Specifying the Plan Language in Maude

José Meseguer & Mark-Oliver Stehr, SRI International

Carolyn Talcott, Stanford University

June 2000

PLAN -:)

- Background
  - What is PLAN
  - Motivation
  - Challenges
- Rewriting Logic and Maude
  a brief review
- The specification
  - Highlights of the Design
  - Example PLAN program
  - Disection
- Future

What is PLAN

- a language for programming mobile agents
  concerns of security

- a language for programming networks
  concerns of resource usage

- a domain specific language
  for packet-level active network programming
  simple, special purpose

- a scripting language for combining node services
  needs rely/guarantee style reasoning

PLAN features

- a PLAN program is a set of packets
  executing in the network
- almost functional [think ML]
  - with the ususal basic data types
  - plus network specific types
  - chunks not thunks
- strongly typed
- resource limited
- primitives for sending packets
- service packages for interacting with nodes

Customers – Who Benefits

- application programmers
  - clarifying the language constructs
  - simulation for testing and analysis

    viz. resource measurement
  - enhance portability of PLAN programs
- implementors
  - a clear and flexible specification
  - a reference model for testing
  - enable common tutorials and test suites

Customers – Who Benefits

- language designers
  - a working document
  - means to communicate PLAN design ideas
  - a tool for experimenting with language modifications
  - a basis for
    * specifying safety properties of the service layer functions
    * expressing compiler/platform independent guarantees for network protection
    * analysis, clear statements of properties, and formal verification of PLAN programs

Challenges

- network model – getting the right level of abstraction

- conformance flexibility
  - anytime typechecking
  - range of possible effects

- Pleasing many customers: programmers, designers, implementors ...

History

- The first step was to specify the FBAR algorithm in Maude
- This went well and we were able to do some analysis: an instance verification for a small network
- But this was a *hand* translation of a PLAN program into Maude
- The is a paper specification, but
  - not executable
  - gaps, inconsistencies, and ambiguities typical of semi-formal (on paper) operational semantics

Rewriting logic – Syntax

- RWL = EQL + rewrite rules to model concurrent action
- Signature: (Σ, E) – an equational theory
- Rewriting operates on equivalence classes of terms $[t] \in [T_\Sigma]_E$
- Rewrite theory: $\mathcal{R} = (\Sigma, E, L, R)$
  - (Σ, E) is a signature,
  - R – rewrite rules with labels from L

$$r : \; [t] \longrightarrow [t'] \;\; \texttt{if} \;\; cond$$

- $\mathcal{R} \vdash t \longrightarrow t'$
  if $t \longrightarrow t'$ is derivable using the deduction rules.

The deduction rules of rewriting logic

For $[t] \in T_{\Sigma,E}(X)$:

1. **Reflexivity**. $[t] \longrightarrow [t]$

2. **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}$$

3. **Replacement**. For each rewrite rule

$$r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)] \ \text{ in } \ R,$$

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

4. **Transitivity**.

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

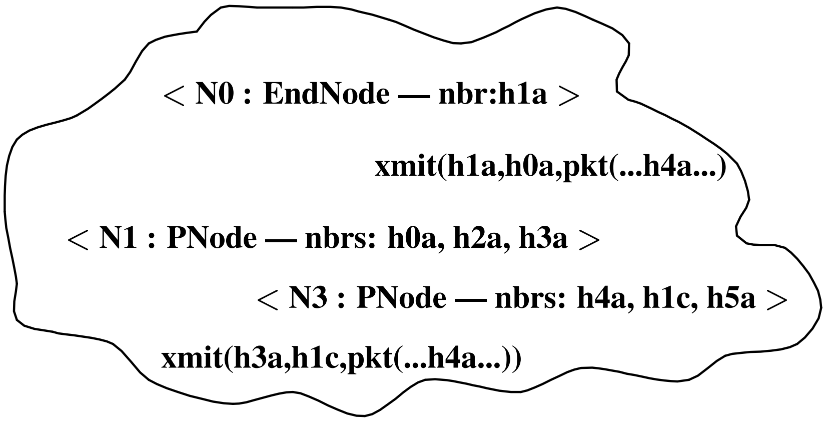
Concurrent Objects

An *object* 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, the a_i's are the names of the object's attributes, and the v_i's are their corresponding values.

A *configuration* is a multiset of objects and messages formed by a multiset union operation "_ _" that is *associative* and *commutative*



Object-Oriented Rules

A system evolves by concurrent *ACI*-rewriting of its configuration according to rules of the general form:

$$
\begin{aligned}
& M_1 \ldots M_n \ \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\
& \qquad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
& \qquad M'_1 \ldots M'_q \\
& \mathit{if}\ C
\end{aligned}
$$

By specializing to patterns involving only one object and one message, we can obtain an abstract, declarative, and truly concurrent model of asynchronous interaction in distributed systems.

Developing the Maude Specification I

Preparation

- Listening to some talks about PLAN and asking questions
- Reading
  - PLAN Standard Specification and associated paper
  - PLAN Programmers Guide and Tutorial
- The PLAN documents (and some knowledge of semantics of functional languages and networks) were enough to produce a first Maude specification.
- Coding of first version accomplished in about 1 week by 2 people working about half-time.

Using Maude Features

- The network infrastructure is represented in Maude as a multiset of objects (PLAN nodes) and packets (in transit).

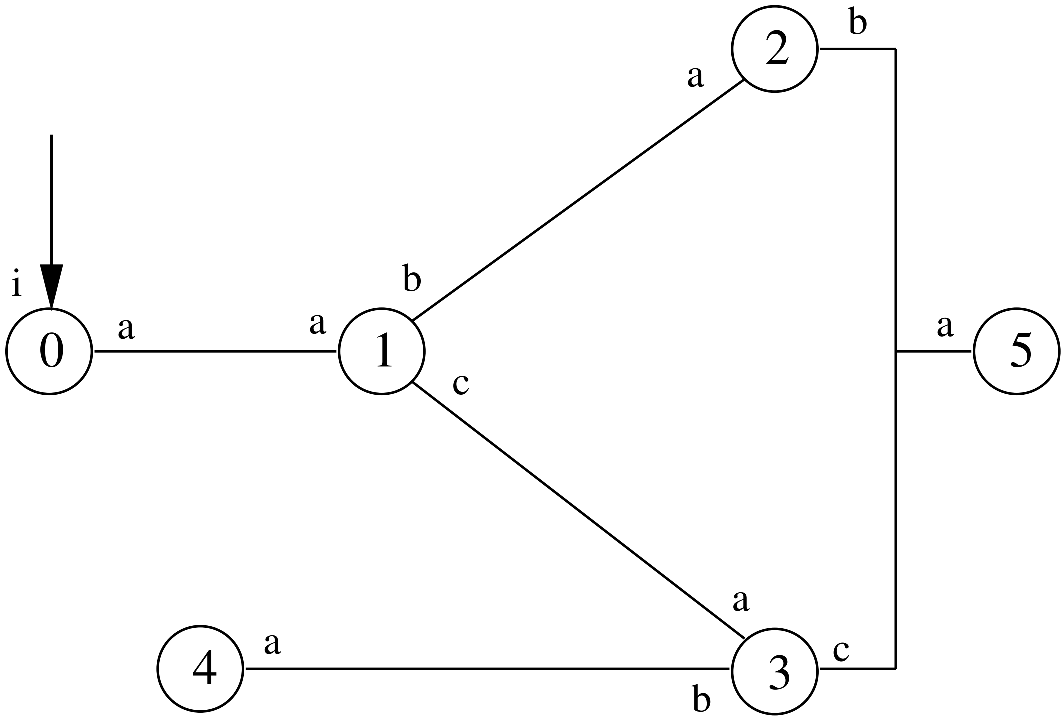
- Service packages modelled by subclassing nodes

- concurrency modelled by multiset rewriting
  - at the configuration level – concurrent activity among nodes
  - at the node level – packets executing concurrently

Design of the PLAN Interpreter

- 3 level semantics: local, task, node
- rules for local steps are purely functional
  - standard techniques for operational semantics
- rules at the task level use resource information carried in the packet
  - factored out to simplify the rules
- rules at the node level use node state and implement PLAN services
  - includes routing

A PLAN Network



Representation of the PLAN Network in Maude

```
 eq conf = inject(node0,pgm(topdef),
                  find, (host(h5a),emptylst,emptylst), 100)

< node0 : PlanEndNode | output : emptyTVList,
  interfaces : (d0i,d0a), addrs : (h0i,h0a),
  neighbors : (nbr(h0i,d0i),nbr(h1a,d0a)),
  me : h0i, port : d0i,
  tasks : noTask >

< node1 : PlanNode | output : emptyTVList,
  interfaces : (d1a,d1b,d1c), addrs : (h1a,h1b,h1c),
  neighbors : (nbr(h0a,d1a),nbr(h2a,d1b),nbr(h3a,d1b)),
  tasks : noTask >

< node2 : PlanNode | output : emptyTVList,
  interfaces : (d2a,d2b), addrs : (h2a,h2b),
  neighbors : (nbr(h1b,d2a),nbr(h3c,d2b),nbr(h5a,d2b)),
  tasks : noTask >

< node3 : PlanNode | output : emptyTVList,
  interfaces : (d3a,d3b,d3c), addrs : (h3a,h3b,h3c),
  neighbors : (nbr(h1c,d3a),nbr(h4a,d3b),
               nbr(h2b,d3c),nbr(h5a,d3c)),
  tasks : noTask >

...
```

Packet Generation Rule

```
crl [netw] :
   < xOid : PlanNode |
     interfaces : ( xDevL0, xDev, xDevL1 ),
     addrs : xHostL,
     tasks : xTasks
       task(lstate(next, env,
             appc((tv(xChunk, chunk), tv(hostV(xHost),host),
                   tv(intV(xInt),int), tv(devV(xDev),dev)),
                  emptyExpList, OnNbrOp, cont)),
            rstate(xInt',rf,xDev',xHost',xPgm)) >
  =>
   < xOid : PlanNode |
     tasks : xTasks
       task(lstate(val(tv(unitV,unit)), env, cont),
            rstate(xInt' - xInt, rf, xDev',xHost',xPgm)) >

   xmit(xHost, xDev,
        pkt(xChunk, xHost, xInt, nullRf,
            nth(xHostL,length(xDevL0))))

   if ( not ( xInt' < xInt ) ) .
```

Packet Delivery Rule

```
rl [netw] :
   < xOid : PlanNode |
     interfaces : xDevL,
     addrs : (xHostL0,xHost,xHostL1),
     stdenv : env,
     tasks : xTasks >

   xmit(xHost,xDev, pkt(xChunk,xHost,rb,rf,src))

  =>

   < xOid : PlanNode |
     tasks : xTasks
             newTask(src, xHost, env, xChunk,rb,rf) > .
```

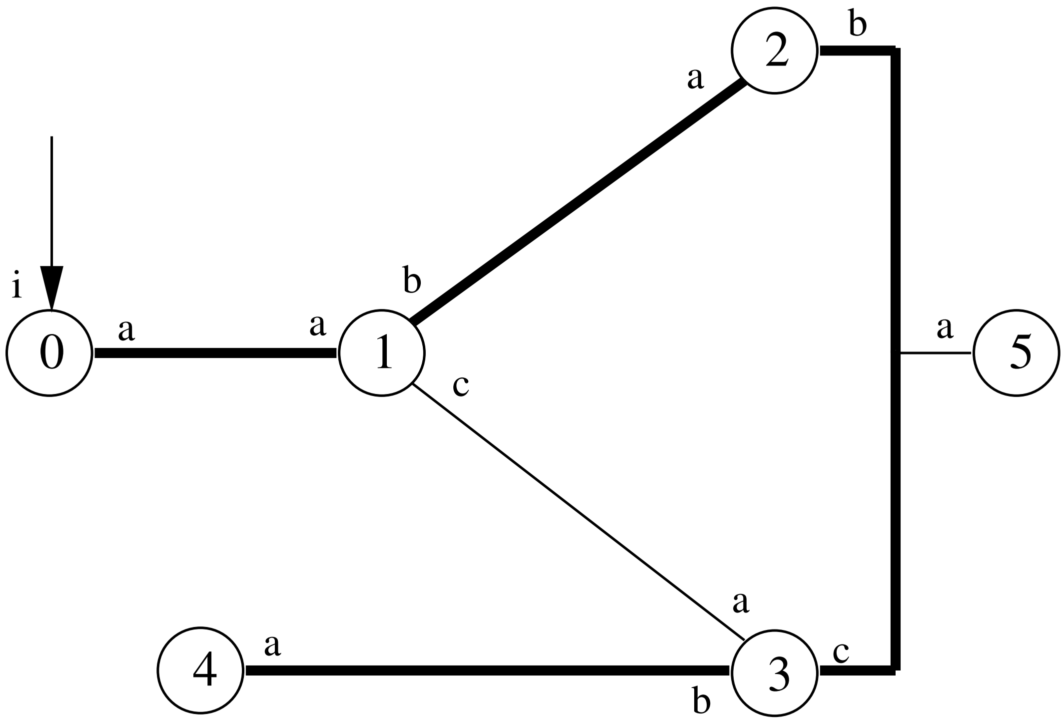
Representation of a PLAN Program in Maude

The distributed route-finding example can be directly represented and executed in Maude. Some errors in the paper version of the original program have been detected by dynamic typechecking and have been corrected.

```
eq findfun =

fun find [ (dest : host),
           (route : (host * host) list),
           (visited : host list) ] : unit =
 let
   (val neighbors : (host * dev) list =
     getNeighbors[emptyExpList]),
   ....
   (fun sendChild [(n : host * dev), (u : unit) ] : unit =
        OnNeighbor[
        | find |[var(dest),
                 cons[tup[thisHostOf[var(srcdev)],
                          thisHostOf[snd[var(n)]]],var(route)],
                 append[var(myaddrs),var(visited)]],
        fst[var(n)],
        var(childRB),
        snd[var(n)]])
 in
   (if thisHostIs[var(dest)]
    then goback[var(route),reverse[var(route)]]
    else foldr[var(sendChild), var(newNeighbors), unitC])
 end .
```

A Computed Route from Node 0 to 4



Type Flexibility

- Informal requirement: anytime typechecking
  - a node may typecheck statically, dynamically, ...
  - typechecking can be node-specific
- Formally captured in Maude
  - by rules that
    - catch ill-typed operations in the execution
    - statically type check programs at injection time
    - type-check execution states
  - combined with use of meta level strategies capturing implementation level design — a node conforms if it raises exceptions according to some strategy for applying the rules.

Future Evolution

The goal is a living specification that evolves with PLAN and that can be used as a simulator and program analysis tool, and can play with other artifacts.

- PLAN is an evolving language
  - PLAN 3.1 – the paper specifiation
  - PLAN 3.2 – current implementation with polymorphic types
  - PLAN 3.3 – being designed to add cryptographic reflection

Possible Role of Maude in PLAN 3.3

- Experiment with alternative designs
- Execute test cases for multiple nodes
- Ask questions like “who got to see this procedure?”
- Analyze information flow and secrecy requirements