Semantic Interoperation of Dynamic Heterogeneous Architectures

José Meseguer
SRI International

Carolyn Talcott
Stanford University

Part II: A Case Study in Semantic Interoperation of Dynamic Heterogenous Architectures

Seattle EDCS Workshop, July 1997

Motivation I

In the Santa Fe workshop we proposed *rewriting logic* as a promising semantic framework to combine and interoperate the different semantic models of heterogenous and possibly dynmaic ADLs.

```
            GADL

      /       |      \

    ADL1     ADL2    ADL3

     |        |        |

    M1        M2      M3

      \       |      /

          Rewriting
            Logic
```

Motivation II

To show more concretely the naturalness with which executable formal specification in rewriting logic written the Maude language can achieve this semantic interoperation, we have chosen a case study that combines:

- Dynamic/Reflective Pipes and Filters,
- Message Passing and Object Orientation,
- Implicit Invocation

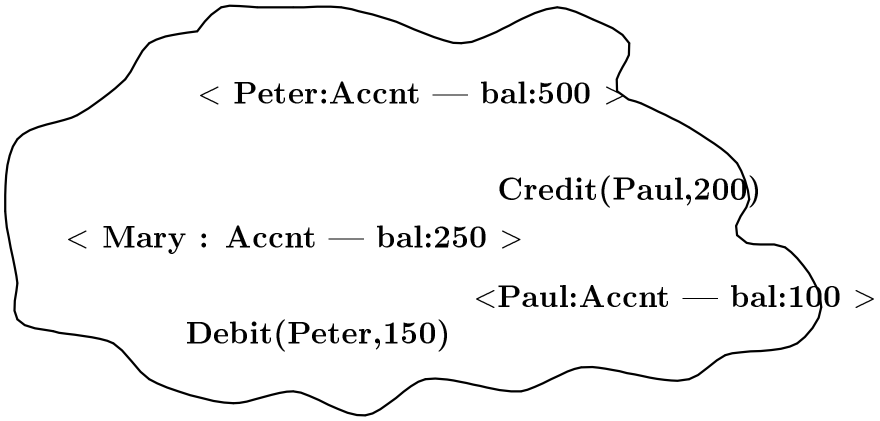
within a single command and control application example

Semantic Interoperability of Heterogenous Architectures

SYSTEM PICTURE goes here

Reminder

Recall that a distributed system of objects and messges is represented as a multiset



That objects have the form

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name, C is its class, $a_i : v_i$ are attributes and associated values.

And that concurrent change in such a system is axiomatized by rewrite rules of the form

$$\begin{aligned}&(M)\langle O_1 : C_1 \mid atts_1\rangle \ldots \langle O_n : C_n \mid atts_n\rangle \\ &\qquad \longrightarrow \langle O'_1 : C'_1 \mid atts'_1\rangle \ldots \langle O'_m : C'_m \mid atts'_m\rangle M_1 \ldots M_k\end{aligned}$$

where some of the O' objects can be new and some O objects may vanish.

Summary of Maude Data Types

The Maude data type modules used in the example are:

- `NAT` (the natural numbers);
- `DEFAULT` (adding a default value to a set);
- `SET` (finite sets);
- `TOSET` (totally ordered sets);
- `MAX` (computing the maximun on a totally ordered sets);
- `LIST` (lists);
- `SIGHTING` (abstract images);

Summary of Maude Object Modules for Case Study

The Maude type modules specific to the example are:

- `MSGS` (the messages exchanged by objects)
- `PIPE` (the pipe object class);
- `FANOUT` (filter that forwards input to set of outputs);
- `RECOGNIZER` (filter that compares images to a model);
- `COLLECTOR` (filter that collects reports from set of recognizers);
- `DISCRIMINATOR` (throws out unreliable reports);
- `IMAGE-RECOGNITION` and `IMAGE-RECOGNITION2` (defines recognition subsystem architecture);
- `META` (used to dynamically install a new recognizer);
- `IMPLICIT-INVOCATION` (rules for implicit invocation);
- `THE-OVERALL-SYSTEM` (defines commander class and overall system architecture);

```
fmod NAT is
  protecting BOOL .
  sorts Nat NzNat Zero .
  subsorts Zero < Nat .
  subsorts NzNat < Nat .
  op 0 : -> Zero .
  ops 1 2 3 4 5 6 7 8 9 10 : -> NzNat .
  ops 11 12 13 14 15 16 17 18 19 20 : -> NzNat .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  op _>_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .
  vars N M : Nat .
  vars N' M' : NzNat .

  eq  p s N = N .
  eq  N + 0 = N .
  eq  (s N) + (s M) = s s (N + M) .
  eq  N * 0 = 0 .
  eq  0 * N = 0 .
  eq  (s N) * (s M) = s (N + (M + (N * M))) .
  eq  0 > M = false .
  eq  N' > 0 = true .
  eq  s N > s M = N > M .
  eq  d(N,0) = N .
  eq  d(0,N) = N .
  eq  d(s N, s M) = d(N,M) .
  eq  max(N,M) = if N > M then N else M fi .
```

```
  eq 1 = s 0 .
  eq 2 = s s 0 .
  eq 3 = s s s 0 .
  eq 4 = s s s s 0 .
  eq 5 = s s s s s 0 .
  eq 6 = s 5 .
  eq 7 = s s 5 .
  eq 8 = s s s 5 .
  eq 9 = s s s s 5 .
  eq 10 = s s s s s 5 .
  eq 11 = s 10 .
  eq 12 = s s 10 .
  eq 13 = s s s 10 .
  eq 14 = s s s s 10 .
  eq 15 = s s s s s 10 .
  eq 16 = s 15 .
  eq 17 = s s 15 .
  eq 18 = s s s 15 .
  eq 19 = s s s s 15 .
  eq 20 = s s s s s 15 .
endfm


fmod DEFAULT[Y :: TRIV] is
  sort Default .
  op null : -> Default .
  subsorts Elt < Default .
endfm
```

```
fmod SET[X :: TRIV] is
  protecting BOOL .
  sorts Set NeSet .
  subsorts Elt < NeSet < Set .
  op mt : -> Set .
  op __ : Set Set -> Set [assoc comm id: mt] .
  op __ : NeSet NeSet -> NeSet [assoc comm id: mt] .
  op _in_ : Elt Set -> Bool .
  vars E E' : Elt . var S : Set .
  eq E E = E .
  eq E in mt = false .
  eq E in (E' S) = (E == E') or (E in S) .
endfm

fth TOSET is
  protecting BOOL .
  sort Elt .
  op _>_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X > X = false .
  eq X > Z = true if (X > Y) and (Y > Z) .
  eq (X > Y) or (X == Y) or (Y > X) = true .
endfth

fmod MAX[T :: TOSET] is
  protecting SET[T] .
  op max : NeSet -> Elt .
  var E : Elt . var S : NeSet .
  eq max(E) = E .
  eq max(E S) = if E > max(S) then E else max(S) fi .
endfm

fmod LIST[X :: TRIV] is
  sort List .  subsorts Elt < List .
  op nil : -> List .
  op _ ; _ : List List -> List [assoc idr: nil] .
endfm
```

```
fmod SIGHTING is protecting NAT .
  sort ShipName .
  sort Image .
  op nil : -> Image .
  op _ ; _ : Nat Image -> Image .
  op listdiff : Image Image -> Nat .
  op size_ : Image -> Nat .
  ops aircraft-carrier oil-tanker cruiser
                      ptboat cruiseship : -> Image .
  ops Enterprise Valdez Vincennes PT109 LoveBoat : -> ShipName .
  vars N M : Nat .  vars L Q : Image .

  eq size(nil) = 0 .
  eq size(N ; L) = N + size(L) .
  eq listdiff(nil,L) = size(L) .
  eq listdiff(L,nil) = size(L) .
  eq listdiff((0 ; L), Q) = listdiff(L,Q) .
  eq listdiff(L, (0 ; Q)) = listdiff(L,Q) .
  eq listdiff((N ; L),(M ; Q)) =
         (if N == M then 0 else 1 fi) + listdiff(L,Q) .
  eq aircraft-carrier = (3 ; 3 ; 3 ; 3 ; 3 ; 4 ; 3 ; 3 ; 3 ; nil) .
  eq oil-tanker = (2 ; 2 ; 2 ; 2 ; 2 ; 2 ; 2 ; 3 ; nil) .
  eq cruiser = (2 ; 2 ; 4 ; 3 ; 3 ; 2 ; nil) .
  eq cruiseship = (2 ; 2 ; 4 ; 2 ; 4 ; 2 ; nil) .
  eq ptboat = (1 ; 1 ; nil) .
endfm
```

SIGHTING PICTURES go here

```
omod MSGS[X :: TRIV] is
  protecting NAT .
  protecting SIGHTING .
  msg to:_(_,_) : OId Elt Nat -> Msg .
  msg score:_(_,_ -> _,_) : OId Elt Nat ShipName Nat -> Msg .
endom
```

```
omod PIPE[X :: TRIV] is
  protecting NAT .
  protecting MSGS[X] .
  protecting LIST[X] .
  class Pipe | in: OId, content: List, out: OId, count: Nat .
  vars P In Out : OId .
  var E : Elt .
  var L : List .
  var N : Nat .

  rl : < P ; Pipe | in: In, content: nil,
                    out: Out, count: N >
       (to: P (E,s(N)))
    => < P ; Pipe | in: In, content: (E ; nil),
                    out: Out, count: s(N) > .

  rl : < P ; Pipe | in: In, content: (E ; nil),
                    out: Out, count: N >
    => < P ; Pipe | in: In, content: nil,
                    out: Out, count: N >
       (to: Out (E,N)) .
endom
```

PIPE PICTURES go here

```
omod FANOUT[X :: TRIV] is
  protecting NAT .
  protecting MSGS[X] .
  protecting LIST[X] .
  sort Pipes .
  op null : -> Pipes .
  op [_ _] : OId Pipes -> Pipes .
  class Fanout | in: OId, content: List, out-pipes2: Pipes,
                 out-pipes: Pipes, count: Nat .
  vars F In O : OId .  var Out : Pipes .
  vars E, X, Y : Elt .  var B : Bool .
  var S : Pipes .  var N : Nat .

  rl : < F ; Fanout | in: In, content: nil, out-pipes2: null,
                      out-pipes: Out, count: N >
       (to: F (E,s(N)))
    => < F ; Fanout | in: In, content: (E ; nil), out-pipes2: Out,
                      out-pipes: Out, count: s(N) > .

  rl : < F ; Fanout | in: In, content: (E ; nil), out-pipes2: [O Out],
                      out-pipes: Out, count: N >
    => < F ; Fanout | in: In, content: (E ; nil), out-pipes2: Out,
                      out-pipes: Out, count: N >
       (to: F (E,s(N))) .

  rl : < F ; Fanout | in: In, content: (E ; nil), out-pipes2: null,
                      out-pipes: Out, count: N >
    => < F ; Fanout | in: In, content: nil, out-pipes2: null,
                      out-pipes: Out, count: N >
       (to: F (E,s(N))) .
endom
```

FANOUT PICTURES go here

```
omod RECOGNIZER[X :: TRIV] is
  protecting NAT .
  protecting SIGHTING .
  protecting MSGS[X] .
  protecting LIST[X] .
  class Recognizer | in: OId, content: List, out: OId,
                     count: Nat, model: Elt, name: ShipName .
  vars R In Out : OId .
  var E : Elt .
  var N : Nat .
  var Im : Elt .
  var SN : ShipName .

  rl : < R ; Recognizer | in: In, content: nil, out: Out,
                          count: N, model: Im, name: SN >
       (to: R (E,s(N)))
    => < R ; Recognizer | in: In, content: (E ; nil), out: Out,
                          count: s(N), model: Im, name: SN > .

  rl : < R ; Recognizer | in: In, content: (E ; nil), out: Out,
                          count: N, model: Im, name: SN >
    => < R ; Recognizer | in: In, content: nil, out: Out,
                          count: N, model: Im, name: SN >
       (score: Out (E,N -> SN, 0)) . --- listdiff(E,Im)
endom
```

```
omod COLLECTOR[X :: TRIV] is
  protecting NAT .
  protecting SIGHTING .
  protecting MSGS[X] .
  protecting LIST[X] .
  sort Pipes .
  op null : -> Pipes .
  op [_ _] : OId Pipes -> Pipes .
  class Collector | in-pipes: Pipes, in-pipes2: Pipes,
                    content: List, out: OId, count: Nat .
  vars P Pip Out : OId .
  var In Pi : Pipes .
  var E : Elt .
  var L : List .
  vars N M : Nat .
  var SN : ShipName .

  rl : < P ; Collector | in-pipes: In, in-pipes2: [Pip Pi],
                         content: nil, out: Out, count: N >
       (score: P (E, s(N) -> SN, M))
    => < P ; Collector | in-pipes: In, in-pipes2: Pi,
                         content: nil, out: Out, count: N >
       (score: P (E, s(N) -> SN, M)) .

  rl : < P ; Collector | in-pipes: In, in-pipes2: null,
                         content: nil, out: Out, count: N >
       (score: P (E, s(N) -> SN, M))
    => < P ; Collector | in-pipes: In, in-pipes2: In,
                         content: nil, out: Out, count: s(N) > .
endom
```

```
omod DISCRIMINATOR[X :: TRIV] is
  protecting NAT .
  protecting SIGHTING .
  protecting MSGS[X] .
  protecting LIST[X] .
  class Discriminator | in: OId, content: List, out: OId,
                        count: Nat, threshold: Nat .
  protecting MSGS[X] .
  vars P In Out : OId .  var E : Elt .
  var L : List .  vars T N M M2 X Y : Nat .
  vars SN2 SN : ShipName .

  crl : < P ; Discriminator | in: In, content: nil, out: Out,
                              count: Y, threshold: T >
       (score: P (E,N -> SN,M))  (score: P (E,N -> SN2,M2))
    => < P ; Discriminator | in: In, content: nil, out: Out,
                             count: Y, threshold: T >
       (score: P (E,N -> SN,M))
    if (M2 > M or M2 == M) .

  crl : < P ; Discriminator | in: In, content: nil, out: Out,
                              count: N, threshold: T >
       (score: P (E,X -> SN,M))
    => < P ; Discriminator | in: In, content: (E ; nil), out: Out,
                             count: s(N), threshold: T >
    if (M > T) .  --- Throw report away

  crl : < P ; Discriminator | in: In, content: nil, out: Out,
                              count: N, threshold: T >
       (score: P (E,X -> SN,M))
    => < P ; Discriminator | in: In, content: (E ; nil), out: Out,
                             count: s(N), threshold: T >
       (score: Out (E,X -> SN,M))
    if (T > M or T == M) .  --- We found an SN, report it
endom
```

```
smod IMAGE-RECOGNITION is
  protecting PIPE[SIGHTING] .
  ops r1 r2 : -> OId .
  ops gen1 lt1 lt2 lt3 : -> Object .
  eq  gen1 = < r1 ; Pipe | in: r1, content: nil,
                           out: r1, count: 0 >  .
endsm


omod IMAGE-RECOGNITION2[X :: TRIV] is
  protecting MSGS[X] .
  protecting PIPE[SIGHTING] .
  protecting FANOUT[SIGHTING] .
  protecting RECOGNIZER[SIGHTING] .
  protecting COLLECTOR[SIGHTING] .
  protecting DISCRIMINATOR[SIGHTING] .
  op init-conf1 : -> Configuration .

  ops info-source1 fanout1 r1 r2 r3 r4 p1 p2 p3 p4
                           fanin1 discriminator1 : -> OId .
  op inf1 : -> Object .



--- Design: start with pipe with incoming sensor data
--- that is fanouted to a list of pipes
--- each of those pipes has a special recognizer
--- of each type of boat
--- then a fanin from those pipes to a single pipe
--- then a discriminator (threshold filter)
```

```
eq init-conf1 =
 (to: info-source1 ((2 ; 2 ; 4 ; 4 ; 3 ; 2 ; 0 ; 0 ; 0), 0))
 (to: info-source1 ((0 ; 0 ; 0 ; 2 ; 1 ; 0 ; 0 ; 0 ; 0), 1))
 (to: info-source1 ((3 ; 3 ; 3 ; 3 ; 3 ; 5 ; 4 ; 3 ; 3), 2))
 (to: info-source1 ((2 ; 2 ; 2 ; 2 ; 2 ; 2 ; 2 ; 3 ; 0), 3))
 < info-source1 ; Pipe | in: info-source1, content: nil,
                         out: fanout1, count: 0 >
 < fanout1 ; Fanout | in: info-source1, content: nil,
                      out-pipes2: nil,
                      out-pipes: (p1 ; (p2 ; (p3 ; nil))), count: 0 >
 < p1 ; Pipe | from: fanout1, content: nil, to: r1, count: 0 >
 < p2 ; Pipe | from: fanout1, content: nil, to: r2, count: 0 >
 < p3 ; Pipe | from: fanout1, content: nil, to: r3, count: 0 >
 < r1 ; Recognizer | in: p1, content: nil, out: fanin1, count: 0,
                     model: destroyer, name: Vincennes >
 < r2 ; Recognizer | in: p2, content: nil, out: fanin1, count: 0,
                     model: aircraft-carrier, name: Enterprise >
 < r3 ; Recognizer | in: p3, content: nil, out: fanin1, count: 0,
                     model: ptboat, name: PT109 >
 < fanin1 ; Collector | in-pipes: (r1 ; (r2 ; (r3 ; nil))),
                        in-pipes2: (r1 ; (r2 ; (r3 ; nil))),
                        content: nil, out: discriminator1, count: 0 >
 < discriminator1 ; Discriminator | in: fanin1, content: nil,
                                out: USER, count: 0, threshold: 3 >  .

---  eq conf2 =
---  (init-conf (to o(13) install-recognizer oil-tanker ot)) .
endom
```

```
omod META is
  extending RECOGNIZER DISCRIMINATOR .
  extending FANOUT[Set[Data]]*(sort Ports to Sports) .
  protecting MAX[NAT] .
  protecting DEFAULT[OId]*(sort Default to DftOId) .
  protecting DEFAULT[Nat]*(sort Default to DftNat) .
  sort State .
  op lock : ClId -> ClId .
  op (_._) : OId Nat -> OId .
  ops ready busy : -> State .
  msg to_install-recognizer_ot : OId Image -> Msg .
  class Meta | in-pipe: DftOId, out-pipe: DftOId, fanout: DftOId,
               recognizer: DftOId, collector: DftOId,
               state: State, cnt: DftNat, tag: Nat .
  vars M P P' R F C : OId .
  var I : Image .
  vars N N' : Nat .
  vars S S' : Ports .
  var ATTS : Atts .


  *** A meta-object can, upon request, dynamically change
  *** the configuration of the dataflow network it controls,
  *** by adding a new recognizer object to recognized a given
  *** image and two new pipes and by hooking it up through
  *** the pipes to the fanout and collector objects.
```

```
  rl [ create ] : (to M install-recognizer I ot)
         < M : Meta | state: ready, tag: N >
     =>  < M : Meta | state: busy, recognizer: M.N,
                      in-pipe: M . (s N),
                      out-pipe: M . (s s N), tag: s s s N >
         < M . N : Recognizer | in: < 1 ; null >, out: < 1 ; null >,
                                model: I >
         < M . (s N) : Pipe | from: null, to: (M . N), q: nil >
         < M . (s s N) : Pipe | from: (M . N), to: null, q: nil > .

  rl [ hook-fanout ] : < M : Meta | state: busy, recognizer: R,
                                    in-pipe: P, fanout: F, cnt: null >
        < F : Fanout | cnt: N', out: S >
        < P : Pipe | from: null >
     => < M : Meta | recognizer: null, in-pipe: null, cnt: N' >
        < F : lock(Fanout) | out: (S < s max(dom(S)) ; null) > >
        < P : Pipe | from: < F ; s max(dom(S)) > > if empty(S) .

  rl [ hook-collector ] : < M : Meta | in-pipe: null, out-pipe: P',
                                       cnt: N', collector: C,
                                       fanout: F >
         < C : Collector | cnt: N', in: S' >
         < P' : Pipe | to: null >
         < F : lock(Fanout) | ATTS >
     =>  < M : Meta | out-pipe: null, cnt: null, state: ready >
         < C : Collector | in: (S' < s max(dom(S')) ; null >) >
         < P' : Pipe | to: < C ; s max(dom(S')) > >
         < F : Fanout | ATTS > if empty(S') .
endom
```

PICTURES OF META WORKING go here

IMPLICIT BUBBLE PICTURE goes here

```
omod IMPLICIT-INVOCATION[ M :: TRIV*(Elt to MsgId),
                          E :: Triv*(Elt to Event),
                          P :: TRIV*(Elt to Par)] is
  protecting PFUN[Event,MsgId]*(sort Pfun to Table) .
  class Bubble | conf: Configuration .
  class Implicit | table: Table .
  msg to_msg_with_ot : OId MsgId Par -> Msg .
  op bc_with_in_cb : Event Par Configuration -> Configuration .
  subsort ExtAct < Msg .
  vars B O : OId .  var M : MsgId .
  var E : Event .  var P : Par .
  var T : Pfun .  vars C C' : NeConfiguration .
  var MSG : Msg .  var EA : ExtAct .

  rl [bc1] : bc E with P in C C' cb
          => (bc E with P in C cb) (bc E with P in C' cb) .

  rl [bc2] : bc E with P in null cb => null .

  rl [bc3] : bc E with P in MSG cb => MSG .

  rl [bc4] : bc E with P in < O : Implicit | table: T > cb
          => if T [ E ] == undef then < O : Implicit | table: T >
             else to O msg T [ E ] with P ot
                  < O : Implicit | table: T >
             fi .

  rl [out] : < B : Bubble | conf: (EA C) >
          => < B : Bubble | conf: C > EA .
endom
```

```
omod OVERALL-SYSTEM is
  extending IMAGE-RECOGNITION .
  sorts Event MsgId .
  op sghtng : -> Event .
  sort air-act bttlshp-act -> MsgId .
  extending IMPLICIT-INVOCATION[MsgId,Event,
             PAIR[Location,Evals]]*(sort Pair to Info,
                                   sort Bubble to Controller) .
  op to_threat-at_ot : OId Location -> ExtAct .
  op to_recon-at_to : OId Location -> ExtAct .
  op q : Nat -> OId .
  class Commander | table: Table, subordinate: OId .
  subclass Commander < Implicit .
  vars O Q S : OId .  var L : Location .
  var EVS : Evals .  var C : Configuration .

  rl [control] : (to O at L evals EVS ot)
        < O : Controller | conf: C >
     => < O : Controller | conf: bc sghtng with < L ; EVS > in C cb > .

  rl [bttlshp-act] : (to Q msg bttlshp-act with < L ; EVS > ot)
        < O : Commander | subordinate: S >
     => < O : Commander | subordinate: S > (to S threat-at L ot) .

  rl [air-act] : (to Q msg air-act with < L ; EVS > ot)
        < O : Commander | subordinate: S >
     => < O : Commander | subordinate: S > (to S recon-at L ot) .
```

```
  op syst-conf : -> Configuration .
  eq syst-conf =
     (init-conf < o(s s s 10) : Controller | conf:
     (< q(0) : Commander | table: < sghtng ; air-act >,
                           subordinate: q(1) >
      < q(2) : Commander | table: < sghtng ; air-act >,
                          subordinate: q(3) >) >) .
endom



--- rw init-conf .

--- rw (init-conf (to o(s s s 10) install-recognizer oil-tanker ot)) .

--- rw syst-conf .
```

Conclusions

We have found rewriting logic very useful in the task of combining at the semantic level hetergeneous and dynamic architectures.

- the specifications are clear and succinct
- modularity and genericity allow scalable and reusable solutions
- the specifications are executable in Maude

The following future directions seem quite promising

- Study semantics of composition at more abstract levels
- Further develop the reflective semantics of evolution and adaptation
- Interoperation of existing ADLs
- Formal reasoning about and analysis of architectures
- Formally based tools