Introduction to Maude

Alexander Maletzky

2013-05-15

1. Some Facts
2. Structure of Maude
3. Using Maude
4. Live Demonstration
5. Conclusion

1. Some Facts
2. Structure of Maude
3. Using Maude
4. Live Demonstration
5. Conclusion

Maude

- Rewriting system operating on (typed) terms
- Developed at SRI International
- Open source (C++)
- Current version: 2.6
- Operating systems: Linux, MacOSX (sources may be compiled on other platforms as well)
- Lots of documentation available
- URL: http://maude.cs.uiuc.edu/

1 Some Facts

2 Structure of Maude

3 Using Maude

4 Live Demonstration

5 Conclusion

Types in Maude: *Sorts*

- Maude is strictly typed
- Types are called *sorts*
- User may define sorts as he wants
- Sorts have no deeper meaning, only needed to build well-formed terms
- Hierarchies of sorts possible: *Subsorts*

Example: Sorts

```
sorts Real Irrational Rational Integer Nat .
subsorts Irrational Rational < Real .
subsorts Nat < Integer < Rational .
```

- Line 1: Declare several sorts (of numbers)
- Lines 2-3: Define hierarchy of sorts, e. g. all rational and irrational numbers are real numbers as well

Data Elements: *Operators*

- Maude operates on *terms*
- Terms are built from *operators*
- Operators are declared/defined by user
- Operator: *n*-ary function
- 0-ary operators: Constants
- When declaring operators, sorts of arguments/result have to be given explicitly
- Both prefix and mixfix notation possible

Example: Operators

```
op 0 :  -> Nat .
op S : Nat -> Nat .
ops _+_ _*_ :  Nat Nat -> Nat .
```

- Declare operators for arithmetic: Constant 0, unary successor function S, binary functions + and *
- S has to be "applied" in prefix notation, + and * may be "applied" in mixfix notation
- Example term: `S(0 + S(S(0) * S(S(0))))`

Definition of Operators: *Equations* and *Attributes*

- Operators are defined in terms of *equations* and *attributes*
- Equations consist of left-hand-side (LHS), right-hand-side (RHS), and condition (optional)
- May involve *variables* to achieve more generality
- Attributes equip operators with certain properties, e. g. associativity, commutativity, identity element, ...

Equations

- Equations are special kind of rewrite rules
- Can be used to reduce given term to normal form
- If LHS matches subterm (and condition is fulfilled), then this subterm is replaced by RHS
- Equations are supposed to “replace equals by equals”
- However, RHS should be in some sense “simpler” than LHS
- Hence, equations are used to simplify terms until normal form is reached
- Further properties are assumed implicitly: Church-Rosser, termination
- Properties are not checked, but can be checked by tools provided by Maude

Example: Equations

```
vars M, N : Nat .
eq N + 0 = N .
eq N + S(M) = S(N + M) .
ceq N * M = N if M == S(0) .
```

- Line 1: Declare 2 variables `M`, `N` of sort `Nat`
- Lines 2-3: Define addition as usual
- Line 4: Conditional equation: Result of multiplication is first argument if second argument is `S(0)`

Attributes

- Attributes of operator are taken into account when matching is attempted
- Example: If operator `f` is declared to be commutative and LHS of equation is `f(a, b)`, then LHS also matches term `f(b, a)`
- Most attributes could also be stated by means of equations, **but**
- Matching algorithm takes into account attributes in very efficient way **and**
- RHS would not be simpler than LHS in most cases (consider commutativity)

Example: Attributes

```
sorts Nat Set .  subsort Nat < Set .
ops 0 1 2 :  -> Nat .
op _ _ :  Set Set -> Set [comm, assoc] .
op containsZero :  Set -> Bool .
```

- Line 3: Operator `_ _` is commutative and associative
- This operator can be regarded as "union of (multi-)sets"
- We could then write, for instance

  ```
  eq containsZero(0 Rest) = true .
  ```

  where `Rest` is variable of sort `Set`
- This equation is sufficient to get positive answer whenever set contains 0
- Reason: Although 0 may not be first element, due to commutativity and associativity, any set containing 0 is matched by LHS of equation

State Transitions: *Rules*

- Similar to equations, but not the same
- Consist of LHS, RHS, label and condition (optional)
- Again, if LHS matches some subterm, then this subterm is replaced by RHS
- Used to model state transitions (no “replace equals by equals”)
- Not assumed to have Church-Rosser/termination property

Example: Rules

```
rl [birthday] :  person(X, N) => person(X, N + S(0)) .
crl [get-married] :  person(single, N) =>
    person(married, N) if N >= 16 .
```

- A person may have birthday at any time, but may get married only if at least 16 years old
- RHSs of rules are not simpler than LHSs
- Labels (`birthday`, `married`) are optional
- Operator `person` is only used to combine several properties of persons

Main Building Block: *Modules*

- Modules define theories/systems
- Combine all previously mentioned concepts
- 2 types of modules:
  - *Functional* modules: Define functional theories (e. g. natural numbers) by means of equations, **may not contain rules**
  - *System* modules: Define systems (concurrent, non-deterministic) by means of rules
- Hierarchy: Modules may be built upon other modules, but *functional* modules may only be built upon other *functional* modules
- Lots of predefined modules available

Example: Functional Module

```
fmod NAT-NUMBERS is
  sort Nat .

  op 0 :  -> Nat .
  op S : Nat -> Nat .
  op _+_ :  Nat Nat -> Nat .
  op _>=_ :  Nat Nat -> Bool .

  vars M, N : Nat .
  eq N + 0 = N .
  eq N + S(M) = S(N + M) .
  eq N >= 0 = true .
  eq 0 >= S(M) = false .
  eq S(N) >= S(M) = N >= M .
endfm
```

Example: System Module

```
mod RELATIONSHIP is
  protecting NAT-NUMBERS .

  sorts Person State .

  ops single engaged married :  -> State .
  op person :  State Nat -> Person .

  var X : State .
  var N : Nat .
  rl [birthday] :  person(X, N) => person(X, N + S(0)) .
  crl [get-engaged] :  person(single, N) => person(engaged, N)
    if N >= 16 .
  rl [get-married] :  person(engaged, N) => person(married, N) .
  crl [las-vegas] :  person(single, N) => person(married, N)
    if N >= 16 .
  crl [split-up] :  X => single if X =/= single .
endm
```

1 Some Facts

2 Structure of Maude

3 Using Maude

4 Live Demonstration

5 Conclusion

Command `reduce`

- `reduce in` *module* `:` *term* `.`
- Reduces term *term* to canonical form using equations from module *module* (no rules!)
- Module may be functional or system
- Output:
  - Number of rewrites (= equations)
  - CPU time
  - Sort of resulting term
  - Resulting term

Example: reduce

- Input:

```
reduce in NAT-NUMBERS :
  (S(S(0)) + S(S(0))) >= (S(0) + S(S(0))) .
```

- Output:

```
reduce in NAT-NUMBERS :
  (S(S(0)) + S(S(0))) >= (S(0) + S(S(0))) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool:  true
```

Command `rewrite`

- `rewrite [`*`bound`*`] in `*`module`*` : `*`term`*` .`
- Rewrites term *term* using rules and equations from module *module*
- At most *bound* **rules** are applied
- Top-down rule-fair strategy: All rules that can be applied to outermost operator are applied in fair way
- Other rules might not be applied at all
- In each step:
  1. Rewrite term in 1 step
  2. Reduce resulting term to normal form

  $\rightarrow$ Only normal forms are rewritten!
- Output: Same as with `reduce`

Command `frewrite`

- `frewrite [`*bound*`] in` *module* `:` *term* `.`
- Behaves similar to `rewrite`, **but**
- Depth-first position-fair strategy
- Output: Same as with `rewrite`

System Module RELATIONSHIP

```
mod RELATIONSHIP is
  protecting NAT-NUMBERS .

  sorts Person State .

  ops single engaged married :  -> State .
  op person :  State Nat -> Person .

  var X : State .
  var N : Nat .
  rl [birthday] :  person(X, N) => person(X, N + S(0)) .
  crl [get-engaged] :  person(single, N) => person(engaged, N)
    if N >= 16 .
  rl [get-married] :  person(engaged, N) => person(married, N) .
  crl [las-vegas] :  person(single, N) => person(married, N)
    if N >= 16 .
  crl [split-up] :  X => single if X =/= single .
endm
```

Example: `rewrite`

- Input:
  ```
  rewrite [7] in RELATIONSHIP : person(single, 15) .
  ```
- Output:
  ```
  rewrite [7] in RELATIONSHIP : person(single, 15) .
  rewrites: 34 in 0ms cpu (0ms real) (~ rewrites/second)
  result Person: person(married, 20)
  ```
- Rule `[split-up]` will never be applied, since outermost operator in its LHS is not `person`
- Different if `frewrite` was used instead
- Whenever rule `[birthday]` is applied, age is automatically reduced to normal form

Coherence

- *Coherence*: Property of system module (equations, attributes, rules)
- t, t', u arbitrary terms such that
  - t can be rewritten in 1 step into t'
  - u is normal from of t
- If u can be rewritten into u' in 1 step such that t' and u' have same normal from, then coherence property holds
- Coherence allows using strategy pursued by `rewrite` and `frewrite`: Only rewrite normal forms
- Coherence is implicitly assumed and may be checked by tools provided by Maude

Command search

- `search [`*n*`, `*m*`] in `*module*` : `*t1 arrow t2*` such that `*C*` .`
- Search for all states reachable from initial state that meet certain conditions
- *n*: Maximum number of solutions
- *m*: Maximum search depth
- *t*1: Initial state
- *t*2: Pattern of final states (may involve variables)
- *arrow*: Defines *how* final states are reached:
  - `=>1`: Exactly 1 step
  - `=>+`: At least one step
  - `=>*`: Arbitrarily many steps
  - `=>!`: Final states must be terminal
- *C*: Optional condition the final states have to meet

Example: search

- Input:
```
search [1,10] in RELATIONSHIP :
  person(single, 15) =>* person(married, 20) .
```
- Output:
```
search [1,10] in RELATIONSHIP :
  person(single, 15) =>* person(married, 20) .
Solution 1 (state 16)
states:  17
rewrites: 264 in 0ms cpu(2ms real) (~ rewrites/second)
empty substitution
```
- It is also possible to see path from initial state to final state

Model Checking: Invariants

- `search` can be used to model-check systems w. r. t. *invariants*
- Invariant: Property that holds in all states reachable from initial state
- Just search for states that violate invariant
- If none found → Invariant holds
- Otherwise → Counterexample
- Drawback: Only works for finitely many states

LTL Model Checking

- Maude supports LTL model checking
- No Maude-command, but predefined functional module with main operator `modelCheck`
- Systems that have to be checked need to include this module
- Command: `reduce modelCheck(`*state*`,`*formula*`) .`
- *state*: Initial state
- *formula*: LTL formula
- Constraint: Finitely many reachable states
- Example: $\rightarrow$ Later (live demonstration)

LTL Satisfiability/Tautology

- Maude supports testing LTL formulas for *satisfiability* and *tautology*
- Satisfiability: There exists system that satisfies formula
- Tautology: Formula always holds, i. e. negation of formula is unsatisfiable
- In case of satisfiability, Maude returns model in terms of initial path and cycle

Example: Satisfiability

- Input:
```
reduce in SAT-SOLVER-TEST :
  satSolve(a /\ (O b) /\ (O O ((~ c) /\ [](c \/ (O c))))) .
```
- Output:
```
reduce in SAT-SOLVER-TEST :
  satSolve(O O (~ c /\ [](c \/ O c)) /\ (a /\ O b)) .
rewrites:  2 in 0ms cpu (0ms real) (~ rewrites/second)
result SatSolveResult:  model(a ; b, (~ c) ; c)
```
- Hence, formula is satisfiable

Example: Tautology

- Input:
```
reduce in SAT-SOLVER-TEST :
  tautCheck((a => (O a)) <-> (a => ([] a))) .
```
- Output:
```
reduce in SAT-SOLVER-TEST :
  tautCheck((a => O a) <-> a => []a) .
rewrites:  49 in 0ms cpu (1ms real)
(~ rewrites/second)
result Bool:  true
```
- Hence, LTL formula is tautology
- Otherwise we would also get counterexample

1 Some Facts

2 Structure of Maude

3 Using Maude

4 Live Demonstration

5 Conclusion

Example System BANK-ACCOUNT

- Message-passing system
- Objects: Bank accounts
  - ID
  - Balance
- Messages:
  - Credit
  - Debit
  - Transfer-from-to
- Objects and messages are contained in set $\rightarrow$ Order is not relevant
- Set is built from binary operator having commutativity and associativity attributes
- Powerful predefined module for modelling such (object-oriented) systems

Model-Checking BANK-ACCOUNT

- Atomic predicate `debts`(A)
- `debts`(A) holds in state S iff balance of account A is negative
- System is model-checked for never reaching state where `debts`(A) holds for some account A
- LTL formula: $\Box\neg\mathtt{debts}(A)$
- Since this is an invariant, command `search` could be used as well

1 Some Facts

2 Structure of Maude

3 Using Maude

4 Live Demonstration

5 Conclusion

Additional Features

- Highly flexible, user-definable syntax (additional attributes for correct parsing of mixfix operators)
- Efficient implementation
- Verification capabilities
  - Church-Rosser
  - Termination
  - Coherence
  - Sufficient completeness
  - ...
- Reflection: Represent terms, equations, rules, modules, ... as terms at meta-level and work with them
- Reflection is useful to define different rewriting-strategies

Sources

- http://maude.cs.uiuc.edu/
- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott. *Maude Manual (Version 2.6)*. January 2011
- T. McCombs. *Maude 2.0 Primer*. August 2003
- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. Quesada. *A Maude Tutorial*. March 2000
- M. Clavel, S. Eker, P. Lincoln, J. Meseguer. *Principles of Maude*. In: Electronic Notes in Theoretical Computer Science, Vol. 4, 1996