

An Introduction to the Maude Formal Tool Environment

Joe Hendrix

Galois, Inc

February 2nd, 2010

Maude in One Slide

- Maude is a high-level declarative language based on term rewriting.
- Functional computation based on **membership equational logic** which refines order-sorted logic.
- **State-based computation** described using rewriting logic.
- Strong support for **reflection**:
 - Enables metaprogramming and more easily rewriting analysis tools.
- Often used for representing other logics, or giving semantics to a wide range of languages and models of concurrency.

`http://maude.cs.uiuc.edu`

Rewriting

Rewriting is a proces of applying rules oriented left-to-right to algebraic terms

Equational Rewriting

Peano's Axioms:

$$\mathbf{x} + 0 \rightarrow \mathbf{x}$$

$$\mathbf{x} + \mathbf{s}(\mathbf{y}) \rightarrow \mathbf{s}(\mathbf{x} + \mathbf{y})$$

Computing addition:

$$\mathbf{s}(0) + \mathbf{s}(0) \rightarrow \mathbf{s}(\mathbf{s}(0) + 0) \rightarrow \mathbf{s}(\mathbf{s}(0))$$

Rewriting

Rewriting is a process of applying rules oriented left-to-right to algebraic terms

Equational Rewriting

Peano's Axioms:

$$x + 0 \rightarrow x$$

$$x + s(y) \rightarrow s(x + y)$$

Computing addition:

$$s(0) + s(0) \rightarrow s(s(0) + 0) \rightarrow s(s(0))$$

Rewriting

Rewriting is a proces of applying rules oriented left-to-right to algebraic terms

Transition Systems [Dershowitz, Jounaud 90]

Game with a list of **red** and **blue** chips and moves:

red blue → **red** **blue red** → **red** **red red** → **blue**

Possible moves:

(1) blue red red blue → blue blue blue

(2) blue red red blue → red red blue → red red → blue

Rewriting

Rewriting is a proces of applying rules oriented left-to-right to algebraic terms

Transition Systems [Dershowitz, Jounaud 90]

Game with a list of **red** and **blue** chips and moves:

red blue \rightarrow **red** **blue red** \rightarrow **red** **red red** \rightarrow **blue**

Possible moves:

(1) blue red red blue \rightarrow blue blue blue

(2) blue red red blue \rightarrow red red blue \rightarrow red red \rightarrow blue

Outline

- 1 Overview
- 2 Functional Computation
 - Membership Equational Logic
 - Parameterization
 - Reflection
 - Maude ITP
- 3 Rewriting Logic
 - Introduction
 - Applications
- 4 More Information

Membership Equational Logic (MEL)

- Membership Equational Logic is a Horn logic with two levels of typing:
 - Kinds** Defined by the **signature**
 - Sorts** More refined mechanism defined by **memberships** in the theory.
- Maude only typechecks at the kind level.
- Axioms in MEL may be **conditional**. Interpreting memberships in a theory may require interpreting the equations as well.
- We will examine a formalization of **Powerlists** in MEL.

Powerlists Example [Misra, 94]

- Powerlists are lists of length 2^n for any $n \in \mathbb{N}$.
- Data structure for parallel algorithms.
- In our specification, every natural number is a Powerlist, and powerlists may be built up by **concatenation**.
- Powerlists may also be **interleaved** together.
- Only powerlists with equal length may be combined by concatenation or interleaving.
- Conditional memberships are needed to formalize this.

Powerlists Example [Misra, 94]

- Powerlists are lists of length 2^n for any $n \in \mathbb{N}$.
- Data structure for parallel algorithms.
- In our specification, every natural number is a Powerlist, and powerlists may be built up by **concatenation**.
- Powerlists may also be **interleaved** together.
- Only powerlists with equal length may be combined by concatenation or interleaving.
- Conditional memberships are needed to formalize this.

Powerlists Example [Misra, 94]

- Powerlists are lists of length 2^n for any $n \in \mathbb{N}$.
- Data structure for parallel algorithms.
- In our specification, every natural number is a Powerlist, and powerlists may be built up by **concatenation**.
- Powerlists may also be **interleaved** together.
- Only powerlists with equal length may be combined by concatenation or interleaving.
- Conditional memberships are needed to formalize this.

Powerlists over Natural Numbers in Maude

```

fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _|_ : [Pow] [Pow] -> [Pow] .
  cmb P | Q : Pow if len(P) = len(Q) .
  vars M N : Nat .      vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len([ M ]) = 1 .
  eq len(P | Q) = len(P) + len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb P X Q : Pow if len(P) = len(Q) [metadata "dfn"] .
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm

```

Powerlists over Natural Numbers in Maude

```

fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _|_ : [Pow] [Pow] -> [Pow] .
  cmb P | Q : Pow if len(P) = len(Q) .
  vars M N : Nat .      vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len([ M ]) = 1 .
  eq len(P | Q) = len(P) + len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb P X Q : Pow if len(P) = len(Q) [metadata "dfn"] .
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm

```

Powerlists over Natural Numbers in Maude

```

fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _|_ : [Pow] [Pow] -> [Pow] .
  cmb P | Q : Pow if len(P) = len(Q) .
  vars M N : Nat .      vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len([ M ]) = 1 .
  eq len(P | Q) = len(P) + len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb P X Q : Pow if len(P) = len(Q) [metadata "dfn"] .
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm

```

Powerlists over Natural Numbers in Maude

```

fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op |_| : [Pow] [Pow] -> [Pow] .
  cmb P | Q : Pow if len(P) = len(Q) .
  vars M N : Nat .      vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len([ M ]) = 1 .
  eq len(P | Q) = len(P) + len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb P X Q : Pow if len(P) = len(Q) [metadata "dfn"] .
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm

```

Powerlists over Natural Numbers in Maude

```

fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _|_ : [Pow] [Pow] -> [Pow] .
  cmb P | Q : Pow if len(P) = len(Q) .
  vars M N : Nat .      vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len([ M ]) = 1 .
  eq len(P | Q) = len(P) + len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb P X Q : Pow if len(P) = len(Q) [metadata "dfn"].
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm

```

Rewriting Modulo Axioms

- Orienting an equation $t = u$ as a rule $t \rightarrow u$ is not always a good strategy.
 - Equations like **commutativity** $f(x, y) = f(y, x)$ cannot be oriented without introducing **nontermination**.
 - Specifying certain data structures such as **queues** more difficult when the underlying list constructor is not **associative**.
- Some rewrite engines allow specific types of equations to be interpreted directly as equational **axioms**.
 - Maude supports rewriting modulo any combination of associativity, commutativity, and identity.
 - Implementing this requires specialized **matching algorithms** and **rewriting modulo axioms**.

Rewriting Modulo Axioms

- Orienting an equation $t = u$ as a rule $t \rightarrow u$ is not always a good strategy.
 - Equations like **commutativity** $f(x, y) = f(y, x)$ cannot be oriented without introducing **nontermination**.
 - Specifying certain data structures such as **queues** more difficult when the underlying list constructor is not **associative**.
- Some rewrite engines allow specific types of equations to be interpreted directly as equational **axioms**.
 - Maude supports rewriting modulo any combination of associativity, commutativity, and identity.
 - Implementing this requires specialized **matching algorithms** and **rewriting modulo axioms**.

Lists with Rewriting Modulo Associativity

```
fmod NAT-LIST is
  protecting NAT .
  sorts NeList List .  subsorts Nat < NeList < List .
  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].
  var N : Nat .    var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op last : NeList -> Nat .
  eq last(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm
```

Lists with Rewriting Modulo Associativity

```
fmod NAT-LIST is
  protecting NAT .
  sorts NeList List .  subsorts Nat < NeList < List .
  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].
  var N : Nat .    var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op last : NeList -> Nat .
  eq last(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm
```

Lists with Rewriting Modulo Associativity

```
fmod NAT-LIST is
  protecting NAT .
  sorts NeList List .  subsorts Nat < NeList < List .
  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].
  var N : Nat .    var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op last : NeList -> Nat .
  eq last(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm
```

Lists with Rewriting Modulo Associativity

```
fmod NAT-LIST is
  protecting NAT .
  sorts NeList List .  subsorts Nat < NeList < List .
  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].
  var N : Nat .    var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op last : NeList -> Nat .
  eq last(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm
```

Lists with Rewriting Modulo Associativity

```

fmod NAT-LIST is
  protecting NAT .
  sorts NeList List .  subsorts Nat < NeList < List .
  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].
  var N : Nat .    var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op last : NeList -> Nat .
  eq last(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm

```

Termination

- Equational specifications in Maude should be **terminating**.
- Though undecidable, **The Maude Termination Tool (MTT)** can automatically prove termination in some cases:

`http://www.lcc.uma.es/~duran/MTT/`

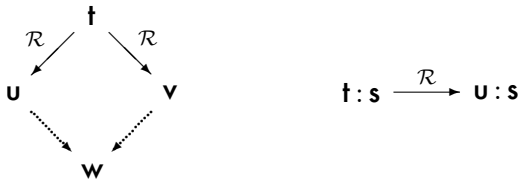
- MTT simplifies Maude module to an unsorted specification while preserving **non-termination**.
- Simplified unsorted specification can then be sent to AProVE, MU-TERM, or CiME.

Confluence

A second important property is **confluence** – which holds if the order in which rewriting is applied does not affect the ultimate normal forms.

Definition

A rewrite-membership system \mathcal{R} is **confluent** if



Maude has a confluence checker:

<http://www.lcc.uma.es/~duran/CRC/>

Parameterized Theories

- Parameterized modules use **theories** to specify the sorts, operations, and requirements a parameter must satisfy.
- Theories are membership equational specifications with a **loose** semantics.
- Simplest theory only requires existence of a sort:

```
fth TRIV is
  sort Elt .
endfth
```

- Can also have operations in theories:

```
fth A-OP is
  inc TRIV .
  op + : Elt Elt -> Elt .
  var X Y Z : Elt .
  eq X + (Y + Z) = (X + Y) + Z [nonexec].
endfth
```

Parameterized Lists

```

fmod LISTX :: TRIV is
  sorts NeList{X} List{X} .
  subsort X$Elt < NeList{X} < List{X} .

  op nil : -> List{X} [ctor] .
  op __ : NeList{X} NeList{X} -> NeList{X} [ctor assoc id: nil].
  op __ : List{X} List{X} -> List{X} [ctor assoc id: nil].
  var E : X$Elt .      var L : List{X} .

  op head : NeList{X} -> X$Elt .
  eq head(E L) = E .
  op last : NeList{X} -> X$Elt .
  eq last(L E) = E .

  op reverse : List{X} -> List{X} .
  eq reverse(E L) = reverse(L) E .
  eq reverse(nil) = nil .
endfm

```

Views

- One instantiates a parameterized module by first defining a view from the theory to the parameter.

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

- Lists over natural numbers can then be instantiated with:

```
LIST{Nat}
```

Reflection

- Maude has strong support for reflection and meta-programming.
- The reflection module `META-LEVEL` in the prelude supports:
 - Retrieving a registered module by name:
`op upModule : Qid Bool ~> Module`
 - Parsing a Maude term from a list of tokens:
`op metaParse : Module QidList Type? ~> ResultPair?`
`op _,_ : Term Type -> ResultPair [ctor] .`
 - Invoking the rewriter and other builtin procedures:
`op metaReduce : Module Term ~> ResultPair`

The Maude ITP

- Inductive theorem prover for showing that

$$T_{\mathcal{E}} \models \phi$$

where \mathcal{E} is a membership equational specification.

$T_{\mathcal{E}}$ is the term algebra for \mathcal{E} .

ϕ is a first-order logic formula.

- **Reflective** architecture:
 - Written in Maude and proves properties of Maude specifications.
 - Supports **partiality** through membership equational logic.
 - Deduction uses rewriting modulo axioms.
- Most recent version is available at:

`http://maude.cs.uiuc.edu/tools/itp/`

Rewriting Logic

- Rewriting logic drops the symmetry imposed by the interpreting rules as equations.
- In a rewrite theory, there are two types of axioms:
 - **Equations** define the static operations of the system.
 - **Rewrite rules** defines the local transitions.

```
mod CLIMATE is
sort weathercondition .
op sunnyday : -> weathercondition .
op rainyday : -> weathercondition .
rl [raincloud] : sunnyday => rainyday .
endm
```

Crossing the Bridge [Marti-Oliet, 2010]

- The four members of U2 are in a tight situation. Their concert starts in 17 minutes and to get to the stage they must first cross an old bridge through which only at most two people can cross at the same time.
- It is dark and, because of the bad condition of the bridge, to avoid falling into the darkness it is necessary to cross it with the help of a flashlight. Unfortunately, they only have one.
- Knowing that Bono, Edge, Adam, and Larry take 1, 2, 5, and 10 minutes, respectively, to cross the bridge, is there a way that they can make it to the concert on time?

Crossing the Bridge [Marti-Oliet, 2010]

```
mod U2 is protecting NAT .
  sort Place .
  ops left right : -> Place [ctor].

  op changePos : Place -> Place .
  eq changePos(left) = right .
  eq changePos(right) = left .

  sorts Performer Object Group .
  subsorts Performer Object < Group .
  op flashlight : Place -> Object [ctor].
  op watch : Nat -> Object [ctor].
  op performer : Nat Place -> Performer [ctor].
  op __ : Group Group -> Group [ctor assoc comm] .
```

Crossing the Bridge [Marti-Oliet, 2010]

```

op initial : -> Group .
eq initial
= watch(0) flashlight(left) performer(1, left)
  performer(2, left) performer(5, left) performer(10, left) .

var P : Place .      vars M N N1 N2 : Nat .

r1 [one-crosses] :
  watch(M) flashlight(P) performer(N, P)
=> watch(M + N) flashlight(changePos(P))
  performer(N, changePos(P)) .
cr1 [two-cross] :
  watch(M) flashlight(P) performer(N1, P) performer(N2, P)
=> watch(M + N1) flashlight(changePos(P))
  performer(N1, changePos(P))
  performer(N2, changePos(P))
  if N1 > N2 .

endm

```

Crossing the Bridge [Marti-Oliet, 2010]

- A solution can be found by looking for a state in which all performers and the flashlight are to the right of the bridge.
- The **search** command can be invoked to find a path from the initial state to a target state:

```
Maude> search [1] initial
=>* flashlight(right) watch(N:Nat)
    performer(1, right) performer(2, right)
    performer(5, right) performer(10, right)
  such that N:Nat <= 17 .
Solution 1 (state 402)
N --> 17
```

The path can then be retrieved with the command:

```
Maude> show path 402 .
```

Application Areas

- Models of concurrent computation (Verdejo)
 - Equational programming
 - Lambda calculi
 - Petri nets
 - CCS and μ -calculus
 - Actors
- Operational semantics of languages
 - Structural operational semantics (Braga)
 - Agent languages (Duran)
 - Mobile Maude (Eker, Lincoln)
 - K Framework (Rosu, Hills)
 - Orc Language (ALTurki)
 - MProlog (Sasse, ALTurki)
 - Java (Farzan, Chen)

Application Areas

- Distributed architectures and components (Boronaut, Clavel, Duran)
 - UML diagrams and metamodels
 - Middleware architecture for composable services
 - Reference Model for Open Distributed Processing
 - Validation of OCL properties
 - Model management and model transformations
- Modeling and analysis of security protocols
 - Model-checking of communication protocols (Olveczky)
 - Cryptographic protocol specification language CAPSL (Millen)
 - MSR security specification formalism (Sasse)
 - Maude-NPA (Escobar, Meadows, Meseguer)
- Real-time, biological, probabilistic systems
 - Real-Time Maude Tool (Olveczky)
 - Pathway Logic (SRI)
 - PMaude (Sen)

More Information

How to get more information about Maude:

- Visit the Maude website:

`http://maude.cs.uiuc.edu/`

- To get a flavor for programming, start with the **Maude Primer**.

`http://maude.cs.uiuc.edu/primer/`