

Pathway Logic Tutorial

Carolyn Talcott
SRI International

September 2005

Plan

Part I : Maude

- Lecture
- break
- Exercises
- <http://maude.cs.uiuc.edu/>

Lunch

Part II: PLA

- Lecture
- break
- Demo/Exercises
- <http://www.csl.sri.com/~clt/PLweb/>
- <http://www.csl.sri.com/~clt/PLweb/install-notes.html>

Pathway Logic Tutorial
Part I
Using Maude to Model and Analyze
Current Systems

Carolyn Talcott
SRI International

<http://maude.cs.uiuc.edu/>

Maude

- o Maude is a language and environment based on rewriting logic
- o See: <http://maude.cs.uiuc.edu>
- o Features:
 - Executability -- position /rule/object fair rewriting
 - High performance engine --- {ACI} matching
 - Modularity and parameterization
 - Builtins -- booleans, number hierarchy, strings
 - Reflection -- using descent and ascent functions
 - Search and model-checking

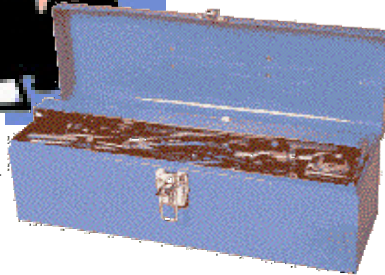
Maude Formal Methodology



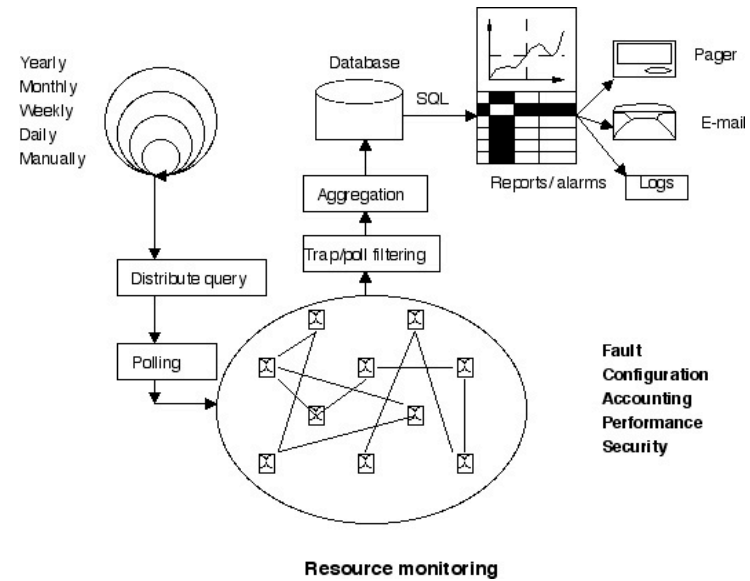
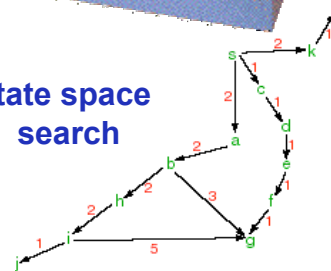
model building



rapid
prototyping



state space
search



$S \models \Phi$

model
checking

$S \vdash \Phi$

theorem
proving

Topics

About rewriting logic

Specifying data types

Specifying dynamics

Analysis (search and model-checking)

Reflection

Rewriting Logic

Q: What is rewriting logic?

A1: A logic for executable specification and analysis of software systems, that may be concurrent, distributed, or even mobile.

A2: A logic to specify other logics or languages

A3: An extension of equational logic with local rewrite rules to express

1. concurrent change over time
2. inference rules

What Rewriting Logic Is/IsNot

- 0 A rewrite theory plus a term describes a state transition system
 - states can have rich algebraic structure
 - transitions are local and possibly concurrent
- 0 The equational part of a rewrite theory is similar to a term rewriting system (modulo ACI axioms)
 - it is usually desirable for equations to be CR and terminating
 - rewrite rules are often non-deterministic and non-terminating

Rewriting Logic as a Semantic Framework

A wide variety of models of computation can be naturally expressed as rewrite theories

- o Lambda Calculus, Turing Machines
- o Concurrent Objects and Actors
- o CCS, CSP, pi-calculus, Petri Nets
- o Chemical Abstract Machine, Unity
- o Graph rewriting, Dataflow, Neural Networks
- o Real-time Systems
- o Physical Systems (Biological processes)

Rewriting Logic as a Logical Framework

A wide variety of logics have a natural representation as rewrite theories

- o Equational logic
- o Horn logic
- o Linear logic
- o Quantifiers
- o Higher-order logics (HOL, NuPrl)
- o Open Calculus of Constructions
- o Rewriting logic !

Rewriting Logic is Reflective

A reflective logic is a logic in which important aspects of its metatheory (entailment relation, theories, proofs) can be represented at the object level in a consistent way.

This has many applications:

- Transforming, combining rewrite theories
- Execution / proof strategies
- Meta tools: theorem provers, coherence checkers ...
- Language extensions: object-oriented, real-time, ...
- Higher-order capabilities in a first-order framework
- Model of reflection for concurrent objects
- Domain specific assistants

Rewrite Theories

- 0 Rewrite theory: (Signature, Labels, Rules)
- 0 Signature: (Sorts, Ops, Eqns) -- an equational theory
 - Describe data types, system state
- 0 Rules have the form $\text{label} : t \Rightarrow t' \text{ if cond}$
- 0 Rewriting operates modulo equations
 - Describes computations or deductions, also modulo equations

Maude System Specifications

A Maude specification has two parts

- o An equational part describing structure and properties of system states (and ADT)
- o A rules part specifying how the system might change over time.

ADTs (Abstract Data Types)

An ADT consists of one or more sets of values, together with a set of operations

$$(D_1 \dots D_k, c_1 \dots c_m, f_1 \dots f_n)$$

- o Values are given by constructors: $c(t_1, \dots, t_l)$
- o Functions $f_1 \dots f_n$ are defined by mathematical relations (usually equations).

ADTs in Maude

ADTs are specified in Maude using functional modules

```
fmod <modname> is
  <imports>          *** reuse, modularity
  <sorts>            *** data types and subtyping
  <opdecls>          *** names/and arities of operations
  <eqns>             *** how to compute functions
endfm
```

- 0 The module components can appear interleaved and in any order
- 0 The semantics of an fmod is an initial model
 - no junk---every data value is constructable
 - no confusion---two terms are equal only forced by the equations

Operator declarations

The <opdecls> part of a module consists of operator declarations having the form

```
op <opname> : <argSorts> -> <resultSort> [<attributes>] .
```

opname can be an identifier (without special characters) or identifiers interleave with `_'s (mixfix notation).

<attributes> for binary operators include assoc, comm, id: <term>

Including assoc declares the operator to be associative, ...

```
*** unary constructor
```

```
  op s_ : Nat -> Nat .
```

```
*** unary operator
```

```
  op sub1 : Nat -> Nat .
```

```
*** associative commutative infix operator with identity
```

```
  op _+_ : Nat -> Nat [assoc comm id: 0] .
```


The NAT and NATLIST Data Types

The natural numbers are specified by

```
fmod NAT is
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  ....
endfm
```

Examples: 0, s s s 0 (also printed as 3)

Lists of natural numbers are specified by

```
fmod NATLIST is
  pr NAT .
  sort NatList .  subsort Nat < NatList .
  op nil : -> NatList .
  op _ _ : NatList NatList -> NatList [assoc id: nil] .
  ....
endfm
```

Example lists: nil, 1 2 3

Defining NatList Functions I

Definition by pattern matching (extending NATLIST)

```
var n: Nat. var nl: NatList .  
op head : NatList ~> Nat .  
op tail : NatList -> NatList .  
eq head(n nl) = n .  
eq tail(n nl) = nl .    eq tail(nil) = nil .
```

Consider the list : 1 2 3

It matches the pattern (n nl) with $n := 1$, $nl := 2\ 3$

```
Maude> reduce head(1 2 3) .  
result NzNat: 1
```

```
Maude> red tail(1 2 3) .  
result NatList: 2 3
```

```
Maude> red head(nil) .  
result [Nat]: head(nil)
```

```
Maude> red tail(nil) .  
result NatList: nil
```

Defining NatList Functions II

Definition by recursion

Define a function `sum` that adds the elements of a NatList.

```
var n : Nat. var nl : NatList .  
op sum : NatList -> Nat .  
eq sum(nil) = 0 .  
eq sum(n nl) = n + sum(nl) .
```

The equations are used to reduce terms to canonical form

$$\text{sum}(1\ 2\ 3) = 1 + \text{sum}(2\ 3) = 1 + 2 + \text{sum}(3) = 1 + 2 + 3 = 6$$

```
Maude> reduce sum(1 2 3) .  
result NzNat: 6
```

Defining NatList Functions III

Conditional equations

Define a function `isElt` that takes a Nat and a NatList and returns true if the Nat is in the NatList and false otherwise.

```
vars n n' : Nat.    var nl : NatList .  
op isElt : Nat NatList -> Bool .  
eq isElt(n, n' nl) =  
    if n == n' then true else isElt(n,nl) fi .  
eq isElt(n, nil) = false .
```

Examples:

```
isElt(4, 4 6 3) = true  
isElt(4, 7 2 4 6) = isElt(4,2 4 6)  
isElt(4,nil) = false
```

Defining NatList Functions IV

Alternate definition of `isElt`: via AC pattern matching

```
vars n : Nat .    vars nl nl' : NatList .  
op isElt : Nat NatList -> Bool .  
eq isElt(n, nl n nl') = true .  
eq isElt(n, nl) = false [owise] .
```

Examples:

`isElt(4, 7 2 4 6 3)` matches the 1st equation

taking `n := 4, nl' := 7 2, nl := 6 3`

and so reduces to `true`

`isElt(4, 7 2 3)` does not match the 1st equation, so
reduces to `false` by the [owise] equation

Caveats

Caveats for writing Maude specifications:

- 0 Don't forget the period at the end of a declaration or statement! .
- 0 Check keywords / symbols are correct.
 - correct use of fmod, mod
 - op vs ops
 - variables not used before bound
- 0 Make sure all cases are covered in defining equations.

Specifying behavior/dynamics

System dynamics are specified in system modules using rewrite rules

```
mod <modname> is
  *** functional part
    <imports>      *** reuse, modularity
    <sorts>        *** data types and subtyping
    <opdecls>      *** names/and arities of operations
    <eqns>         *** how to compute functions
  ***
  <rules>
endm
```

A system module defines a set of computations (aka derivations) over the ADT specified by the functional part.

Rule declarations

The <rules> part of a system module consists of rule declarations having one of the forms

`rl[<id>]: <lhs> => <rhs> .`

`crl[<id>]: <lhs> => <rhs> if <cond> .`

<lhs>, <rhs>, <cond> are terms, possibly containing variables.

A rule applies to a term T if there is a substitution S (mapping variables to terms) such that $S<lhs>$ is a subterm of T ($<lhs>$ matches a subterm of T) and $S<cond>$ rewrites to `true`.

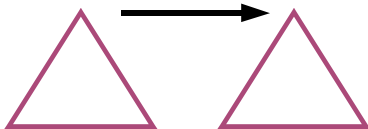
In this case T can be rewritten by replacing the matched subterm by the matching instance of $<rhs>$ ($S<rhs>$).

Deduction/Computation Rules

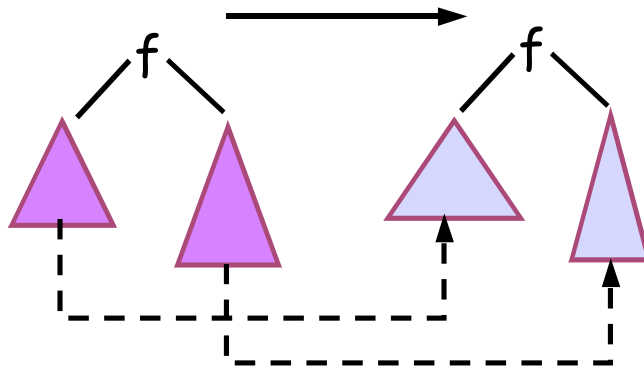
one step rewrite:  \longrightarrow 

closed under

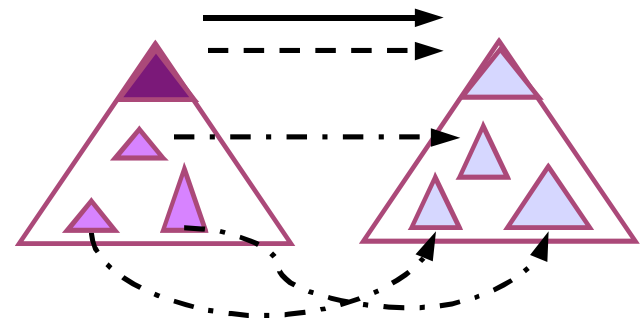
reflexivity:



congruence:



replacement:



Rule application

Suppose we have the following rules (in a suitable module).

$\text{r1}[\text{fa2b}] : f(a, x) \Rightarrow f(b, x) \ .$

$\text{r1}[\text{hh2h}] : h\ h(y) \Rightarrow h(y) \ .$

then

$g(c, f(a, d)) \Rightarrow g(c, f(b, d))$

using $[\text{fa2b}]$ and congruence

and

$h\ h(g(c, f(a, d))) \Rightarrow h(g(c, f(b, d)))$

also using replacement.

Before applying a rewrite rule, Maude reduces a term to canonical form using equations.

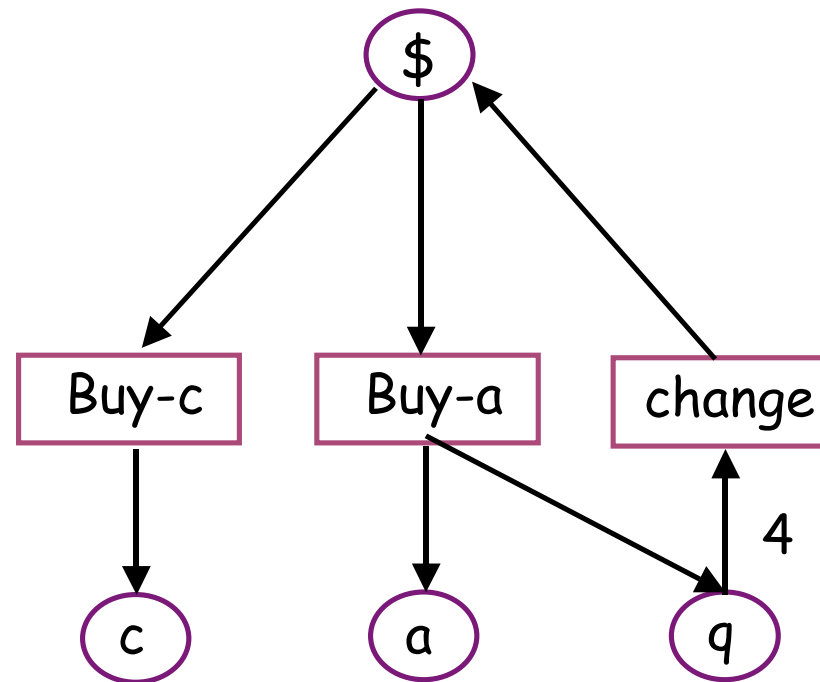
Petri Net Example

Petri nets (Place-transition nets) are a formalism for modeling concurrent system behavior.

- 0 Places represent properties or component features.
- 0 System state is represented by marking of places
- 0 Transitions specify transfer of marks from one place to another
- 0 Petri nets have a natural graphical representation and a natural representation in Maude

Petri Net Example

A vending machine



The vending machine in Maude

```
mod VENDING-MACHINE is
  sorts Coin Item Place Marking .
  subsorts Coin Item < Place < Marking .
  op null : -> Marking .          *** empty marking
  ops $ q : -> Coin .
  ops a c : -> Item .
  op _ _ : Marking Marking -> Marking
      [assoc comm id: null] .
  rl[buy-c]: $ => c .
  rl[buy-a]: $ => a q .
  rl[change]: q q q q => $ .
endm
```

What about AC Rewriting?

For terms constructed from operators that are declared with attributes such as `assoc`, `comm` and `/` or `id`, rule matching is extended to match AC terms to segments.

`q M:Marking q` matches a subterm of `a q c $ q q`

with `M:Marking := c $`

`q M:Marking a M1:Marking` matches a subterm of

`$ q c $ q a`

with `M:Marking := c $ q` and `M1:Marking := nil`

Using the vending machine

What is one way to use 3 \$s?

```
Maude> rew $ $ $ .  
result Marking: q a c c
```

How can I get 2 apples with 3 \$s?

```
Maude> search $ $ $ =>! a a M:Marking .
```

```
Solution 1 (state 8)
```

```
M:Marking --> q q c
```

```
Solution 2 (state 9)
```

```
M:Marking --> q q q a
```

```
No more solutions.
```

```
states: 10  rewrites: 12)
```

Model checking

Algorithm for determining if $M \models P$ (M satisfies P) where M is a 'model' and ' P ' is a property.

In our case a model is a Maude specification of a system together with a state of interest.

A property is a 'temporal logic' formula to be interpreted as a property of computations of the system (linear sequences of states generated by application of rewrite rules).

Model checking

Temporal formulas are built from propositions about states, using boolean connectives, and temporal operators $[]$ (always) and $\langle \rangle$ (eventually).

Satisfaction for $M \models A$ is axiomatized by equations

$M \models P$ if $C \models P$ for every computation C of M

$C \models A$ if the first state of C satisfies A

$C \models []P$ if every suffix of C satisfies P

$C \models \langle \rangle P$ if some suffix of C satisfies P

Model checking: Defining properties

```
mod MC-VENDING-MACHINE is
  inc VENDING-MACHINE .  inc MODEL-CHECKER .  inc NAT .

  op vm : Marking -> State [ctor] .
  op countPlace : Marking Place -> Nat .
  op value : Marking -> Nat .  *** in units of quarters
  ....

  ops fiveQ lte4Q : -> Prop .
  ops nCakes nApples val : Nat -> Prop .

  eq vm(M) |= fiveQ = countPlace(M,q) == 5 .
  eq vm(M) |= lte4Q = countPlace(M,q) <= 4 .
  eq vm(M) |= nApples(n) = countPlace(M,a) == n .
  eq vm(M) |= val(n) = value(M) == n .
endm
```

Model checking the vending machine I

Starting with 5 \$s, can we get 6 apples without accumulating more than 4 quarters? Model check the claim that we can't.

Maude>

```
red modelCheck(vm($ $ $ $ $), [] ~ (lte4Q U nApples(6))) .
result ModelCheckResult: counterexample(
  {vm($ $ $ $ $), 'buy-a}
  {vm($ $ $ $ q a), 'buy-a}
  {vm($ $ $ q q a a), 'buy-a}
  {vm($ $ q q q a a a), 'buy-a}
  {vm($ q q q q a a a a), 'change}
  {vm($ $ a a a a), 'buy-a}
  {vm($ q a a a a a), 'buy-a},
  {vm(q q a a a a a a), deadlock})
```

A counterexample to a formula is a pair of transition lists representing an infinite computation which fails to satisfy the formula. A transition is a state and a rule identifier. The second list represents a loop.

Model checking the vending machine II

Starting with 5 \$s, can we get 5 quarters then 6 apples?

Maude>

```
red modelCheck(vm($ $ $ $ $),  
               []~(<>fiveQ /\ (fiveQ |-> nApples(6))) .  
result ModelCheckResult: counterexample(...)
```

Is value conserved?

```
Maude> red modelCheck(vm($ $ $ $ $),[]val(20) .  
result Bool: true
```

Extracting Paths from CounterExamples

A simple path from a given initial state S_0 , to a state satisfying a property P is a list of rules together with a state S_1 satisfying P such that applying the rules starting with S_0 leads to S_1 .

One way to find a simple path is to model check the assertion that from S_0 no state can be reached satisfying P : $\text{modelCheck}(S_0, \sim \langle \rangle P)$. If there is a reachable state satisfying P , a counterexample will be returned. The counterexample contains the list of rules applied.

Using findPath

The module `SIMPLE` defines a function `findPath` that extracts a simple path from a counterexample and a property.

```
mod FP-VENDING-MACHINE is
  inc MC-VENDING-MACHINE .
  inc SIMPLE .
endm
```

```
Maude> red findPath(vm($ $ $ $), nApples(5)) .
result SimplePath:
  spath('buy-a 'buy-a 'buy-a 'buy-a 'change 'buy-a,
        vm(q a a a a a)) .
```

A Vending Machine Mall

To illustrate systems states with more structure, consider a `Mall' which contains possibly multiple vending machines, each vending a specific item. In addition the Mall contains `Goods', items purchased, and coins available to make purchases.

We add rules to put coins into machines and extract items purchased. The original vending machine rules are used to turn coins into items.

The Mall specification

```
mod VENDING-MALL is
  inc VENDING-MACHINE .
  sorts Machine Goods Mall .
  subsort Machine Goods < Mall .

  op mt : -> Mall .
  op ___ : Mall Mall -> Mall [ctor assoc comm id: mt] .

  op m : Item Marking -> Machine [ctor] .
  op g : Place -> Goods [ctor] .

  var M : Marking . var C : Coin . var I : Item .

  rl [coin-in] : m(I, M) g(C) => m(I, M C) .
  rl [apple-out] : m(a, M a) => m(a, M) g(a) .
  rl [cake-out] : m(c, M c) => m(c, M) g(c) .

endm
```


Shopping in the Mall

Consider an initial state with an two vending machines, one for apples, one for cakes, and 4 \$s.

Default shopping yields 2 apples.

```
Maude> rew m(a,null) m(c,null) g($) g($) g($) g($) .  
result Mall: g(a) g(a) m(a, q q c c) m(c, null)
```

Can I get 4 apples and a cake?

```
Maude> search m(a,null) m(c,null) g($) g($) g($) g($) =>!  
X:Mall g(a) g(a) g(a) g(a) g(c) .  
No solution.
```

But I can get 4 apples and 4 quarters stuck in the machine!

```
search m(a,null) m(c,null) g($) g($) g($) g($)  
=>! m(a, q q q q) X:Mall .  
Solution 1 (state 725)  
X:Mall --> g(a) g(a) g(a) g(a) m(c, null)
```

Reflection example: Module analysis

```
fmod CONSUMERS is
  inc MY-META .
  var M : Module .    var T : Term .
  var R : Rule .      var RS : RuleSet .

  op consumes : Module Rule Term -> Bool .
  eq consumes (M,R,T) =
    *** assumes M defines 'has
    getTerm(metaReduce (M, 'has[getLhs (R) ,T] ))
    == 'true.Bool .

  op consumerRules : Module Term -> QidSet .
  op consumerRules : Module RuleSet Term -> QidSet .

  eq consumerRules (M,T) =
    consumerRules (M,upRls (getName (M) ,true) ,T) .
  eq consumerRules (M,none,T) = none .
  eq consumerRules (M,R RS,T) =
    (if consumes (M,R,T) then getRuleId(R) else none fi) ;
    consumerRules (M,RS,T) .
endfm
```

Reflection Example: Module analysis cntd.

```
mod VEND-X is
  inc VENDING-MACHINE .

  vars M0 M1 : Marking .
  op has : Marking Marking -> Bool .

  eq has(M0 M1, M1) = true .
  eq has(M0, M1) = false [owise] .
endm
```

```
select CONSUMERS .
```

```
Maude> red consumerRules(['VEND-X], '$.Coin) .
result: QidSet 'buy-a ; buy-c
```

```
Maude> red consumerRules(['VEND-X], 'q.Coin) .
result: Qid 'change
```

Reflection example: Strategy

```
fmod METAREWRITE-LIST is
  inc MY-META .
  var M : Module .    vars T T' : Term .
  var res : Result4Tuple? .
  var rid : Qid .    var ql : QidList .

  op metaRewList : Module QidList Term -> Term .
  eq metaRewList(M,nil,T) = T .
  ceq metaRewList(M,rid ql,T) = metaRewList(M,ql,T')
    if res := metaXapply(M,T,rid,none,0,unbounded,0)
    /\ T' := if res :: Result4Tuple
              then getTerm(res)
              else T fi .

endfm
```

Reflection example: Strategy cntd.

```
Maude> red metaRewList(['VENDING-MACHINE],  
    'change 'buy-a,  
    '___['q.Coin,'q.Coin,'q.Coin,'q.Coin]) .
```

```
result GroundTerm: '___['q.Coin,'a.Item]
```

```
Maude> red metaRewList(['VENDING-MACHINE],  
    'buy-a 'change,  
    '___['q.Coin,'q.Coin,'q.Coin,'q.Coin]) .
```

```
result Constant: '$.Coin
```