0, snd : 'a >
===[ rl < A : V:Sender | Atts:AttributeSet, buff : (N ; L), rec : B,
cnt : M, ack-w : false > =>
< A : V:Sender | Atts:AttributeSet, 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, cnt : 0, snd : 'a > to 'b from 'a val 1 cnt 0
===[ rl < B : V:Receiver | Atts:AttributeSet, buff : L, cnt : M, snd : A >
to B from A val N cnt M =>
< B : V:Receiver | Atts:AttributeSet, buff : (L ; N), cnt : s M, snd : A >
```

```
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, cnt : 1, snd : 'a > to 'a from 'b ack 0
===[ rl < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M,
ack-w : true > to A from B ack M =>
< A : V:Sender | Atts:AttributeSet, 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, cnt : 1, snd : 'a >
```

Using Maude's **search** Command (IV)

Recall also that we can restrict a search command 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)
states: 3  rewrites: 5 in 0ms cpu (0ms real) (32679 rewrites/second)
```

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

Solution 2 (state 5)
states: 6  rewrites: 11 in 0ms cpu (0ms real) (40293 rewrites/second)
C:Configuration --> to 'a from 'b ack 1
L --> 3
N --> 1
TV --> true
Q --> 1 ; 2
M --> 2

Solution 3 (state 8)
states: 9  rewrites: 17 in 0ms cpu (0ms real) (44736 rewrites/second)
C:Configuration --> to 'a from 'b ack 2
L --> nil
N --> 2
```

```
TV --> true
Q --> 1 ; 2 ; 3
M --> 3

No more solutions.
```

Using Maude's **search** Command (V)

Remember that 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)
states: 3  rewrites: 5 in 0ms cpu (0ms real) (30303 rewrites/second)
C:Configuration --> to 'a from 'b ack 0
L --> 2 ; 3
N --> 0
TV --> true
Q --> 1
M --> 1
```

Using Maude's **search** Command (VI)

In our communication protocol example the number states of reachable from an initial state is finite; but for a general rewrite theory the set of states reachable from an initial state can be infinite. Remember from Lecture 17 that, to make search terminating, we can add as a second parameter a bound on the length of the paths searched from the initial state.

```
Maude> 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.
```

Maude's Object-Oriented Modules

Our communication protocol example can be specified in Maude as a system module. However, to help users directly express object-oriented concepts such as:

1. Objects and Messages
2. Object Classes, and Class Inheritance

Maude 3.3.1 allows users to specify concurrent object systems in object-oriented modules with the `omod` and `endom` keywords.

Maude's object-oriented modules provide syntactic sugar to make explicit the object-oriented concepts (1)–(2). However, this useful information can be desugared into the syntax of a usual Maude system module to make explicit the actual rewrite theory giving a rewriting logic semantics to the given concurent object system. Let us see how our communication protocol is thus specified.

Maude's Object-Oriented Modules (II)

```
fmod NAT-LIST is
 protecting NAT .
 sort List .
 subsorts Nat < List .
 op nil : -> List .
 op _;_ : List List -> List [assoc id: nil] .
 endfm

omod COMM is protecting NAT-LIST .
 protecting  QID .
 subsort Qid < Oid .
 class Sender |  buff : List, rec : Oid, cnt : Nat, ack-w : Bool .
 class Receiver |  buff : List, snd : Oid, cnt : Nat .
 msg to_from_val_cnt_ : Oid Oid Nat Nat -> Msg .
 msg to_from_ack_ : Oid Oid Nat -> Msg .

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

```
 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, cnt : M, ack-w : true  > .

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

Internally, Maude transforms any object-oriented module into a semantically equivalent desugared system module, which we can make explicit by giving the command:

Maude's Object-Oriented Modules (III)

```
Maude> show desugared .
mod COMM is including BOOL . including CONFIGURATION .
  protecting NAT-LIST . protecting QID .
  sorts Sender Receiver .  subsort Qid < Oid .
  subsorts Sender Receiver < Cid .
  op to_from_val_cnt_ : Oid Oid Nat Nat -> Msg
                                        [ctor msg prec 41 gather (& & & E)] .
  op to_from_ack_ : Oid Oid Nat -> Msg [ctor msg prec 41 gather (& & E)] .
  op Sender : -> Sender [ctor] .
  op buff`:_ : List -> Attribute [ctor prec 15 gather (&)] .
  op buff`:_ : List -> Attribute [ctor prec 15 gather (&)] .
  op rec`:_ : Oid -> Attribute [ctor prec 15 gather (&)] .
  op cnt`:_ : Nat -> Attribute [ctor prec 15 gather (&)] .
  op cnt`:_ : Nat -> Attribute [ctor prec 15 gather (&)] .
  op ack-w`:_ : Bool -> Attribute [ctor prec 15 gather (&)] .
  op Receiver : -> Receiver [ctor] .
  op snd`:_ : Oid -> Attribute [ctor prec 15 gather (&)] .
  var A : Oid . var B : Oid . var N : Nat . var M : Nat .
```

```
  var Q : List . var L : List . var TV : Bool .
  rl < A : V:Sender | Atts:AttributeSet, buff : (N ; L), rec : B, cnt : M,
    ack-w : false > => < A :
    V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M, ack-w : true >
    to B from A val N cnt M [label snd] .
  rl < B : V:Receiver | Atts:AttributeSet, buff : L, cnt : M, snd : A >
    to B from A val N cnt M =>
    < B : V:Receiver | Atts:AttributeSet, buff : (L ; N), cnt : s M, snd : A >
    to A from B ack M [label rec] .
  rl < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : M,
     ack-w : true > to A from B ack M =>
    < A : V:Sender | Atts:AttributeSet, buff : L, rec : B, cnt : s M,
    ack-w : false > [label ack-rec] .
endm
```

The desugared module makes clear that the rewrite rules can be inherited by subclasses, thanks to: (1) the `Atts:AttributeSet` variable to match extra attributes in a subclass, and (2) the variables `V:Sender` and `V:Receiver` to match objects in respective subclasses. For more details see §6.5 of the Maude 3.3.1 Manual.