An introduction to Maude and some of its applications

Narciso Martí-Oliet

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
narciso@esi.ucm.es

PADL 2010, Madrid

What is Maude?

Maude in a nutshell

http://maude.cs.uiuc.edu

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
- Membership equational logic improves order-sorted algebra.
- Rewriting logic is a logic of concurrent change.
- It is a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency.
- It is also a good logical framework, i.e., a metalogic in which many other logics can be naturally represented and implemented.
- Moreover, rewriting logic is reflective.
- This makes possible many advanced metaprogramming and metalanguage applications.

Why declarative?

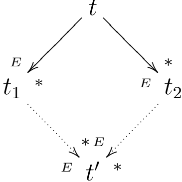
- Maude follows a long tradition of algebraic specification languages in the OBJ family, including
  - OBJ3,
  - CafeOBJ,
  - Elan.
- Computation = Deduction in an appropriate logic.
- Functional modules = (Admissible) specifications in membership equational logic.
- System modules = (Admissible) specifications in rewriting logic.
- Operational semantics based on matching and rewriting.

Matching and rewriting

- Given a term t, the pattern, and a subject term u, we say that t matches u if there is a substitution σ such that $\sigma(t) \equiv u$, that is, $\sigma(t)$ and u are syntactically equal terms.
- In an admissible equation $l = r$, all variables in the righthand side r must appear among the variables of the lefthand side l.
- A term t rewrites to a term t' using such an equation $l = r$ in E if
  1. there is a subterm $t|_p$ of t at a position p of t such that l matches $t|_p$ via a substitution σ, and then
  2. $t' = t[\sigma(r)]_p$ is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$.
- We denote this step of equational simplification by $t \rightarrow_E t'$.
- As usual, $\rightarrow_E^*$ denotes the reflexive and transitive closure of $\rightarrow_E$.

Confluence and termination

- A set of equations E is confluent (or Church-Rosser) when any two rewritings of a term can always be joined by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.



- A set of equations E is terminating when there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \dots$

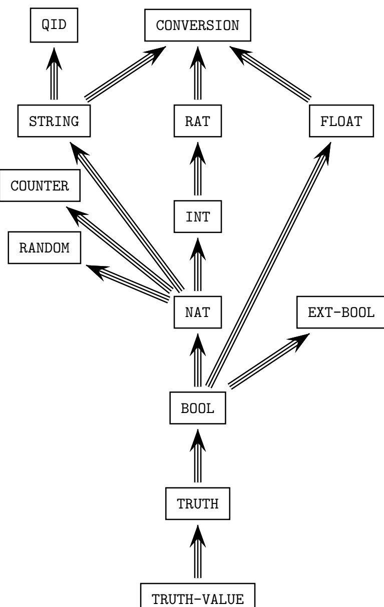
Conditional equations

- If E is both confluent and terminating, a term t can be reduced to a unique canonical form $t\downarrow_E$, that is, to a term that can no longer be rewritten.
- Therefore, in order to check semantic equality of two terms $t = t'$, it is enough to check that their respective canonical forms are equal, $t\downarrow_E = t'\downarrow_E$, but, since canonical forms cannot be rewritten anymore, the last equality is just syntactic coincidence: $t\downarrow_E \equiv t'\downarrow_E$.
- This is the way to check satisfaction of equational conditions in conditional equations.

Order-sorted equational specifications

- We can often avoid some partiality by extending many-sorted equational logic to order-sorted equational logic.
- We can define subsorts corresponding to the domain of definition of a function, whenever such subsorts can be specified by means of constructors.
- Subsorts are interpreted semantically by subset inclusion.
- Operations can be overloaded.
- A term can have several different sorts. Preregularity requires each term to have a least sort that can be assigned to it.
- Maude assumes that modules are preregular, and generates warnings when a module is loaded if the property does not hold.
- Admissible equations are assumed sort-decreasing.

Predefined modules



Lists of natural numbers

```
fmod NAT-LIST-CONS is
   protecting NAT .

   sorts NeList List .
   subsort NeList < List .

   op [] : -> List [ctor] .                    *** empty list
   op _:_ : Nat List -> NeList [ctor] .        *** cons
   op tail : NeList -> List .
   op head : NeList -> Nat .
   op _++_ : List List -> List .               *** concatenation
   op length : List -> Nat .
   op reverse : List -> List .
   op take_from_ : Nat List -> List .
   op throw_from_ : Nat List -> List .

   vars N M : Nat .
   vars L L' : List .
```

Lists of natural numbers

```
    eq tail(N : L) = L .
    eq head(N : L) = N .
    eq [] ++ L = L .
    eq (N : L) ++ L' = N : (L ++ L') .
    eq length([]) = 0 .
    eq length(N : L) = 1 + length(L) .
    eq reverse([]) = [] .
    eq reverse(N : L) = reverse(L) ++ (N : []) .

    eq take 0 from L = [] .
    eq take N from [] = [] .
    eq take s(N) from (M : L) = M : take N from L .
    eq throw 0 from L = L .
    eq throw N from [] = [] .
    eq throw s(N) from (M : L) = throw N from L .
endfm
```

Equational attributes

- Equational attributes are a means of declaring certain kinds of structural axioms in a way that allows Maude to use these equations efficiently in a built-in way.
    - `assoc` (associativity),
    - `comm` (commutativity),
    - `idem` (idempotency),
    - `id:` t (identity, where t is the identity element),
    - left identity and right identity.
- These attributes are only allowed for binary operators satisfying some appropriate requirements depending on the attributes.

Matching and simplification modulo

- In the Maude implementation, rewriting modulo A is accomplished by using a matching modulo A algorithm.
- More precisely, given an equational theory A, a term t (corresponding to the lefthand side of an equation) and a subject term u, we say that t matches u modulo A if there is a substitution σ such that $\sigma(t) =_A u$, that is, $\sigma(t)$ and u are equal modulo the equational theory A.
- Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operators, Maude synthesizes a combined matching algorithm for the theory A, and does equational simplification modulo the axioms A.

A hierarchy of data types

- nonempty binary trees, with elements only in their leaves, built with a free binary constructor, that is, a constructor with no equational axioms,
- nonempty lists, built with an associative constructor,
- lists, built with an associative constructor and an identity,
- multisets (or bags), built with an associative and commutative constructor and an identity,
- sets, built with an associative, commutative, and idempotent constructor and an identity.

Basic natural numbers

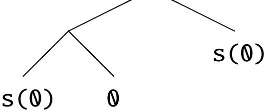
```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .

  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq max(0, M) = M .
  eq max(N, 0) = N .
  eq max(s(N), s(M)) = s(max(N, M)) .
endfm
```

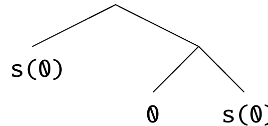
Nonempty binary trees

```
fmod NAT-TREES is
  protecting BASIC-NAT .

  sorts Tree .
  subsort Nat < Tree .
  op __ : Tree Tree -> Tree [ctor] .
  op depth : Tree -> Nat .
  op width : Tree -> Nat .

  var N : Nat .
  vars T T' : Tree .

  eq depth(N) = s(0) .
  eq depth(T T') =  s(max(depth(T), depth(T'))) .
  eq width(N) = s(0) .
  eq width(T T') = width(T) + width(T') .
endfm
```

Nonempty binary trees

- An expression such as `s(0) 0 s(0)` is ambiguous because it can be parsed in two different ways, and parentheses are necessary to disambiguate `(s(0) 0) s(0)` from `s(0) (0 s(0))`.
- These two different terms correspond to the following two different trees:





Nonempty lists

```
fmod NAT-NE-LISTS is
  protecting BASIC-NAT .

  sort NeList .
  subsort Nat < NeList .
  op __ : NeList NeList -> NeList [ctor assoc] .
  op length : NeList -> Nat .
  op reverse : NeList -> NeList .

  var N : Nat .
  var L L' : NeList .

  eq length(N) = s(0) .
  eq length(L L') = length(L) + length(L') .
  eq reverse(N) = N .
  eq reverse(L L') = reverse(L') reverse(L) .
endfm
```

Lists

```
fmod NAT-LISTS is
  protecting BASIC-NAT .

  sorts NeList List .
  subsorts Nat < NeList < List .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op __ : NeList NeList -> NeList [ctor assoc id: nil] .
  op tail : NeList -> List .
  op head : NeList -> Nat .
  op length : List -> Nat .
  op reverse : List -> List .

  var N : Nat .
  var L : List .

  eq tail(N L) = L .
  eq head(N L) = N .
```

Lists

```
  eq length(nil) = 0 .
  eq length(N L) = s(0) + length(L) .
  eq reverse(nil) = nil .
  eq reverse(N L) = reverse(L) N .
endfm
```

- The alternative equation `length(L L') = length(L) + length(L')` (with `L` and `L'` variables of sort `List`) causes problems of nontermination.
- Consider the instantiation with `L'` $\mapsto$ `nil` that gives

```
  length(L nil) = length(L) + length(nil)
                = length(L nil) + length(nil)
                = (length(L) + length(nil)) + length(nil)
                = ...
```

  because of the identification `L = L nil`.

Multisets

```
fmod NAT-MSETS is
  protecting BASIC-NAT .
  sort Mset .
  subsorts Nat < Mset .
  op empty-mset : -> Mset [ctor] .
  op __ : Mset Mset -> Mset [ctor assoc comm id: empty-mset] .
  op size : Mset -> Nat .
  op mult : Nat Mset -> Nat .
  op _in_ : Nat Mset -> Bool .

  vars N N' : Nat .
  var  S : Mset .

  eq size(empty-mset) = 0 .
  eq size(N S) = s(0) + size(S) .
  eq mult(N, empty-mset) = 0 .
  eq mult(N, N S) = s(0) + mult(N, S) .
  ceq mult(N, N' S) = mult(N, S) if N =/= N' .
  eq N in S = (mult(N, S) =/= 0) .
endfm
```

Sets

```
fmod NAT-SETS is
  protecting BASIC-NAT .
  sort Set .
  subsorts Nat < Set .
  op empty-set : -> Set [ctor] .
  op __ : Set Set -> Set [ctor assoc comm id: empty-set] .

  vars N N' : Nat .
  vars S S' : Set .

  eq N N = N .
```

The idempotency equation is stated only for singleton sets, because stating it for arbitrary sets in the form `S S = S` would cause nontermination due to the identity attribute:

$$\texttt{empty-set} = \texttt{empty-set empty-set} \rightarrow \texttt{empty-set} \ldots$$

Sets

```
    op _in_ : Nat Set -> Bool .
    op delete : Nat Set -> Set .
    op card : Set -> Nat .

    eq N in empty-set = false .
    eq N in (N' S) = (N == N') or (N in S) .
    eq delete(N, empty-set) = empty-set .
    eq delete(N, N S) = delete(N, S) .
    ceq delete(N, N' S) = N' delete(N, S) if N =/= N' .
    eq card(empty-set) = 0 .
    eq card(N S) = s(0) + card(delete(N,S)) .
  endfm
```

The equations for `delete` and `card` make sure that further occurrences of `N` in `S` on the righthand side are also deleted or not counted, resp., because we cannot rely on the order in which equations are applied.

Membership equational logic specifications

- In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of sorted lists, for example, defined by a property over lists.
- There is also a different problem of a more syntactic character. For example, with operations of difference and division on natural numbers, the term `s(s(s(0))) div (s(s(0)) - s(0))` would not be well formed, because the subterm `s(s(0)) - s(0)` has least sort `Nat`, while the `div` operation would expect its second argument to be of sort `NzNat < Nat`.
- This is too restrictive and makes most (really) order-sorted specifications useless, unless there is a mechanism that gives at parsing time the benefit of the doubt to this kind of terms.
- Membership equational logic solves both problems, by introducing sorts as predicates and allowing subsort definition by means of conditions involving equations and/or sort predicates.

Membership equational logic

- A signature in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ where K is a set of kinds, (K, Σ) is a many-kinded signature, and $S = \{S_k\}_{k \in K}$ is a K-kinded set of sorts.
- An Ω-algebra is then a (K, Σ)-algebra **A** together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$.
- Atomic formulas are either Σ-equations, or membership assertions of the form $t : s$, where the term t has kind k and $s \in S_k$.
- General sentences are Horn clauses on these atomic formulas, quantified by finite sets of K-kinded variables.

$$(\forall X) \;\; t = t' \;\; \text{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

$$(\forall X) \;\; t : s \;\; \text{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic in Maude

- Maude functional modules are membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification.
- Maude does automatic kind inference from the sorts declared by the user and their subsort relations.
- Kinds are not declared explicitly, and correspond to the connected components of the subsort relation.
- The kind corresponding to a sort `s` is denoted `[s]`.
- If `NzNat` < `Nat`, then `[NzNat]` = `[Nat]`.

Membership equational logic in Maude

- An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

  can be understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

  together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

- A subsort declaration `NzNat < Nat` can be understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

Sorted lists

```
fmod NAT-SORTED-LIST is
  protecting NAT-LIST-CONS .

  sorts SortedList NeSortedList .
  subsort NeSortedList < SortedList NeList < List .

  op insertion-sort : List -> SortedList .
  op insert-list : SortedList Nat -> SortedList .

  op mergesort : List -> SortedList .
  op merge : SortedList SortedList -> SortedList [comm] .

  op quicksort : List -> SortedList .
  op leq-elems : List Nat -> List .
  op gr-elems : List Nat -> List .

  vars N M : Nat .
  vars L L' : List .
  vars OL OL' : SortedList .
  var NEOL : NeSortedList .
```

Sorted lists

```
mb [] : SortedList .
mb N : [] : NeSortedList .
cmb N : NEOL : NeSortedList if N <= head(NEOL) .

eq insertion-sort([]) = [] .
eq insertion-sort(N : L) = insert-list(insertion-sort(L), N) .

eq insert-list([], M) = M : [] .
ceq insert-list(N : OL, M) = M : N : OL if M <= N .
ceq insert-list(N : OL, M) = N : insert-list(OL, M) if M > N .

eq mergesort([]) = [] .
eq mergesort(N : []) = N : [] .
ceq mergesort(L) =
    merge(mergesort(take (length(L) quo 2) from L),
          mergesort(throw (length(L) quo 2) from L))
 if length(L) > s(0) .
```

Sorted lists

```
  eq merge(OL, []) = OL .
  ceq merge(N : OL, M : OL') = N : merge(OL, M : OL') if N <= M .

  eq quicksort([]) = [] .
  eq quicksort(N : L)
    = quicksort(leq-elems(L,N)) ++ (N : quicksort(gr-elems(L,N))) .

  eq leq-elems([], M) = [] .
  ceq leq-elems(N : L, M) = N : leq-elems(L, M) if N <= M .
  ceq leq-elems(N : L, M) = leq-elems(L, M) if N > M .
  eq gr-elems([], M) = [] .
  ceq gr-elems(N : L, M) = gr-elems(L, M) if N <= M .
  ceq gr-elems(N : L, M) = N : gr-elems(L, M) if N > M .
endfm
```

Parameterization: theories

- Parameterized datatypes use theories to specify the requirements that the parameter must satisfy.
- A (functional) theory is a membership equational specification whose semantics is loose.
- Equations in a theory are not used for rewriting or equational simplication and, thus, they need not be confluent or terminating.
- Simplest theory only requires existence of a sort:

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

Order theories

- Theory requiring a strict total order over a given sort:

```
fth STOSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false [nonexec label irreflexive] .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X [nonexec label antisymmetric] .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth
```

Order theories

- Theory requiring a nonstrict total order over a given sort:

```
fth TOSET is
  including STOSET .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= X = true [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth
```

Parameterization: views

- Theories are used in a parameterized module expression such as

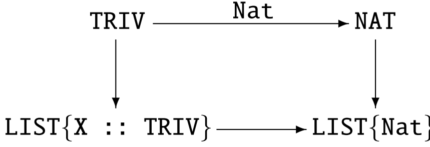
```
fmod LIST{X :: TRIV} is ... endfm
```

  to make explicit the requirements over the argument module.
- A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module.
- Each view declaration has an associated set of proof obligations, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. This may in general require inductive proof techniques.
- In many simple cases it is completely obvious:

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

Parameterization: instantiation

- A module expression such as LIST{Nat} denotes the instantiation of the parameterized module LIST{X :: TRIV} by means of the previous view Nat.



- Views can also go from theories to theories, meaning an instantiation that is still parameterized.

```
view Toset from TRIV to TOSET is
  sort Elt to Elt .
endv
```

- It is possible to have more than one view from a theory to a module or to another theory.

Parameterized lists

```
fmod LIST-CONS{X :: TRIV} is
  protecting NAT .

  sorts NeList{X} List{X} .
  subsort NeList{X} < List{X} .

  op [] : -> List{X} [ctor] .
  op _:_ : X$Elt List{X} -> NeList{X} [ctor] .
  op tail : NeList{X} -> List{X} .
  op head : NeList{X} -> X$Elt .

  var E : X$Elt .
  var N : Nat .
  vars L L' : List{X} .

  eq tail(E : L) = L .
  eq head(E : L) = E .
```

Parameterized lists

```
    op _++_ : List{X} List{X} -> List{X} .
    op length : List{X} -> Nat .
    op reverse : List{X} -> List{X} .
    op take_from_ : Nat List{X} -> List{X} .
    op throw_from_ : Nat List{X} -> List{X} .

    eq [] ++ L = L .
    eq (E : L) ++ L' = E : (L ++ L') .
    eq length([]) = 0 .
    eq length(E : L) = 1 + length(L) .
    eq reverse([]) = [] .
    eq reverse(E : L) = reverse(L) ++ (E : []) .
    eq take 0 from L = [] .
    eq take N from [] = [] .
    eq take s(N) from (E : L) = E : take N from L .
    eq throw 0 from L = L .
    eq throw N from [] = [] .
    eq throw s(N) from (E : L) = throw N from L .
  endfm
```

Parameterized sorted lists

```
view Toset from TRIV to TOSET is
  sort Elt to Elt .
endv

fmod SORTED-LIST{X :: TOSET} is
  protecting LIST-CONS{Toset}{X} .

  sorts SortedList{X} NeSortedList{X} .
  subsorts NeSortedList{X} < SortedList{X} < List{Toset}{X} .
  subsort NeSortedList{X} < NeList{Toset}{X} .

  vars N M : X$Elt .
  vars L L' : List{Toset}{X} .
  vars OL OL' : SortedList{X} .
  var NEOL : NeSortedList{X} .
```

Parameterized sorted lists

```
    mb [] : SortedList{X} .
    mb (N : []) : NeSortedList{X} .
    cmb (N : NEOL) : NeSortedList{X} if N <= head(NEOL) .

    op insertion-sort : List{Toset}{X} -> SortedList{X} .
    op insert-list : SortedList{X} X$Elt -> SortedList{X} .

    op mergesort : List{Toset}{X} -> SortedList{X} .
    op merge : SortedList{X} SortedList{X} -> SortedList{X} [comm] .

    op quicksort : List{Toset}{X} -> SortedList{X} .
    op leq-elems : List{Toset}{X} X$Elt -> List{Toset}{X} .
    op gr-elems : List{Toset}{X} X$Elt -> List{Toset}{X} .

    *** equations as before
endfm
```

Parameterized sorted lists

```
view NatAsToset from TOSET to NAT is
  sort Elt to Nat .
endv

fmod SORTED-LIST-TEST is
  protecting SORTED-LIST{NatAsToset} .
endfm

Maude> red insertion-sort(5 : 4 : 3 : 2 : 1 : 0 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red mergesort(5 : 3 : 1 : 0 : 2 : 4 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red quicksort(0 : 1 : 2 : 5 : 4 : 3 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []
```

Rewriting logic

- We arrive at the main idea behind rewriting logic by dropping symmetry and the equational interpretation of rules.
- We interpret a rule $t \longrightarrow t'$ computationally as a local concurrent transition of a system, and logically as an inference step from formulas of type t to formulas of type t'.
- Rewriting logic is a logic of becoming or change, that allows us to specify the dynamic aspects of systems.
- Representation of systems in rewriting logic:
  - The static part is specified as an equational theory.
  - The dynamics is specified by means of possibly conditional rules that rewrite terms, representing parts of the system, into others.
  - The rules need only specify the part of the system that actually changes: the frame problem is avoided.

Rewriting logic

- A rewriting logic signature is an equational specification (Ω, E) that makes explicit the set of equations in order to emphasize that rewriting will operate on congruence classes of terms modulo E.
- Sentences are rewrites of the form $[t]_E \longrightarrow [t']_E$.
- A rewriting logic specification $\mathcal{R} = (\Omega, E, L, R)$ consists of:
  - a signature (Ω, E),
  - a set L of labels, and
  - a set R of labelled rewrite rules $\quad r : [t]_E \longrightarrow [t']_E$ where r is a label and $[t]_E, [t']_E$ are congruence classes of terms in $\mathcal{T}_{\Omega,E}(X)$.
- The most general form of a rewrite rule is conditional:

$$r : t \to t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

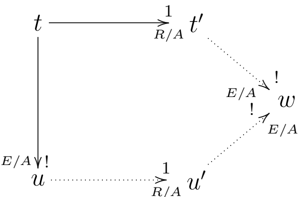
System modules

- System modules in Maude correspond to rewrite theories in rewriting logic.
- A rewrite theory has both rules and equations, so that rewriting is performed modulo such equations.
- The equations are divided into
  - a set A of structural axioms, for which matching algorithms exist in Maude, and
  - a set E of equations that are Church-Rosser and terminating modulo A;

  that is, the equational part must be equivalent to a functional module.

System modules

- The rules R in the module must be coherent with the equations E modulo A, allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken.



- A simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R.
- In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A.

Crossing the bridge

- The four components of U2 are in a tight situation. Their concert starts in 17 minutes and in order to get to the stage they must first cross an old bridge through which only a maximum of two persons can walk over at the same time.
- It is already 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

- The current state of the group can be represented by a multiset (a term of sort `Group` below) consisting of performers, the flashlight, and a watch to keep record of the time.
- The flashlight and the performers have a `Place` associated to them, indicating whether their current position is to the left or to the right of the bridge.
- Each performer, in addition, also carries the time it takes him to cross the bridge.
- In order to change the position from `left` to `right` and vice versa, we use an auxiliary operation `changePos`.
- The traversing of the bridge is modeled by two rewrite rules: the first one for the case in which a single person crosses it, and the second one for when there are two.

Crossing the bridge

```
mod U2 is
  protecting NAT .

  sorts Performer Object Group Place .
  subsorts Performer Object < Group .

  ops left right : -> Place .
  op flashlight : Place -> Object .
  op watch : Nat -> Object .
  op performer : Nat Place -> Performer .
  op __ : Group Group -> Group [assoc comm] .

  op changePos : Place -> Place .

  eq changePos(left) = right .
  eq changePos(right) = left .
```

Crossing the bridge

```
    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 .

    rl [one-crosses] :
      watch(M) flashlight(P) performer(N, P)
      => watch(M + N) flashlight(changePos(P))
         performer(N, changePos(P)) .

    crl [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

- 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 is invoked with a `such that` clause that allows to introduce a condition that solutions have to fulfill, in our example, that the total time is less than or equal to 17 minutes:

```
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
```

Crossing the bridge

- The solution takes exactly 17 minutes (a happy ending after all!) and the complete sequence of appropriate actions can be shown with the command

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

- After sorting out the information, it becomes clear that Bono and Edge have to be the first to cross. Then Bono returns with the flashlight, which gives to Adam and Larry. Finally, Edge takes the flashlight back to Bono and they cross the bridge together for the last time.
- Note that, in order for the `search` command to stop, we need to tell Maude to look only for one solution. Otherwise, it will continue exploring all possible combinations, increasingly taking a larger amount of time, and it will never end.

Model checking

- Two levels of specification:
  - a system specification level, provided by the rewrite theory specified by that system module, and
  - a property specification level, given by some properties that we want to state and prove about our module.
- Temporal logic allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens), related to the infinite behavior of a system.
- Maude 2 includes a model checker to prove properties expressed in linear temporal logic (LTL).

Linear temporal logic

- Main connectives:
  - True: $\top \in \mathrm{LTL}(AP)$.
  - Atomic propositions: If $p \in AP$, then $p \in \mathrm{LTL}(AP)$.
  - Next operator: If $\varphi \in \mathrm{LTL}(AP)$, then $\bigcirc\varphi \in \mathrm{LTL}(AP)$.
  - Until operator: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then $\varphi \,\mathcal{U}\, \psi \in \mathrm{LTL}(AP)$.
  - Boolean connectives: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then the formulae $\neg\varphi$, and $\varphi \vee \psi$ are in LTL(AP).
- Other Boolean connectives:
  - False: $\bot = \neg\top$
  - Conjunction: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - Implication: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.

Linear temporal logic

- Other temporal operators:
  - Eventually: $\Diamond\varphi = \top\,\mathcal{U}\,\varphi$
  - Henceforth: $\Box\varphi = \neg\Diamond\neg\varphi$
  - Release: $\varphi\,\mathcal{R}\,\psi = \neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$
  - Unless: $\varphi\,\mathcal{W}\,\psi = (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$
  - Leads-to: $\varphi \leadsto \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
  - Strong implication: $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
  - Strong equivalence: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$.

Kripke structures

- A Kripke structure is a triple $\mathcal{A} = (A, \to_{\mathcal{A}}, L)$ such that
  - A is a set, called the set of states,
  - $\to_{\mathcal{A}}$ is a total binary relation on A, called the transition relation, and
  - $L : A \longrightarrow \mathcal{P}(AP)$ is a function, called the labeling function, associating to each state $a \in A$ the set $L(a)$ of those atomic propositions in AP that hold in the state a.
- The semantics of the temporal logic LTL is defined by means of a satisfaction relation between a Kripke structure $\mathcal{A}$, a state $a \in A$, and an LTL formula $\varphi \in \mathrm{LTL}(AP)$:

$$\mathcal{A}, a \models \varphi \iff \mathcal{A}, \pi \models \varphi \quad \text{for all paths } \pi \text{ with } \pi(0) = a.$$

Kripke structures associated to rewrite theories

- Given a system module M specifying a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, we
  - choose a kind k in M as our kind of states;
  - define some state predicates Π and their semantics in a module, say M-PREDS, protecting M by means of the operation

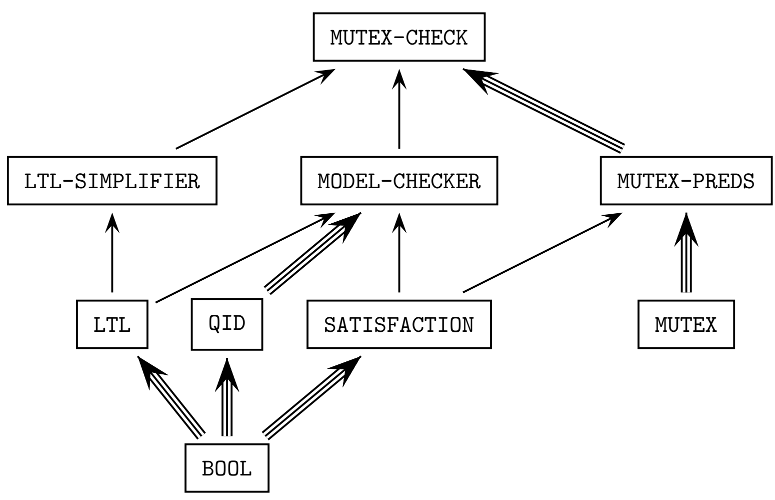
    ```
    op _|=_ : State Prop -> Bool .
    ```

    coming from the predefined SATISFACTION module.
- Then we get a Kripke structure

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi).$$

- Under some assumptions on M and M-PREDS, including that the set of states reachable from $[t]$ is finite, the relation $\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$ becomes decidable.

Model-checking modules



Crossing the river

- A shepherd needs to transport to the other side of a river
  - a wolf,
  - a lamb, and
  - a cabbage.
- He has only a boat with room for the shepherd himself and another item.
- The problem is that in the absence of the shepherd
  - the wolf would eat the lamb, and
  - the lamb would eat the cabbage.

Crossing the river

- The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located.
- Constants `left` and `right` represent the two sides of the river.
- Operation `change` is used to modify the corresponding attributes.
- Rules represent the ways of crossing the river that are allowed by the capacity of the boat.
- Properties define the good and bad states:
  - `success` characterizes the state in which the shepherd and his belongings are in the other side,
  - `disaster` characterizes the states in which some eating takes place.

Crossing the river

```
mod RIVER-CROSSING is
  sorts Side Group .

  ops left right : -> Side [ctor] .
  op change : Side -> Side .
  eq change(left) = right .
  eq change(right) = left .

  ops s w l c : Side -> Group [ctor] .
  op __ : Group Group -> Group [ctor assoc comm] .

  var S : Side .

  rl [shepherd] : s(S) => s(change(S)) .
  rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

Crossing the river

```
mod RIVER-CROSSING-PROP is
  protecting RIVER-CROSSING .
  including MODEL-CHECKER .
  subsort Group < State .

  op initial : -> Group .
  eq initial = s(left) w(left) l(left) c(left) .

  ops disaster success : -> Prop .

  vars S S' S'' : Side .

  ceq (w(S) l(S) s(S') c(S'') |= disaster) = true if S =/= S' .
  ceq (w(S'') l(S) s(S') c(S) |= disaster) = true if S =/= S' .
  eq (s(right) w(right) l(right) c(right) |= success) = true .
endm
```

Crossing the river

- The model checker only returns paths that are counterexamples of properties.
- To find a safe path we need to find a formula that somehow expresses the negation of the property we are interested in: a counterexample will then witness a safe path for the shepherd.
- If no safe path exists, then it is true that whenever success is reached a disastrous state has been traversed before:

```
<> success -> (<> disaster /\ ((~ success) U disaster))
```

  Note that this formula is equivalent to the simpler one

```
<> success -> ((~ success) U disaster)
```

- A counterexample to this formula is a safe path, completed so as to have a cycle.

Crossing the river

```
Maude> red modelCheck(initial,
        <> success -> (<> disaster /\ ((˜ success) U disaster))) .

result ModelCheckResult: counterexample(
    {s(left) w(left) l(left) c(left),'lamb}
    {s(right) w(left) l(right) c(left),'shepherd}
    {s(left) w(left) l(right) c(left),'wolf}
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'cabbage}
    {s(right) w(right) l(left) c(right),'shepherd}
    {s(left) w(right) l(left) c(right),'lamb}
    {s(right) w(right) l(right) c(right),'lamb}
    {s(left) w(right) l(left) c(right),'shepherd}
    {s(right) w(right) l(left) c(right),'wolf}
    {s(left) w(left) l(left) c(right),'lamb}
    {s(right) w(left) l(right) c(right),'cabbage}
    {s(left) w(left) l(right) c(left),'wolf},
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'lamb})
```

Reflection

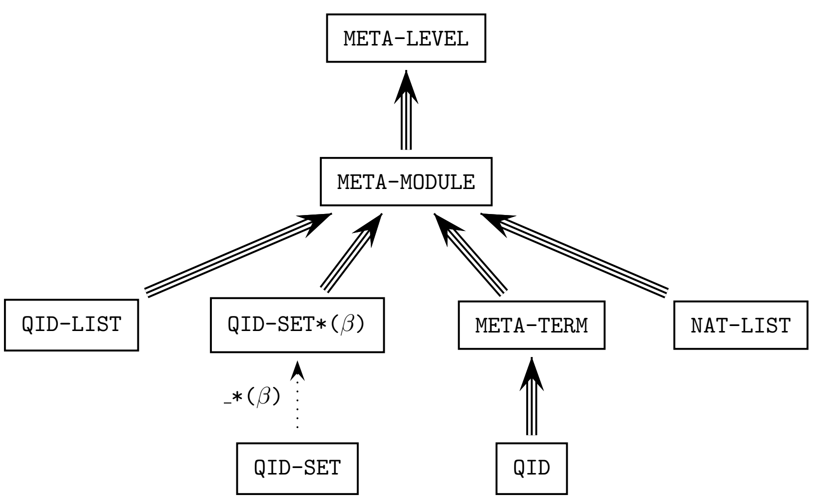
- Rewriting logic is reflective, because there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that:
  - we can represent any finitely presented rewrite theory $\mathcal{R}$ and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$,
  - then we have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

- Since $\mathcal{U}$ is representable in itself, we get a reflective tower

$$\begin{array}{c} \mathcal{R} \vdash t \rightarrow t' \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \\ \vdots \end{array}$$

Maude's metalevel



Maude's metalevel

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`:

- Maude terms are reified as elements of a data type `Term` in the module `META-TERM`;
- Maude modules are reified as terms in a data type `Module` in the module `META-MODULE`;
- operations `upModule`, `upTerm`, `downTerm`, and others allow moving between reflection levels;
- the process of reducing a term to canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
- the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;

Maude's metalevel

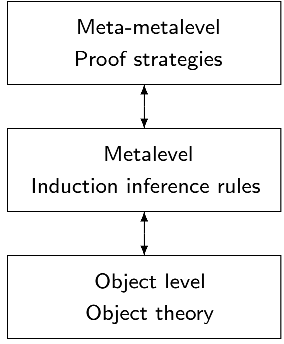
- the process of applying a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
- the process of applying a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
- the process of matching two terms is reified by built-in functions `metaMatch` and `metaXmatch`;
- the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`; and
- parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

Metaprogramming

- Programming at the metalevel: the metalevel equations and rewrite rules operate on representations of lower-level rewrite theories.
- Reflection makes possible many advanced metaprogramming applications, including
  - user-definable strategy languages,
  - language extensions by new module composition operations,
  - development of theorem proving tools, and
  - definition of translations between languages or logics within rewriting logic.
- Theorem provers and other formal tools have underlying inference systems that can be naturally specified and prototyped in rewriting logic. Furthermore, the strategy aspects of such tools and inference systems can then be specified by rewriting strategies.

Developing theorem proving tools

- Theorem-proving tools have a very simple reflective design in Maude.
- The inference system itself may perform theory transformations, so that the theories themselves must be treated as data.
- We need strategies to guide the application of the inference rules.
- Example: Inductive Theorem Prover (ITP).



Full Maude

- The systematic and efficient use of reflection through its predefined `META-LEVEL` module makes Maude remarkably extensible and powerful.
- Full Maude is an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible module algebra of parameterized modules and module composition operations, including parameterized views.
- Full Maude also provides special syntax for object-oriented modules supporting object-oriented concepts such as objects, messages, classes, and multiple class inheritance.

Full Maude

- Full Maude itself can be used as a basis for further extensions, by adding new functionality.
- Full Maude becomes a common infrastructure on top of which one can build other tools:
  - Church-Rosser and coherence checkers for Maude
  - declarative debuggers for Maude, for wrong and missing answers
  - Real-Time Maude tool for specifying and analyzing real-time systems
  - MSOS tool for modular structural operational semantics
  - Maude-NPA for analyzing cryptographic protocols
  - strategy language prototype

Advertising

Application areas

Application areas

- Models of concurrent computation
    - Equational programming
    - Lambda calculi
    - Petri nets
    - CCS and π-calculus
    - Actors
- Operational semantics of languages
    - Structural operational semantics (SOS)
    - Agent languages
    - Active networks languages
    - Mobile Maude
    - Hardware description languages

Application areas

- Logical framework and metatool
  - Linear logic
  - Translations between HOL and Nuprl theorem provers
  - Pure type systems
  - Open calculus of constructions
  - Tile logic
- Distributed architectures and components
  - UML diagrams and metamodels
  - Middleware architecture for composable services
  - Reference Model for Open Distributed Processing
  - Validation of OCL properties
  - Model management and model transformations

Application areas

- Specification and analysis of communication protocols
  - Active networks
  - Wireless sensor networks
  - FireWire leader election protocol
- Modeling and analysis of security protocols
  - Cryptographic protocol specification language CAPSL
  - MSR security specification formalism
  - Maude-NPA
- Real-time, biological, probabilistic systems
  - Real-Time Maude Tool
  - Pathway Logic
  - PMaude

From a satisfied user

In any case, I'd like to say thank you for the great job you have been doing with Full Maude. I find it to be incredibly useful. I've used Full Maude to model a distributed virtual memory system for TCP/IP networks, and there's a pretty good chance that this model will turn into real software that becomes part of the product of my employer. I have known Maude for a while, but that was the first time I actually used it to approach a real world problem. I was surprised how simple and straightforward the process turned out to be. I had a working prototype that exposed all tricky design decisions within less than a week. I've modeled software in Haskell before, and quite liked it, but I have to say that Full Maude is the best system I know so far. My favorite feature are parameterized views. Please know that your efforts are appreciated.

More satisfied users

I'm happy to inform you that with my coworker Marc Nieper-Wisskirchen, we successfully used your Maude program to implement the vertex algebra of operators on the cohomology of Hilbert schemes of points on surfaces. We obtained new results on the characteristic classes of some bundles. Our paper is published in the Journal on Mathematics and Computations (London Math. Soc.) and can be accessed at the following address:
http://www.lms.ac.uk/jcm/10/lms2006-045/
I hope this can be of some interest for you!
Best regards,
Samuel Boissiere
Universite de Nice, France

Structural operational semantics

- In general, an inference rule of the form $\dfrac{S_1 \ldots S_n}{S_0}$ can be mapped into a rewrite rule of the form

$$S_1 \ldots S_n \longrightarrow S_0 \quad \text{or} \quad S_0 \longrightarrow S_1 \ldots S_n$$

  that rewrites multisets of judgements S_i.

- In the operational semantics case, it is better to map an inference rule of the form

$$\frac{P_1 \to Q_1 \quad \ldots \quad P_n \to Q_n}{P_0 \to Q_0}$$

  to a conditional rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \textit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n,$$

  where the condition includes rewrites.

Executable semantic framework

- The gap between theory and practice was bridged in Alberto Verdejo's PhD thesis and papers with several case studies:
  - functional language (evaluation and computation semantics, including an abstract machine),
  - imperative language (evaluation and computation semantics),
  - nondeterministic language (computation semantics),
  - Kahn's functional language Mini-ML (evaluation or natural semantics),
  - Milner's CCS (with strong and weak transitions),
  - Full LOTOS (including ACT ONE data type specifications).
- The same techniques were used by other authors for Milner's π-calculus and other languages.

JavaFAN (Java Formal ANalysis)

- Executable rewriting logic semantics of both Java and JVM Bytecode (except for the libraries).
- To keep the framework user-friendly, JavaFAN wraps the Maude specifications and accepts Java or JVM code from the user as input.
- The formal semantic specifications become interpreters to run Java programs on the source code level and/or on the bytecode level.
- Using the underlying features of Maude, JavaFAN can be used to
  - symbolically execute multithreaded programs,
  - detect safety violations searching through an unbounded state space, and
  - verify finite state programs by explicit state model checking.
- JavaFAN's efficiency compares well with other Java analysis tools.
- One of the reasons for efficiency: use of equations instead of rules to express the semantics of deterministic features.

Rewriting logic semantics project

- "The broad goal of the project is to develop a tool-supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on rewriting logic."
- Some fundamental references:
  - J. Meseguer and G. Rosu, Rewriting logic semantics: from language specifications to formal analysis tools, *IJCAR 2004*, Springer LNCS 3097.
  - J. Meseguer and G. Rosu, The rewriting logic semantics project, *Theoretical Computer Science*, 2007.
  - T. F. Serbanuta, G. Rosu, and J. Meseguer, A rewriting logic approach to operational semantics, *Information and Computation*, 2009.

"Ecumenical" approach

- Embedding operational semantics styles in rewriting logic
- "Each of these language definitional styles can be faithfully captured as an RLS theory: there is a one-to-one correspondence between computational steps in the original language definition and computational steps in the corresponding RLS theory"
  - Big-step operational semantics (natural semantics)
  - Small-step operational semantics (transition semantics)
  - Modular structural operational semantics (MSOS)
  - Reduction semantics with evaluation contexts
  - Chemical abstract machine
  - First-order continuation-based semantics
- "RLS does not force or pre-impose any given language definitional style, and its exibility and ease of use makes RLS an appealing framework for exploring new definitional styles."

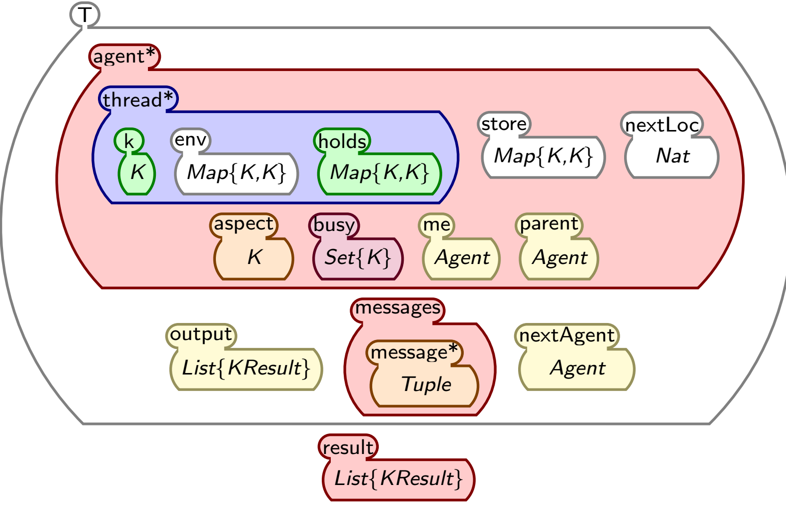
K framework

- Two main technical reports by Grigore Rosu (and many other papers):
  - K: A Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation
  - K: A Rewriting-Based Framework for Computations
- Ambitious features:
  - Methodology to define languages ...
  - ... and type checkers, abstract interpreters, domain-specific checkers, etc.
  - Arbitrarily complex language features
  - Modular (crucial for scalability and reuse)
  - Generic (multi-language and multi-paradigm)
  - Support for non-determinism and concurrency
  - Efficient executability
  - State-exploration capabilities (e.g., finite-state model-checking)
  - Formal semantics

K basics: computations

- K is based around concepts from Rewriting Logic Semantics, with some intuitions from Chemical Abstract Machines (CHAMs) and Reduction Semantics (RS).
- Abstract computational structures contain context needed to produce a future computation (like continuations).
- Computations take place in the context of a configuration.
- Configurations are hierarchical (like in RLS), made up of K cells.
- Each cell holds specific piece of information: computation, environment, store, etc.
- Two regularly used cells:
  - $\top$ (*top*), representing entire configuration
  - k, representing current computation
- Cells can be repeated (e.g., multiple computations in a concurrent language).

K configuration: nested cells

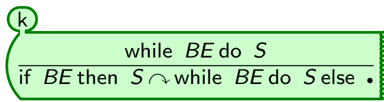


K basics: equations and rules

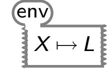
- Cell $\mathtt{k}$ is made up of a list of computational tasks separated by $\curvearrowright$, like $t_1 \curvearrowright t_2 \curvearrowright ... \curvearrowright t_n$.
- Intuition from CHAMs: language constructs can heat (break apart into pieces for evaluation) and cool (form back together).
- Represented using $\rightleftharpoons$, like $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \Box + a_2$.
- A heating/cooling pair can be seen as an equation.
- Intuition from RS: $\Box$ can be seen as similar to evaluation contexts, marking the location where evaluation can occur.
- Computations are defined used equations and rules
- Heating/Cooling Rules (Structural Equations): manipulate term structure, non-computational, reversible, can think of as just equations
- Rules: computational, not reversible, may be concurrent

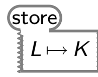
K basics: equations and rules

EQUATION:

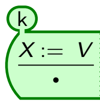


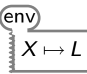
RULE:

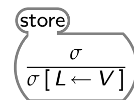




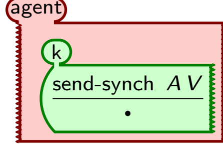
RULE:

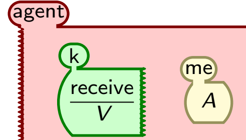






RULE:





K framework: some accomplishments

- Embeddings of different operational semantics styles in K.
- Semantics of many languages, toy and real.
- Implementation in Maude: K-Maude.
- Static checking of units of measurement in C.
- Runtime verification of C memory safety.
- Definition of type systems and type inference.
- Compilation of language definitions into competitive interpreters (in OCaml).

Unification

- Given terms t and u, we say that t and u are unifiable if there is a substitution σ such that $\sigma(t) \equiv \sigma(u)$.
- Given an equational theory A and terms t and u, we say that t and u are unifiable modulo A if there is a substitution σ such that $\sigma(t) \equiv_A \sigma(u)$.
- Maude 2.4 supports at the core level and at the metalevel order-sorted equational unification modulo combinations of `comm` and `assoc comm` attributes as well as free function symbols.

Narrowing

- A term t narrows to a term t' using a rule $l \Rightarrow r$ in R and a substitution σ if
  1. there is a subterm $t|_p$ of t at a nonvariable position p of t such that l and $t|_p$ are unifiable via σ, and
  2. $t' = \sigma(t[r]_p)$ is obtained from $\sigma(t)$ by replacing the subterm $\sigma(t|_p) \equiv \sigma(l)$ with the term $\sigma(r)$.
- Narrowing can also be defined modulo an equational theory A.
- Full Maude 2.4 supports a version of narrowing modulo with simplification, where each narrowing step with a rule is followed by simplification to canonical form with the equations.
- There are some restrictions on the allowed rules; for example, they cannot be conditional.

Narrowing reachability analysis

Narrowing can be used as a general deductive procedure for solving reachability problems of the form

$$(\exists \vec{x})\ t_1(\vec{x}) \longrightarrow t'_1(\vec{x}) \ \wedge \ldots \wedge \ t_n(\vec{x}) \longrightarrow t'_n(\vec{x})$$

in a given rewrite theory.

- The terms t_i and t'_i denote sets of states.
- For what subset of states denoted by t_i are the states denoted by t'_i reachable?
- No finiteness assumptions about the state space.
- Sound and complete for topmost rewrite theories.

Maude-NPA

- Maude-NPA (NRL Protocol Analyzer) is a tool to find or prove the absence of attacks using backwards search.
- It analyzes infinite state systems:
  - Active Dolev-Yao intruder,
  - No abstraction or approximation of nonces,
  - Unbounded number of sessions.
- Intruder and honest protocol transitions are represented by a variant of strand space model: strands with a marker denoting the current state.
  - Searches backwards through strands from final state.
  - Set of rewrite rules governs how search is conducted.
  - Sensitive to past and future.

Maude-NPA

- Maude-NPA supports as equations the algebraic properties of the cryptographic functions used in the given protocol:
  - explicit encryption and decryption,
  - exclusive or,
  - modular exponentiation,
  - homomorphism.
- Reasoning modulo such algebraic properties is very important, since it is well-known that some protocols that can be proved secure under the standard Dolev-Yao model, in which the cryptographic functions are treated as a "black box," can actually be broken by an attacker that makes clever use of the algebraic properties of the cryptographic functions of the protocol.

Maude-NPA

- Use rewriting logic as general theoretical framework:
  - protocols and intruder rules are specified as rewrite rules,
  - crypto properties as oriented equational properties and axioms.
- Use narrowing modulo equational theories in two ways:
  - as a symbolic reachability analysis method,
  - as an extensible equational unification method.
- Combine with state reduction techniques of NRL Protocol Analyzer (grammars, optimizations, etc.) by C. Meadows.
- Implement in Maude programming environment:
  - rewriting logic provides theoretical framework and understanding,
  - Maude implementation provides tool support.

Basic structure of Maude-NPA

- Each local execution, or session of a honest principal is represented by a sequence of positive and negative terms called a strand.
  - Negative terms stand for received messages
  - Positive terms stand for sent messages
  - Example: $[\ pke(B,N_A;A)^+,\ pke(A,N_A;N_B)^-,\ pke(B,N_B)^+\ ]$
- Each intruder computation is also represented by a strand
  - Example: $[X^-, pke(A,X)^+]$ and $[X^-, Y^-, (X;Y)^+]$
- Modified strand notation: a marker denotes the current state
  - Example: $[\ pke(B,N_A;A)^+ \mid pke(A,N_A;N_B)^-,\ pke(B,N_B)^+\ ]$
- Sensitive to past and future.
- No data or nonce abstraction.

Basic structure of Maude-NPA

To execute a protocol, associate to it a rewrite theory on sets of strands. $\mathcal{I}$ informally denotes the set of terms known by the intruder, and K the facts known or unknown by the intruder. Then,

- $[\ L \mid M^{-}, L'\ ]\ \&\ \{M \in \mathcal{I}, K\} \rightarrow [\ L, M^{-} \mid L'\ ]\ \&\ \{M \in \mathcal{I}, K\}$
  Moves input messages into the past.
- $[\ L \mid M^{+}, L'\ ]\ \&\ \{K\} \rightarrow [\ L, M^{+} \mid L'\ ]\ \&\ \{K\}$
  Moves output messages that are not read into the past.
- $[\ L \mid M^{+}, L'\ ]\ \&\ \{M \notin \mathcal{I}, K\} \rightarrow [\ L, M^{+} \mid L'\ ]\ \&\ \{M \in \mathcal{I}, K\}$
  Joins output message with term in intruder knowledge.

Some work in progress

- Connecting Maude to HETS, heterogeneous verification system developed at Bremen, Germany, which is already connected to theorem provers like Isabelle.
- Semantics of modeling, real-time, and hardware languages.
- Modeling of cyberphysical systems (avionics, medical systems, ...).
- Secure-by-design browsers.
- More and better equational unification algorithms.
- Temporal logic of rewriting.
- Matching logic on top of K framework.
- Multicore reimplementation of Maude.

Many thanks

- Maude team:
  - José Meseguer
  - Francisco Durán
  - Steven Eker
  - Manuel Clavel
  - Carolyn Talcott
  - Pat Lincoln
- Very helpful colleagues:
  - Santiago Escobar
  - Grigore Roșu
- PADL 2010 organizers:
  - Manuel Carro
  - Ricardo Peña