Unification and Narrowing in Maude 3.1

Santiago Escobar
Valencian Research Institute for Artificial Intelligence (VRAIN)
Universitat Politècnica de València Spain

UNIVERSIDAD DE MÁLAGA
SRI International
ILLINOIS UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
UNIVERSIDAD COMPLUTENSE MADRID

Outline

Outline

Why rewriting logic?

1. Models and formal specification are easily written in Maude (simplicity, expressiveness, and performance)
2. Rewriting modulo associativity, commutativity and identity
3. Differentiation between concurrent and functional fragments of a model
4. Order-sorted and parameterized specifications
5. Infrastructure for formal analysis and verification (including search command, LTL model checker, theorem prover, etc.)
6. Reflection (meta-modeling, symbolic execution, building tools)
7. Application areas:
   - Models of computation (λ-calculi, π-calculus, petri nets, CCS),
   - Programming languages (C, Java, Haskell, Prolog),
   - Distributed algorithms and systems (security protocols, real-time, probabilistic),
   - Biological systems

Why adding logical features to Rewriting Logic?

1. Logical features were included in preliminary designs of the language (80's) but never implemented in Maude
2. Automated reasoning capabilities by adding logical variables
3. Differentiation between concurrent and functional fragments of a model is lifted to differentiation between symbolic models and equational reasoning.
4. Unification and Narrowing modulo combinations of A,C,U
5. Infrastructure for formal analysis and verification lifted:
   - from equational reduction to equational unification,
   - from search to symbolic reachability,
   - from LTL model checker to logical LTL model checker,
   - from theorem proving to narrowing-based theorem proving,
   - from SMT solving to variant-based SMT solving.

   ⋮

Outline

What have we done!!

- Maude 2.4 (2009)
  - Built-in Unification: free or associative-commutative (AC)
  - Narrowing-based search: rules modulo axioms (no equations).
- Maude 2.6 (2011)
  - Built-in Unification: free, C, AC, or ACU (AC + identity)
  - Variant Unification: Restricted equations modulo axioms.
  - Narrowing-based search: rules modulo equations and axioms.
- Maude 2.7 (2015)
  - Built-in Unification: free, C, AC, or ACU, CU, U, Ul, Ur
  - Built-in Variant unification: wide class of equational theories.
  - Narrowing-based search: rules modulo equations and axioms.
- Maude 2.7.1 (2016)
  - Built-in Unification: previous cases + associativity
  - Built-in Variant unification: modulo all combinations
  - Narrowing-based search: modulo all combinations
- Maude 3.0 (2019) Built-in Narrowing-based search: modulo all combinations
- Maude 3.1 (2020) Minimal (equational) unifiers, better unification modulo associavity

Outline

Rewriting logic in a nutshell

A rewrite theory is

$\mathcal{R} = (\Sigma, Ax \uplus E, R)$, with:

1. (Σ, R) a set of rewrite rules of the form $t \to s$
   (i.e., system transitions)
2. $(\Sigma, Ax \uplus E)$ a set of equational properties of the form $t = s$
   (i.e., E are equations and Ax are axioms such as ACU)

Intuitively, $\mathcal{R}$ specifies a concurrent system, whose states are elements of the initial algebra $T_{\Sigma/(Ax \uplus E)}$ specified by $(\Sigma, Ax \uplus E)$, and whose concurrent transitions are specified by the rules R.

Rewriting logic in a nutshell

```
mod VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty :  -> Money .
  op __ :  Money Money -> Money [assoc comm id:  empty] .
  subsort Money Item < Marking .
  op __ :  Marking Marking -> Marking [assoc comm id:  empty] .
  op <_> :  Marking -> State .
  ops $ q :  -> Coin .
  ops cookie cap :  -> Item .
  var M : Marking .
  rl [add-$] :  < M > => < M $ > .
  rl [add-q] :  < M > => < M q > .
  rl [buy-c] :  < M $ > => < M cap > .
  rl [buy-a] :  < M $ > => < M cookie q > .
  eq [change]:  q q q q = $ [variant] .
endm
```

Rewriting logic in a nutshell

```
Maude> search <$ q q q> =>! <cookie cap St:State> .
Solution 1 (state 3)
states:  6 rewrites:  5 in 0ms cpu (0ms real)
St:State --> null
No more solutions.
states:  6 rewrites:  5 in 0ms cpu (1ms real)
Maude> show path 3 .
state 0, State: < $ q q q >
===[ rl St $ => St cookie q .  ]===>
state 2, State: < $ cookie >
===[ rl St $ => St cap .  ]===>
state 3, State: < cap cookie >
```

Rewriting modulo

Rewriting is

Given $(\Sigma, Ax \uplus E, R)$, $t \to_{R,(Ax\uplus E)} s$ if there is

- a non-variable position $p \in Pos(t)$;
- a rule $l \to r$ in R;
- a matching σ (E-normalized and modulo Ax) such that $t|_p =_{(Ax\uplus E)} \sigma(l)$, and $s = t[\sigma(r)]_p$.

Ex: `< $ q q q >` $\to$ `< $ cookie >`
using "`rl < M $ > => < M cookie q > .`"
modulo AC of symbol "`__`"

Ex: `< q q q q >` $\to$ `< cap >`
using "`rl < M $ > => < M cap > .`"
modulo simplification with `q q q q = $` and AC of symbol "`__`"

Narrowing modulo

Narrowing is

Given $(\Sigma, Ax \uplus E, R)$, $t \leadsto_{\sigma,R,(Ax \uplus E)} s$ if there is

- a non-variable position $p \in Pos(t)$;
- a rule $l \to r$ in R;
- a unifier σ (E-normalized and modulo Ax) such that $\sigma(t|_p) =_{(Ax \uplus E)} \sigma(l)$, and $s = \sigma(t[r]_p)$.

Ex: `< X q q >` $\leadsto$ `< $ cookie >`
using "`rl < M $ > => < M cookie q > .`"
using substitution $\{X \mapsto$ `$ q`$\}$ modulo AC of symbol "`_`"

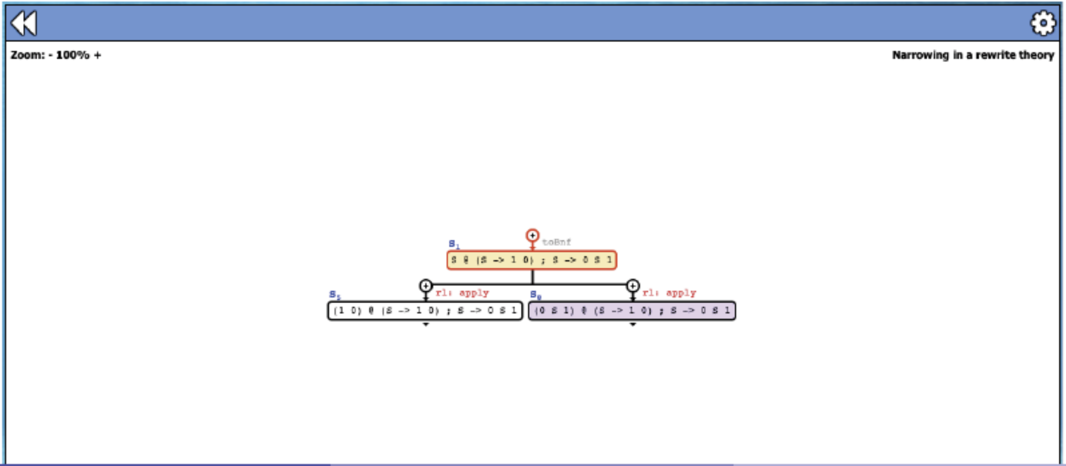
Ex: `< X q q >` $\leadsto$ `< cap >`
using "`rl < M $ > => < M cap > .`"
using substitution $\{X \mapsto$ `q q`$\}$
modulo simplification with `q q q q = $` and AC of symbol "`_`"

Outline

Symbolic Analysis of Maude Theories (Narval tool)

Four execution modalities are supported by Narval: (i) Rewriting mode (rules&equations), (ii) Narrowing with equations, (iii) Narrowing with rules&equations, (iv) Equational unification

`http://safe-tools.dsic.upv.es/narval`



Outline

Unification modulo axioms

Definition

Given equational theory (Σ, Ax), an *Ax-unification problem* is

$$t \stackrel{?}{=} t'$$

An *Ax-unifier* is an order-sorted substitution σ s.t.

$$\sigma(t) =_{Ax} \sigma(t')$$

Decidability

- at most one mgu (syntactic unification, i.e., empty theory)
- a finite number (associativity–commutativity)
- an infinite number (associativity)

Admissible Theories

Maude provides order-sorted Ax-unification algorithm for all order-sorted theories $(\Sigma, E \cup Ax, R)$ s.t. Σ is preregular modulo Ax and axioms Ax are:

1. arbitrary function symbols and constants with no attributes;
2. `iter` equational attribute declared for some unary symbols;
3. "`comm`", "`assoc`", "`assoc comm`", "`assoc comm id:`", "`comm id:`", "`assoc id:`", "`id:`", "`left id:`", or "`right id:`" attributes declared for some binary function symbols but no other equational attributes can be given for such symbols.

Unification Command in Maude

Maude provides a Ax-unification command of the form:

$$
\begin{array}{l}
\texttt{unify [} n \texttt{] in } \langle \mathit{ModId} \rangle \texttt{ :} \\
\quad \langle \mathit{Term\text{-}1} \rangle \texttt{ =? } \langle \mathit{Term'\text{-}1} \rangle \texttt{ /\textbackslash\ ... /\textbackslash\ } \langle \mathit{Term\text{-}k} \rangle \texttt{ =? } \langle \mathit{Term'\text{-}k} \rangle \texttt{ .} \\
\texttt{irredundant unify [} n \texttt{] in } \langle \mathit{ModId} \rangle \texttt{ :} \\
\quad \langle \mathit{Term\text{-}1} \rangle \texttt{ =? } \langle \mathit{Term'\text{-}1} \rangle \texttt{ /\textbackslash\ ... /\textbackslash\ } \langle \mathit{Term\text{-}k} \rangle \texttt{ =? } \langle \mathit{Term'\text{-}k} \rangle \texttt{ .}
\end{array}
$$

- ModId is the name of the module
- n is a bound on the number of unifiers
- new variables are created as `#n:Sort`
- Implemented at the core level of Maude (C++)

AC-Unification in Maude

```
Maude> unify [100] in NAT :
          X:Nat + X:Nat + Y:Nat =? A:Nat + B:Nat + C:Nat .

Solution 1
X:Nat --> #1:Nat + #2:Nat + #3:Nat + #5:Nat + #6:Nat + #8:Nat
Y:Nat --> #4:Nat + #7:Nat + #9:Nat
A:Nat --> #1:Nat + #1:Nat + #2:Nat + #3:Nat + #4:Nat
B:Nat --> #2:Nat + #5:Nat + #5:Nat + #6:Nat + #7:Nat
C:Nat --> #3:Nat + #6:Nat + #8:Nat + #8:Nat + #9:Nat
...

Solution 100
X:Nat --> #1:Nat + #2:Nat + #3:Nat + #4:Nat
Y:Nat --> #5:Nat
A:Nat --> #1:Nat + #1:Nat + #2:Nat
B:Nat --> #2:Nat + #3:Nat
C:Nat --> #3:Nat + #4:Nat + #4:Nat + #5:Nat
```

ACU-Unification in Maude

```
Maude> unify [100] in QID-SET : X:QidSet , X:QidSet , Y:QidSet =? A:QidSet , B:QidSet , C:QidSet .
unify [100] in QID-SET : X:QidSet, X:QidSet, Y:QidSet =? A:QidSet, B:QidSet, C:QidSet .
Decision time: 0ms cpu (1ms real)

Solution 1
X:QidSet --> empty
Y:QidSet --> empty
A:QidSet --> empty
B:QidSet --> empty
C:QidSet --> empty

Solution 2
X:QidSet --> #1:QidSet
Y:QidSet --> empty
A:QidSet --> #1:QidSet, #1:QidSet
B:QidSet --> empty
C:QidSet --> empty
```

Irredundant Unification in Maude

```
Maude> unify in UNIF-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> $
Y:Marking --> q q

Unifier 2
X:Marking --> $ #1:Marking
Y:Marking --> q q #1:Marking
```

```
Maude> irredundant unify in UNIF-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> $ #1:Marking
Y:Marking --> q q #1:Marking
```

Identity Unification in Maude

```
mod LEFTID-UNIFICATION-EX is
    sorts Magma Elem . subsorts Elem < Magma .
    op __ : Magma Magma -> Magma [left id: e] .
    ops a b c d e : -> Elem .
endm
```

```
Maude> unify in LEFTID-UNIFICATION-EX : X:Magma a =? (Y:Magma a) a .
Solution 1                Solution 2
X:Magma --> a             X:Magma --> #1:Magma a
Y:Magma --> e             Y:Magma --> #1:Magma
```

```
Maude> unify in LEFTID-UNIFICATION-EX : a X:Magma =? (a a) Y:Magma .
No unifier.
```

```
mod COMM-ID-UNIFICATION-EX is
    sorts Magma Elem . subsorts Elem < Magma .
    op __ : Magma Magma -> Magma [comm id: e] .
    ops a b c d e : -> Elem .
endm
```

```
Maude> unify in COMM-ID-UNIFICATION-EX : X:Magma a =? (Y:Magma a) a .
Solution 1       Solution 2                 Solution 3
X:Magma --> a    X:Magma --> a #1:Magma     X:Magma --> a
Y:Magma --> e    Y:Magma --> #1:Magma       Y:Magma --> e
```

A-Unification in Maude

```
Maude> unify in UNIFICATION-EX4 : X:NList : Y:NList : Z:NList =? P:NList : Q:NList .

Solution 1
X:NList --> #1:NList : #2:NList
Y:NList --> #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList
Q:NList --> #2:NList : #3:NList : #4:NList

Solution 2
X:NList --> #1:NList
Y:NList --> #2:NList : #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList : #2:NList
Q:NList --> #3:NList : #4:NList

Solution 3
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList : #4:NList
P:NList --> #1:NList : #2:NList : #3:NList
Q:NList --> #4:NList
```

```
Unifier 4
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList : #2:NList
Q:NList --> #3:NList

Unifier 5
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList
Q:NList --> #2:NList : #3:NList
```

Incomplete A-Unification in Maude

Possible warnings and situations:

- Associative unification using cycle detection.
- Associative unification algorithm detected an infinite family of unifiers.
- Associative unification using depth bound of 5.
- Associative unification algorithm hit depth bound.

Example:

```
Maude> unify in UNIFICATION-EX4 : 0 : X:NList =? X:NList : 0 .
Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

Solution 1
X:NList --> 0
Warning: Some unifiers may have been missed due to incomplete
unification algorithm(s).
```

AU-Unification in Maude

```
Maude> irredundant unify in UNIFICATION-EX5 :
        X:NList : Y:NList : Z:NList =? P:NList : Q:NList .
Decision time: 2ms cpu (2ms real)

Unifier 1
X:NList --> #3:NList : #4:NList
Y:NList --> #1:NList
Z:NList --> #2:NList
P:NList --> #3:NList
Q:NList --> #4:NList : #1:NList : #2:NList

Unifier 2
X:NList --> #1:NList
Y:NList --> #3:NList : #4:NList
Z:NList --> #2:NList
P:NList --> #1:NList : #3:NList
Q:NList --> #4:NList : #2:NList

Unifier 3
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #4:NList : #3:NList
P:NList --> #1:NList : #2:NList : #4:NList
Q:NList --> #3:NList
```

AU fewer unifiers than A (5 vs 3) & unify returns many more than irredundant unify (32 vs 3)

Outline

Narrowing-based Equational Unification

Definition

Given an order-sorted equational theory $(\Sigma, Ax \uplus E)$ and $t \stackrel{?}{=} t'$, an $(Ax \uplus E)$-unifier is an order-sorted subst. σ s.t. $\sigma(t) =_{Ax \uplus E} \sigma(t')$.

When $Ax = \emptyset$ and E convergent TRS

Narrowing provides a complete (but semi-decidable) E-unification algorithm [Hullot80]. e.g. cancellation $d(K, e(K, M)) = M$.

When $Ax \neq \emptyset$ and E convergent and coherent TRS modulo Ax

Narrowing provides a complete (but semi-decidable) E-unification algorithm [Jouannaud-Kirchner-Kirchner-83] e.g. exclusive-or eq $X * 0 = X$, eq $X * X = 0$ symbol * being AC

Narrowing-based Equational Unification

Decidable Classes of Equational Theories

Narrowing is very inefficient and may not terminate.
Narrowing strategies for classes of equational theories.

When $Ax = \emptyset$

Basic narrowing strategy [Hullot80] is complete for normalized substitutions.
Cases where basic narrowing terminates have been studied [Alpuente-Escobar-Iborra-TCS09].

When $Ax \neq \emptyset$

Folding variant-narrowing [Escobar-Meseguer-Sasse-JLAP12] is the optimal strategy for equational unification.

From equational reduction to variants (1/4)

E,Ax-variant

Given a term t and an equational theory $Ax \uplus E$, (t', θ) is an E,Ax-variant of t if $\theta(t){\downarrow_{E,Ax}} =_{Ax} t'$ [Comon-Delaune-RTA05]

Exclusive Or

$$
\begin{array}{rc}
X \oplus 0 \to X & X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z \\
X \oplus X \to 0 & X \oplus Y = Y \oplus X \\
X \oplus X \oplus Y \to Y & (\text{axioms: } Ax)
\end{array}
$$

Computed Variants

For $X \oplus X$: $(0, id), (0, \{X \mapsto a\}), (0, \{X \mapsto a \oplus b\}), \ldots$

From equational reduction to variants (2/4)

Finite and complete set of E,Ax-variants

A preorder relation of generalization between variants provides a notion of most general variant.

Computed Variants

For $X \oplus Y$ there are 7 most general E,Ax-variants

1. $(X \oplus Y, id)$
2. $(0, \{X \mapsto U, Y \mapsto U\})$
3. $(Z, \{X \mapsto 0, Y \mapsto Z\})$
4. $(Z, \{X \mapsto Z \oplus U, Y \mapsto U\})$
5. $(Z, \{X \mapsto Z, Y \mapsto 0\})$
6. $(Z, \{X \mapsto U, Y \mapsto Z \oplus U\})$

From equational reduction to variants (3/4)

Finite Variant Property

Theory has FVP if finite number of most general variants for every term.

Common

- Cryptographic Security Protocols: Public or shared encryption, Exclusive Or, Abelian groups, Diffie-Hellman
- Satisfiability Modulo Theories Natural Presburger Arithmetic, Integer Presburger Arithmetic, Lists, Sets

Used in application areas

Equational Unification, Logical Model Checking, Cyber-Physical systems, Partial evaluation, Confluence tools, Termination tools, Theorem provers

From equational reduction to variants (4/4)

Test for FVP

Whether a theory has FVP is undecidable in general, though there are approximations techniques.

Computing most general variants

Given a theory that has FVP, it is possible to compute all the most general variants by using the Folding Variant Narrowing Strategy (Escobar et al. 2012)

Variant Command in Maude

Maude provides variant generation:

$$\texttt{get variants [} n \texttt{] in } \langle \mathit{ModId} \rangle \texttt{ : } \langle \mathit{Term} \rangle \texttt{ .}$$
$$\texttt{get irredundant variants [} n \texttt{] in } \langle \mathit{ModId} \rangle \texttt{ : } \langle \mathit{Term} \rangle \texttt{ .}$$

- ModId is the name of the module
- n is a bound on the number of variants
- new variables are created as `#n:Sort` and `%n:Sort`
- Implemented at the core level of Maude (C++)
- Folding variant narrowing strategy is used internally
- Terminating if Finite Variant Property
- Incremental output if not Finite Variant Property
- Irredundant version only if Finite Variant Property

Exclusive-or Variants

```
fmod EXCLUSIVE-OR is
  sorts Nat NatSet .  subsort Nat < NatSet .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op mt : -> NatSet .
  op _*_ : NatSet NatSet -> NatSet [assoc comm] .
  vars X Z : [NatSet] .
  eq [idem] :     X * X = mt    [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
  eq [id] :       X * mt = X    [variant] .
endfm
```

```
Maude> get variants in EXCLUSIVE-OR : X * Y .
Variant 1                                            Variant 7
[NatSet]: #1:[NatSet] * #2:[NatSet]   .........  [NatSet]: %1:[NatSet]
X --> #1:[NatSet]                                    X -->  %1:[NatSet]
Y --> #2:[NatSet]                                    Y --> mt
```

Abelian Group Variants

```
fmod ABELIAN-GROUP is
  sorts Elem .
  op _+_ : Elem Elem -> Elem [comm assoc] .
  op -_ : Elem -> Elem .
  op 0 : ->  Elem .
  vars X Y Z : Elem  .
  eq X + 0 = X [variant] .
  eq X + (- X) = 0 [variant] .
  eq X + (- X) + Y = Y [variant] .
  eq - (- X) = X [variant] .
  eq - 0 = 0 [variant] .
  eq (- X) + (- Y) = -(X + Y) [variant] .
  eq -(X + Y) + Y = - X [variant] .
  eq -(- X + Y) = X + (- Y) [variant] .
  eq (- X) + (- Y) + Z  = -(X + Y) + Z [variant] .
  eq -(X + Y) + Y + Z = (- X) + Z [variant] .
endfm
```

```
Maude> get variants in ABELIAN-GROUP : X + Y .
Variant 1                                          Variant 47
Elem: #1:Elem + #2:Elem    ................        Elem: - (%2:Elem + %3:Elem)
X --> #1:Elem                                      X --> %4:Elem + - (%1:Elem + %2:Elem)
Y --> #2:Elem                                      Y --> %1:Elem + - (%3:Elem + %4:Elem)
```

Incremental Variant Generation

```
fmod NAT-VARIANT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  eq [base] : 0 + Y = Y [variant] .
  eq [ind] :  s(X) + Y = s(X + Y) [variant] .
endfm
```

```
Maude> get variants in NAT-VARIANT : s(0) + X .
Variant 1
Nat: s(#1:Nat)
X --> #1:Nat
```

```
Maude> get variants [10] in NAT-VARIANT : X + s(0) .
Variant 1                                                    Variant 10
Nat: #1:Nat + s(0)    ....................................   Nat: s(s(s(s(s(0)))))
X --> #1:Nat                                                 X --> s(s(s(s(0))))
```

Infinite!!!

Outline

Admissible Theories

Maude provides order-sorted $Ax \uplus E$-unification algorithm for all order-sorted theories $(\Sigma, Ax, \vec{E})$ s.t.

1. Maude has an Ax-unification algorithm,
2. E equations specified with the `eq` and `variant` keywords.
3. E is unconditional, convergent, sort-decreasing and coherent modulo Ax.
4. The `owise` feature is not allowed.

Equational Unification Command in Maude

Maude provides a $(Ax \uplus E)$-unification command of the form:

$$\texttt{variant unify [}\ n\ \texttt{] in}\ \langle \mathit{ModId} \rangle\ \texttt{:}$$
$$\langle \mathit{Term\text{-}1} \rangle\ \texttt{=?}\ \langle \mathit{Term'\text{-}1} \rangle\ \texttt{/\textbackslash}\ \ldots\ \texttt{/\textbackslash}\ \langle \mathit{Term\text{-}k} \rangle\ \texttt{=?}\ \langle \mathit{Term'\text{-}k} \rangle\ \texttt{.}$$
$$\texttt{filtered variant unify [}\ n\ \texttt{] in}\ \langle \mathit{ModId} \rangle\ \texttt{:}$$
$$\langle \mathit{Term\text{-}1} \rangle\ \texttt{=?}\ \langle \mathit{Term'\text{-}1} \rangle\ \texttt{/\textbackslash}\ \ldots\ \texttt{/\textbackslash}\ \langle \mathit{Term\text{-}k} \rangle\ \texttt{=?}\ \langle \mathit{Term'\text{-}k} \rangle\ \texttt{.}$$

- ModId is the name of the module
- n is a bound on the number of unifiers
- new variables are created as `#n:Sort` and `%n:Sort`
- Implemented at the core level of Maude (C++)
- Terminating if Finite Variant Property
- Incremental output if not Finite Variant Property

Variant-based Unification Command in Maude

```
fmod NAT-VARIANT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  eq [base] : 0 + Y = Y [variant] .
  eq [ind] :  s(X) + Y = s(X + Y) [variant] .
endfm
```

```
Maude> variant unify in NAT-VARIANT : s(0) + X =? s(s(s(0)))  .

Unifier #1
X --> s(s(0))

No more unifiers.

Maude> variant unify [1] in NAT-VARIANT : X + s(0) =? s(s(s(0)))  .

Unifier #1
X --> s(s(0))
```

Infinite!!!

Filtered Variant-based Unification in Maude

```
Maude> variant unify in VARIANT-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> $ %1:Marking
Y:Marking --> q q %1:Marking

Unifier 2
X:Marking --> q q #1:Marking
Y:Marking --> #1:Marking

Maude> filtered variant unify in VARIANT-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> q q #1:Marking
Y:Marking --> #1:Marking
```

Incomplete Variant Unification (due to assoc)

```
Maude> variant unify in VARIANT-UNIFICATION-ASSOC :
    head(L) =? last(L) /\ prefix(L) =? tail(L)  .

Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

Unifier #1
L --> %1:Nat : %1:Nat : %1:Nat

Unifier #2
L --> %1:Nat : %1:Nat

No more unifiers.
Warning: Some unifiers may have been missed due to incomplete unification algorithm(s).
```

```
eq head(E : L) = E [variant] .
eq tail(E : L) = L [variant] .
eq prefix(L : E) = L [variant] .
eq last(L : E) = E [variant] .
```

Outline

Symbolic reachability analysis in rewrite theories

- Given $(\Sigma, E \cup Ax, R)$ as a concurrent system, a symbolic reachability problem is

$$(\exists X)\ t \longrightarrow^{*} t'$$

- Narrowing provides a sound and complete method for topmost theories.
- Narrowing with R modulo $Ax \uplus E$ requires $Ax \uplus E$-unification at each narrowing step
- Narrowing can be also used for logical model checking

Narrowing in Maude

Narrowing generalizes term rewriting by allowing free variables in terms and by performing unification instead of matching in order to (non–deterministically) reduce a term.

1. Narrowing + simplification (for built-in operators and equational simplification)
2. Frozen arguments, similar to the context-sensitive narrowing
3. Extra variables in right hand sides of the rules for functional logic programming features (e.g. constraint programming and instantiation search).

Narrowing Search Command in Maude

Narrowing-based search command of the form:

$$\texttt{vu-narrow [}\ n,\ m\ \texttt{] in}\ \langle \mathit{ModId} \rangle\ \texttt{:}\ \langle \mathit{Term\text{-}1} \rangle\ \langle \mathit{SearchArrow} \rangle\ \langle \mathit{Term\text{-}2} \rangle\ \texttt{.}$$

- n is the bound on the desired reachability solutions
- m is the maximum depth of the narrowing tree
- `Term-1` is not a variable but may contain variables
- `Term-2` is a pattern to be reached
- `SearchArrow` is either =>1, =>+, =>*, =>!
- =>! denotes strongly irreducible terms or rigid normal forms.
- Implemented at the core level of Maude (C++)
- "`vu-narrow {filter}`" for filtered variant unification

Variant-based unification in Narrowing Search Command

```
mod NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > [narrowing] .
  rl [buy-a] : < M $ > => < M a q > [narrowing] .
  eq [change] : q q q q M = $ M [variant] .
endm
```

```
Maude> vu-narrow [1] in NARROWING-VENDING-MACHINE : < M:Money > =>* < a c > .

Solution 1
state: < a c #1:Money >
accumulated substitution:
M:Money --> $ q q q #1:Money
variant unifier:
#1:Money --> empty
```

Variant-based unification in Narrowing Search Command

```
mod AG-VENDING is
  sorts Item Items State Coin Money .
  subsort Item < Items . subsort Coin < Money .
  op __ : Items Items -> Items [assoc comm id: mt] .
  op <_|_> : Money Items -> State .
  ops a c : -> Item . ops q $ : -> Coin .
  rl < M:Money | I:Items > => < M:Money + - $           | I:Items c > [narrowing] .
  rl < M:Money | I:Items > => < M:Money + - q + - q + - q | I:Items a > [narrowing] .
  eq $ = q + q + q + q [variant] . --- Property of the original vending machine example
  op _+_ : Money Money -> Money [comm assoc] .
  op -_ : Money -> Money .
  op 0 : -> Money .
  vars X Y Z : Money .
  ... (here come the variant equations shown before for Abelian Group)
endm

Maude> vu-narrow [1] in AG-VENDING : < M:Money | mt > =>* < 0 | a c > .
Solution 1
rewrites: 32032 in 247478ms cpu (272327ms real) (129 rewrites/second)
state: < %1:Money + - (q + q + q + q + q + q + q) | a c >
accumulated substitution:
M:Money --> %1:Money
variant unifier:
%1:Money --> q + q + q + q + q + q + q

Maude> vu-narrow {filter} [1] in AG-VENDING : < M:Money | mt > =>* < 0 | a c > .
Solution 1
rewrites: 510 in 236ms cpu (274ms real) (2160 rewrites/second)
state: < %1:Money + - (q + q + q + q + q + q + q) | a c >
accumulated substitution:
M:Money --> %1:Money
variant unifier:
%1:Money --> q + q + q + q + q + q + q
```

Assoc unification in Narrowing Search Command

```
mod GRAMMAR is
  sorts Symbol NSymbol TSymbol String Production Grammar Conf .
  subsorts TSymbol NSymbol < Symbol < String . subsort Production < Grammar .
  ops 0 1 2 eps : -> TSymbol . ops S A B C : -> NSymbol .
  op _@_ : String Grammar -> Conf . op _->_ : String String -> Production .
  op __ : String String -> String [assoc id: eps] . op mt : -> Grammar .
  op _;_ : Grammar Grammar -> Grammar [assoc comm id: mt] .
  vars L1 L2 U V : String . var G : Grammar . var N : NSymbol . var T : TSymbol .
  rl ( L1 U L2 @ (U -> V) ; G) => ( L1 V L2 @ (U -> V) ; G) [narrowing] .
endm

Maude> vu-narrow [1] in GRAMMAR : N @ (S -> eps) ; S -> 0 S 1 =>* (0 0 1 1) @ (S -> eps) ; S -> 0 S 1 .
Solution 1
rewrites: 5 in 1ms cpu (1ms real) (3518 rewrites/second)
state: (0 0 1 1) @ (S -> eps) ; S -> 0 S 1
accumulated substitution:
N --> S
variant unifier:

Maude> vu-narrow [1] in GRAMMAR : S @ (N -> T) ; (S -> eps) ; S -> 0 S 1 =>* (0 0 1) @ (N -> T) ; (S -> eps) ; S -> 0 S 1 .
Solution 1
rewrites: 6 in 1ms cpu (1ms real) (4115 rewrites/second)
state: (0 %1:TSymbol 1) @ (S -> eps) ; (S -> %1:TSymbol) ; S -> 0 S 1
accumulated substitution:
N --> S
T --> %1:TSymbol
variant unifier:
%1:TSymbol --> 0
```

No warning is shown!!!

Outline

Model Checking

- Model checking techniques effective in verification of concurrent systems
- However, standard techniques only work for:
  - specific initial state (or finite set of initial states)
  - the set of states reachable from the initial state is finite
  - abstraction techniques
- Various model checking techniques for infinite-state systems exist, but they are less developed
  - Stronger limitations on the kind of systems and/or the properties that can be model checked

VENDING Example (1/6)

Terminating theory without rules adding money (\$ and q).

$$\boxed{\begin{array}{c} \underline{< \ \$ \ \$ \ >} \\ \downarrow \\ < \ \$ \ c \ > \\ \downarrow \\ < \ a \ q \ c \ > \end{array}}$$

(one initial state - finite space)

VENDING Example (2/6)

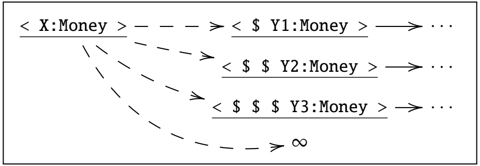
Non-terminating theory with rules adding money (\$ and q).

$$\begin{tikzcd} \underline{<\ \$\ >} \arrow[d] & \underline{<\ c\ \$\ >} \arrow[d] & \underline{<\ c\ c\ \$\ >} \arrow[d] \\ <\ c\ > \arrow[ru] & <\ c\ c\ > \arrow[ru] & \infty \end{tikzcd}$$

(one initial state - infinite space)

VENDING Example (3/6)

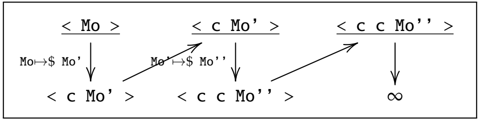
Instantiation is another source of infinity.



(infinite number of initial states)

VENDING Example (4/6)

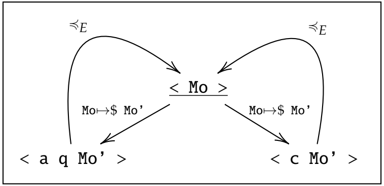
Narrowing usually provides an infinite space due to instantiation even for terminating theories (e.g. without rules adding money ($ and q)).



(one initial state - infinite space)

VENDING Example (5/6)

Narrowing-based state space can be treated in new ways and folded into a finite space in many cases



Narrowing + folding relation ⇒ (multiple initial states - finite space)
(equality $=_E$)
(renaming $\approx_E$)
(instantiation $\preccurlyeq_E$)

VENDING Example (6/6)

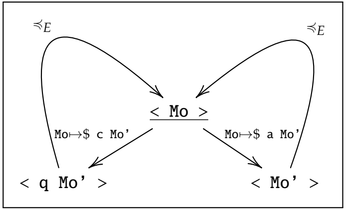
```
Maude> fvu-narrow in NARROWING-VENDING-MACHINE : < M:Marking > =>* < a c > .

Solution 1
state: < #1:Marking >
accumulated substitution:
M:Marking --> #1:Marking
variant unifier:
#1:Marking --> a c

No more solutions.
```

FVU-VENDING Example

```
mod FOLDING-NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ c > => < M > [narrowing] .
  rl [buy-a] : < M $ a > => < M q > [narrowing] .
  eq [change] : q q q q M = $ M [variant] .
endm
```




```
Maude> fvu-narrow in FOLDING-NARROWING-VENDING-MACHINE : < M:Marking a c > =>* < empty > .

Solution 1
state: < #1:Marking >
accumulated substitution:
M:Marking --> $ q q q #1:Marking
variant unifier:
#1:Marking --> empty
```

Outline

Applications

- Variant-based unification itself
- Formal reasoning tools :
  - Relying on unification capabilities:
    - termination proofs
    - proofs of local confluence and coherence
  - Relying on narrowing capabilities:
    - narrowing-based theorem proving
    - testing
- Logical model checking (model checking with logical variables)
- Cryptographic protocol analysis:
  - the Maude-NPA tool (narrowing + unification in Maude)
  - the Tamarin and AKISS protocol analyzers also use Maude capabilities
- Program transformation: partial evaluation, slicing
- SMT based on narrowing or by variant generation.

Thank you!

More information in the Maude webpage.