Modeling and Analysis in Maude

Lecture 3: Modeling parallel & distributed systems

Einar Broch Johnsen

University of Oslo, Norway
einarj@ifi.uio.no

DAT355, 13 April 2021

UNIVERSITY OF OSLO

SIRIUS Center for Scalable Data Access in the Oil and Gas Domain

sfi Centre for Research-based Innovation

The Research Council of Norway

http://www.sirius-labs.no

Plan for the lectures

- **Lecture 1:** Basic ideas and concepts of rewriting logic & Maude
- **Lecture 2:** From equations to rules
- **Lecture 3:** Modeling parallel & distributed systems
- **Lecture 4:** Analyzing models

Modeling parallel & distributed systems

Summary from last week

Functional modules in Maude

- Define Sorts with constructors
- Other functions defined over the constructors
- How do we decide if two terms are equal?
- Termination, confluence, (unique) normal forms
- Collections in Maude: lists, sets, multisets
- Rewriting logic: transition rules between equivalence classes (why?)
- Operational semantics
- Guarded commands

GCL: A Model of Parallel Programs

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \sigma\ \{\ g_1 \mid\mid \ldots \mid\mid g_n\ \} \\
\sigma \in \mathit{State} & ::= & \epsilon \mid \sigma[x \mapsto v] \\
g \in \mathit{GrdStm} & ::= & \mathtt{skip} \mid g; g \\
 & \mid & e \rhd s \mid s \lhd e \rhd s \\
s \in \mathit{Stm} & ::= & x := e \mid \mathtt{spawn}(g) \\
e \in \mathit{Exp} & ::= & \mathtt{true} \mid \mathtt{false} \mid x \mid v \mid e + e \\
 & \mid & e * e \mid e < e \mid e =:= e \\
 & \mid & e\ \mathit{and}\ e \mid e\ \mathit{or}\ e \mid \mathit{not}(e)
\end{array}
$$

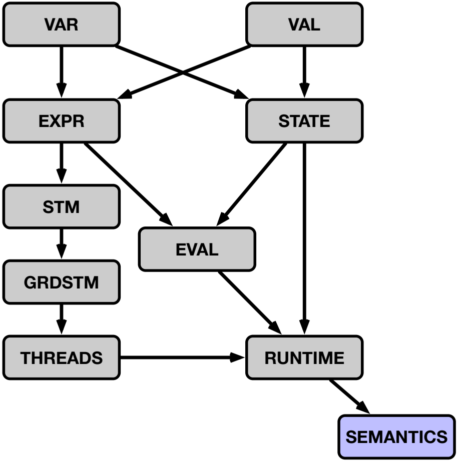
The Guarded Command Language (GCL), invented by Edsger Dijkstra, presents programming concepts in a compact way, before the program is written in some practical programming language. Its simplicity makes reasoning about programs easier.

Informal semantics of GCL

Goal: to make a Maude model of GCL

- We consider two types of values, defined by the sorts Bool and Int
- A runtime state $\sigma \; \{ \; g_1 \; || \; \ldots \; || \; g_n \; \}$ consists of a state σ and a "thread pool" with the threads $g_1, \ldots, g_n$
- We can execute a "thread" g_i from the thread pool if it is enabled in the current state (the guard evaluates to `true`)
  - The guarded statement **`skip`**: the thread can be terminated
  - The guarded statement $e \rhd s$: if the guard e is true in σ, execute s
  - The guarded statement $s_1 \lhd e \rhd s_2$: if the guard e is true in σ, execute s_1, otherwise execute s_2
  - The guarded statement $g_1 ; g_2$ schedules and executes g_1 as above, then returns g_2 to the thread pool
  - The statement $x := e$ evaluates e to some value v in the state σ, then updates x to v to produce the next state
  - The statement $\mathbf{spawn}(g)$ adds g to the thread pool

Structure of the model



- Gray box = fmod (equations)
- Blue box = mod (rules)

- **We shall now discuss**
  VAR, VAL, EXPR, STATE, EVAL
- **Assignment for this week**
  STM, THREADS, RUNTIME, SEMANTICS

Statements

```
fmod GCL−STATEMENTS is pr EXPR .

  sorts GuardStm Statement .

  op _▷_ : Expr Statement → GuardStm [ctor prec 95 ] .
  op _◁_▷_ : Statement Expr Statement → GuardStm [ctor prec 95 ] .
  op _;_ : GuardStm GuardStm → GuardStm [ctor assoc prec 100 id:  idle ] .
  op skip : → GuardStm [ctor] .
  op idle : → GuardStm [ctor] .

  op _:=_ : Var Expr → Statement [ctor prec 90] .
  op spawn : GuardStm → Statement [ctor] .
endfm
```

Note: there is a special attribute format to help with nicer output. Example:

```
  op _▷_ : Expr Statement → GuardStm [ctor prec 95 format (r! or o d)] .
```

Threads & RuntimeState

```
fmod THREADS is pr GCL-STATEMENTS .

    sort Thread .
    subsort GuardStm < Thread .
endfm

fmod RUNTIME-SYNTAX is pr THREADS . pr STATE .

    sort RuntimeState .
    *** We can define a RuntimeState as a multiset of Substitutions and Threads
    *** (A valid RuntimeState will only have one Substitution.)
    subsorts Substitution Thread < RuntimeState .
    *** We can use the idle thread as the empty element of RuntimeState
    *** op none : -> RuntimeState [ctor] .
    op __ : RuntimeState RuntimeState -> RuntimeState [ctor assoc comm id: idle].
endfm
```

Runtime

```
mod GCL-SEMANTICS is pr EVAL . pr RUNTIME-SYNTAX .

    var RS : RuntimeState . var x : Var .
    vars e e' : Expr . vars Sigma Sigma' : Substitution .
    vars s1 s2 : Statement . vars g g' : GuardStm .

    rl [Skip] : skip ; g => g .

    crl [Assignment] : Sigma (e |> x :=e' ; g) => Sigma' g
      if eval(Sigma, e) /\ Sigma' :=insert(x, eval(Sigma, e'), Sigma) .

    crl [Spawn] : Sigma (e |> spawn(g') ; g) => Sigma g g' if eval(Sigma, e) .

    crl [Choice1] : Sigma (s1 <| e |> s2 ; g) => RS
      if eval(Sigma, e) /\ (Sigma (e |> s1 ; g)) => RS .

    crl [Choice2] : Sigma (s1 <| e |> s2 ; g) => RS
      if eval(Sigma, not(e)) /\ (Sigma ((not e) |> s2 ; g)) => RS .
endm
```

Some initial states

ops init1 init2 init3 : $\to$ RuntimeState .

eq init1 =('x $\mapsto$1, 'y $\mapsto$2) (true $\rhd$ spawn(true $\rhd$ 'x :='x +1) ; true $\rhd$ spawn(true $\rhd$ 'y :=3)) .

eq init2 =('x $\mapsto$1) (true $\rhd$ spawn(true $\rhd$ 'x :='x +1) ; true $\rhd$ spawn(true $\rhd$ 'x :=3)) .

eq init3 =('x $\mapsto$0, 'y $\mapsto$0, 'flag $\mapsto$true)
(true $\rhd$ spawn(('x =:=0 or 'flag) $\rhd$ 'x :=100 $*$ 'x) ;
true $\rhd$ spawn(('x :='x +2 $\lhd$ 'flag $\rhd$ 'x :='x +−1 ; true $\rhd$ 'flag :=(not 'flag) ;
'x :='x +2 $\lhd$ 'flag $\rhd$ 'x :='x +−1 ; true $\rhd$ 'flag :=(not 'flag) ;
'x :='x +2 $\lhd$ 'flag $\rhd$ 'x :='x +−1 ; true $\rhd$ 'flag :=(not 'flag) ;
'x :='x +2 $\lhd$ 'flag $\rhd$ 'x :='x +−1 ; true $\rhd$ 'flag :=(not 'flag) ;
'x :='x +2 $\lhd$ 'flag $\rhd$ 'x :='x +−1 ; true $\rhd$ 'flag :=(not 'flag) ;
true $\rhd$ 'y :='y +1))) .

Question for tomorrow: How many final states do we have from init3?

Multisets

Recall multisets from last week

Multisets

- Lists with a commutative concat constructor

```
fmod MSET is protecting NAT .
  sorts NeMSet MSet .
  subsort Nat < NeMSet < MSet .

  op none : -> MSet [ctor] .
  op __ : MSet MSet -> MSet [ctor assoc comm id: none] .
  op __ : NeMSet MSet -> NeMSet [ctor ditto] .
  op __ : MSet NeMSet -> NeMSet [ctor ditto] .
endfm
```

A multiset is a natural way of modeling distributed systems

Objects

- An object is usually represented in Maude by a term

  $$< O : C \mid att_1 : val_1, ..., att_n : val_n >$$

  where
  - O is the object identifier of sort Oid
  - C is the class identifier of the object of sort Cid
  - att_1 to att_n are the attributes of the object
  - val_1 to val_n are their current values

Example: Will Rhea Get Married?

- Our young friend Rhea was at some point concerned about her matrimonial status

- To clarify the situation, she developed the following Maude spec.

- A class is a constructor of objects:

  op <_ : Person | age :_, status :_> : Oid Nat Status → Object [**ctor**] .

- Maude's prelude defines a module CONFIGURATION. The module defines a sort Configuration which is basically a multiset with two subsorts Object and Msg.

  The module includes the following definition of an Object constructor:

  op <_:_|_> : Oid Cid AttributeSet → Object [**ctor**] .

Objects: Internal activity

- Here is Rhea as a Maude object:

  $<$ "Rhea" : Person | age : 12, status : single $>$

  Let's specify her together with her friends Uranus, Kronos and Gaia.

  op single : $\rightarrow$ Status [**ctor**] .
  ops engaged married : Oid $\rightarrow$ Status [**ctor**] .
  op age : _ : Nat $\rightarrow$ Attribute [**ctor**].
  op status : _ : Status $\rightarrow$ Attribute [**ctor**].
  op Person : $\rightarrow$ Cid [**ctor**] .
  subsort String $<$ Oid .

- Example: Rule for aging

  var X : String . **var** N : Nat . **var** S : Status .

  crl [birthday] :
  $<$ X : Person | age : N, status : S $>$
  $\Longrightarrow$ $<$ X : Person | age : N +1, status : S $>$ **if** N $<$ 1000 .

Internal activity: Rules with only *one* object.

Object synchronization

rl [engagement] : < X : Person | age : N, status : single >
< X′ : Person | age : N′, status : single >
$\Longrightarrow$ < X : Person | age : N, status : engaged(X′) >
< X′ : Person | age : N′, status : engaged(X) > .

crl [wedding] : < X : Person | status : engaged(X′), age : N >
< X′ : Person | status : engaged(X), age : N′ >
$\Longrightarrow$ < X : Person | status : married(X′), age : N >
< X′ : Person | status : married(X), age : N′ >
if N > 15 and N′ > 15 .

Object synchronization: rules with multiple objects

Our greek population:

op greeks : $\rightarrow$ Configuration .
eq greeks = < "Gaia" : Person | age : 999, status : single >
< "Uranus" : Person | age : 900, status : single >
< "Kronos" : Person | age : 800, status : single >
< "Rhea" : Person | age : 12, status : single > .

Messages: Asynchronous communication (1)

Messages are defined as terms of sort `Msg`:

> **ops** marry? yes no : Oid Oid $\rightarrow$ Msg .
> **op** waitFor : Oid $\rightarrow$ Status .

Example: Send a `marry?` message:

crl [propose] :
$< X$: Person | age : N, status : single $>$
$< Y$: Person | age : M, status : single $>$
$\Longrightarrow$
$< X$: Person | age : N, status : waitFor(Y) $>$
$< Y$: Person | age : M, status : single $>$
marry?(Y, X)
if $N > 15$.

Asynchronous communication: Rule read or react to messages.

This rule *sends* a message from X to Y.

Messages: Asynchronous communication (2)

Two rules can read a `marry?` request:

crl [accept] :
marry?(Y, X)
$<$ Y : Person | age : N, status : single $>$
$\Longrightarrow$
$<$ Y : Person | age : N, status : engaged(X) $>$ yes(X, Y)
if N $>$ 15 .

rl [reject] :
marry?(Y, X) $<$ Y : Person | age : N, status : S $>$
$\Longrightarrow$ $<$ Y : Person | age : N, status : S $>$ no(X, Y) .

These rules *react* to message by changing
their state and sending new messages.

Messages: Asynchronous communication (3)

Rules for accepting the reply:

rl [yes] :
yes(X, Y)
$< X : \mathrm{Person} \mid \mathrm{age} : N, \mathrm{status} : \mathrm{waitFor}(Y) >$
$\Longrightarrow < X : \mathrm{Person} \mid \mathrm{age} : N, \mathrm{status} : \mathrm{engaged}(Y) > .$

rl [no] :
no(X, Y)
$< X : \mathrm{Person} \mid \mathrm{age} : N, \mathrm{status} : \mathrm{waitFor}(Y) >$
$\Longrightarrow < X : \mathrm{Person} \mid \mathrm{age} : N, \mathrm{status} : \mathrm{single} > .$

These rules *react* to message by
consuming the message and changing their state.

With asynchronous communication,
things can happen in-between the messages...

Summary

Modeling parallel & distributed systems

- GCL: Modeling of interleaved concurrency
- Multisets as a representation of system configurations
- Predefined module CONFIGURATION
- Define constructors for the sorts Object and Msg
- Modelling patterns:
  - internal activity,
  - synchronous communication,
  - asynchronous communication

Questions for tomorrow

1. How many final states from init3 in GCL?
2. How can we model multicast and broadcast?