Unification and Narrowing in Maude 3.5

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

September 30, 2025

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 Symbolic capabilities to Maude?

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 are lifted to differentiation between symbolic models and equational reasoning.
4. Equational unification and narrowing modulo combinations of A,C,U and variant equations
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.

   ⋮

What have we done!!

- Maude 2.4 (2009) Built-in AC Unification and Narrowing-based search (rule & axioms)
- Maude 2.6 (2011) Built-in ACU Unification. Variant Unification. Narrowing search (eqs)
- Maude 2.7 (2015) A+C+U Unification. Built-in Variant unification. Narrowing search
- Maude 2.7.1 (2016) Built-in Bounded Associativity
- Maude 3.0 (2019) Built-in Narrowing-based search
- Maude 3.2 (2022) Minimal Unification for axioms, for variants, for narrowing
- Maude 3.3 (2023) Improved Folding Narrowing-based search
- Maude 3.4 (2024) Folding Narrowing-based search, Meta-interpreters, Object notation, External Processes.
- Maude 3.5 (2025) Improved performance, disjunctive patterns

Valencia: narrowing with constraints, anti-unification, homeomorphic embedding, security

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 $ > [narrowing] .
  rl [add-q] :  < M > => < M q > [narrowing] .
  rl [buy-c] :  < M $ > => < M cap > [narrowing] .
  rl [buy-a] :  < M $ > => < M cookie q > [narrowing] .
  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

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)

Unification Command in Maude

Maude provides a Ax-unification command of the form:

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

- 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)

Axiomatization of Booleans in Maude using axioms and variant equations

```
fmod BOOL-FVP is protecting TRUTH-VALUE .
   op _and_ : Bool Bool -> Bool [assoc comm] .
   op _xor_ : Bool Bool -> Bool [assoc comm] .
   op not_ : Bool -> Bool .
   op _or_ : Bool Bool -> Bool .
   op _<=>_ : Bool Bool -> Bool .
   vars X Y Z W : Bool .

   eq X and true = X [variant] .
   eq X and false = false [variant] .
   eq X and X = X  [variant] .
   eq X and X and Y = X and Y [variant] .   *** AC extension
   eq X xor false = X [variant] .
   eq X xor X = false [variant] .
   eq X xor X xor Y = Y [variant] .         *** AC extension
   eq not X = X xor true [variant] .
   eq X or Y = (X and Y) xor X xor Y [variant] .
   eq X <=> Y = true xor X xor Y [variant] .
endfm
```

Unification modulo axioms w/o minimality

```
========================================
unify in BOOL-FVP : X and not Y and not Z =? W and Y and not X .
Decision time: 0ms cpu (0ms real)

Unifier 1
X --> #2:Bool and not #1:Bool
Z --> #1:Bool
Y --> #2:Bool and not #1:Bool
W --> not #1:Bool

.....

Unifier 5
X --> not #1:Bool
Z --> #1:Bool
Y --> not #1:Bool
W --> not #1:Bool
========================================
irredundant unify in BOOL-FVP : X and not Y and not Z =? W and Y and not X .
Decision time: 0ms cpu (0ms real)

Unifier 1
X --> #1:Bool and #2:Bool
Z --> #1:Bool and #2:Bool
Y --> #1:Bool
W --> #2:Bool and not #1:Bool

Unifier 2
X --> #2:Bool
Z --> #1:Bool
Y --> #2:Bool
W --> not #1:Bool
```

Outline

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

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

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)

E,Ax-variants - Example

$$X \oplus 0 \to X$$
$$X \oplus X \to 0$$
$$X \oplus X \oplus Y \to Y$$
(cancellation rules: E)

$$X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$$
$$X \oplus Y = Y \oplus X$$
(axioms: Ax)

- For $X \oplus X$ only E,Ax-variant is: $(0, id)$
- 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. $(Z, \{X \mapsto U, Y \mapsto Z \oplus U\})$
  7. $(Z_1 \oplus Z_2, \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\})$

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!!!

Variant Generation in Incomplete Theories (due to assoc)

```
fmod VARIANT-UNIFICATION-ASSOC is
 protecting NAT .
 sort NList .
 subsort Nat < NList .

 op _:_ : NList NList -> NList [assoc ctor] .

 var E : Nat .
 var L : NList .

 ops tail prefix : NList ~> NList .
 ops head last : NList ~> Nat .
 eq head(E : L) = E [variant] .
 eq tail(E : L) = L [variant] .
 eq prefix(L : E) = L [variant] .
 eq last(L : E) = E [variant] .

 op duplicate : NList ~> Bool .
 eq duplicate(L : L) = true [variant] .
endfm
```

```
Maude> get variants in VARIANT-UNIFICATION-ASSOC :
         duplicate(prefix(L) : tail(L)) .

Variant 1
[Bool]: duplicate(prefix(#1:NList) : tail(#1:NList))
L --> #1:NList

Variant 2
[Bool]: duplicate(%1:NList : tail(%1:NList : %2:Nat))
L --> %1:NList : %2:Nat

Variant 3
[Bool]: duplicate(prefix(%1:Nat : %2:NList) : %2:NList)
L --> %1:Nat : %2:NList

Variant 4
[Bool]: duplicate(#1:Nat : #2:NList : #2:NList : #3:Nat)
L --> #1:Nat : #2:NList : #3:Nat

Variant 5
[Bool]: duplicate(#1:Nat : #2:Nat)
L --> #1:Nat : #2:Nat

Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.
Variant 6
Bool: true
L --> %1:Nat : %1:Nat : %1:Nat

Variant 7
Bool: true
L --> %1:Nat : %1:Nat

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

Finite Variant Property

- Theory has FVP if there is a finite number of most general E,Ax-variants for every term.
- If finite number of unifiers from t, E,Ax-narrowing must compute them, though infinite redundant E,Ax-narrowing sequences may exist
- [Comon-Delaune-RTA05] An equational theory has the finite variant property if there is a bound n in the number of steps for each term

$$\forall t, \exists n, \forall \sigma \text{ s.t. } (\sigma{\downarrow_{E,Ax}})(t) \xrightarrow{\leq n}_{E,Ax} \sigma(t){\downarrow_{E,Ax}}$$

Finite Variant Property

1. [Comon-Delaune-RTA05]
   Exclusive Or (max. bound 1)
2. [Comon-Delaune-RTA05]
   Abelian group (max. bound 2)
3. [Comon-Delaune-RTA05]
   Diffie-Hellman (max. bound 4)
4. [Comon-Delaune-RTA05]
   Homomorphism (NOT)
5. [Escobar-Meseguer-Sasse-RTA08]
   Sufficient & necessary conditions for FVP

Folding Variant-Narrowing

1. Complete narrowing strategy modulo axioms Ax with smaller search space than unrestricted Ax-narrowing.
2. Terminating when FVP (i.e., decidable narrowing-based $As \uplus E$-unification procedure)
3. Optimally terminating (no other possible narrowing strategy terminates for more equational theories)

- Based on E,Ax-*variant*, folding variant narrowing strategy:
  1. it only uses substitutions in normal form
  2. if a variant is an instance of a more general variant computed previously, stop narrowing path
  3. complete under very general assumptions on Ax and E

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:

```
variant unify [ n ] in ⟨ModId⟩ :
    ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
filtered variant unify [ n ] in ⟨ModId⟩ :
    ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
```

- 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 Unification modulo axioms w/o minimality

```
==========================================
variant unify in BOOL-FVP : (X or Y) <=> Z =? true .

Unifier 1
rewrites: 489 in 1828ms cpu (2110ms real) (267 rewrites/second)
X --> true
Y --> #1:Bool
Z --> true

...

Unifier 12
rewrites: 2934 in 9927ms cpu (11571ms real) (295 rewrites/second)
X --> %1:Bool and %2:Bool
Y --> %1:Bool and %3:Bool
Z --> (%1:Bool and %2:Bool) xor (%1:Bool and %3:Bool) xor %1:Bool and %2:Bool and %3:Bool

No more unifiers.
rewrites: 3006 in 9998ms cpu (11657ms real) (300 rewrites/second)
==========================================
filtered variant unify in BOOL-FVP : (X or Y) <=> Z =? true .
rewrites: 3224 in 10161ms cpu (11957ms real) (317 rewrites/second)

Unifier 1
X --> #1:Bool xor #2:Bool
Y --> #1:Bool
Z --> #2:Bool xor #1:Bool and #1:Bool xor #2:Bool

No more unifiers.
Advisory: Filtering was complete.
```

Outline

Narrowing-based Symbolic Reachability Analysis

- 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

Narrowing Search Command in Maude

Narrowing-based search command of the form:

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

- 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++)
- "{`fold`} `vu-narrow` {`filter,delay`}" is the most general version (new things to come)

Outline

Ex1 - Constrained Horn Clauses for Program Verification TPLP 2022

```
int sum_upto(int x) {
 int r = 0 ;
 while (x > 0) {
 r = r + x; x = x - 1; }
 return r;
}
```

This imperative program is translated into a logic program and the Hoare triple

$$\{m \geq 0\} sum = sum\_upto(m) \{sum \geq m\}$$

is satisfied only if the corresponding logic program is satisfiable.

Ex1 - Constrained Horn Clauses for Program Verification TPLP 2022

```
mod CHC is protecting NAT-FVP * (op _+_ to _++_, op _>_ to _>>_) .
  ...
  eq (nat V) ; P | M = P | (M (V -> 0))  .            --- New Variable
  eq (V = E) ; P | M = E ; (V = {}) ; P | M  .        --- Assignment
  eq N ; (V = {}) ; P | M  = P | (M  (V -> N)) .      --- Cont'd
  eq V ; P | (M  (V -> N)) = N ; P | (M  (V -> N)) .  --- Variable
  eq (E1 > E2) ; P | M = E1 ; E2 ; > ; P | M .        --- Comparison
  eq N ; E2 ; > ; P | M = E2 ; N ; > ; P | M .        --- Cont'd
  eq N2 ; N1 ; > ; P | M = (N1 >> N2) ; P | M .       --- Cont'd
  eq (E1 + E2) ; P | M  = E1 ; E2 ; + ; P | M .       --- Addition
  eq N ; E2 ; + ; P | M = E2 ; N ; + ; P | M .        --- Cont'd
  eq N2 ; N1 ; + ; P | M = (N1 ++ N2) ; P | M .       --- Cont'd
  eq E - 1 ; P | M  = E ; - ; P | M  .                --- Predecessor
  eq N ; - ; P | M   = pred(N) ; P | M .              --- Cont'd
  eq while E {B} ; P | M = E ; while E {B} ; P | M .  --- While
  rl true ; while E {B} ; P | M => B ; while E {B} ; P | M [narrowing] .
  rl false ; while E {B} ; P | M => P | M [narrowing] .
endm
}
```

Ex1 - Constrained Horn Clauses for Program Verification TPLP 2022

```
(X ++ Z) >> 0 ; ...  | (r -> R ++ X ++ Z) (x -> X ++ Z)
```

$\preccurlyeq_{\mathcal{E}}$ true false

```
(X' ++ Z') >> 0 ; ...  | (r -> 1 ++ 1 ++ X' ++ X' ++ R' ++ Z' ++ Z') (x -> X' ++ Z')

skip | (x -> 0) (r -> R')
```

```
Maude> {fold} vu-narrow {delay, filter}
  while (x > 0) {r = r + x ; x = x - 1} | (x -> X ++ Z) (r -> R)
  =>*
  skip | (x -> W) (r -> X) .

No solution.
rewrites: 79 in 16ms cpu (19ms real) (4725 rewrites/second)
```

Ex2 - Constrained Horn Clauses for Program Verification TPLP 2022

```
type tree = Leaf | Node of int * tree * tree ;;
let min x y = if x < y then x else y ;;
let rec min-leafdepth t = match t with
     | Leaf -> 0
     | Node(x,l,r) -> 1+min(min-leafdepth(l),min-leafdepth(r)) ;;
let rec left-drop n t = match t with
    | Leaf -> Leaf
    | Node(x,l,r) -> if n <= 0 then Node(x,l,r) else left-drop (n-1) l ;;
}
```

the Tree-Processing program, written in OCaml syntax is translated into a logic program and the property

$$\forall n, t : n \geq 0 \implies \mathit{min\text{-}leafdepth}(\mathit{left\text{-}drop}(n,t)) + n) \geq \mathit{min\text{-}leafdepth}(t) \tag{1}$$

is satisfied only if the corresponding logic program is satisfiable.

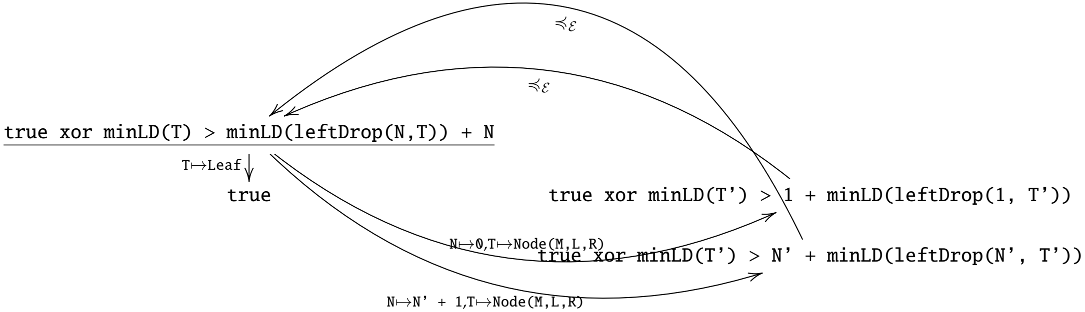
Ex2 - Constrained Horn Clauses for Program Verification TPLP 2022

```
mod TREE is protecting NAT-FVP .
  sort Tree .
  op Leaf : -> Tree .
  op Node : Nat Tree Tree -> Tree .
  vars N M : Nat . vars T L R : Tree .

  op minLD : Tree -> Nat .
  eq minLD(Leaf) = 0 .
  eq minLD(Node(N,L,R)) = 1 + min(minLD(L),minLD(R)) .
  rl minLD(Leaf) => 0 [narrowing] .
  rl minLD(Node(N,L,R)) => 1 + min(minLD(L),minLD(R)) [narrowing] .

  op leftDrop : Nat Tree -> Tree .
  eq leftDrop(N,Leaf) = Leaf .
  eq leftDrop(0,Node(M,L,R)) = Node(M,L,R) .
  eq leftDrop(N + 1,Node(M,L,R)) = leftDrop(N,L) .
  rl leftDrop(N,Leaf) => Leaf [narrowing] .
  rl leftDrop(0,Node(M,L,R)) => Node(M,L,R) [narrowing] .
  rl leftDrop(N + 1,Node(M,L,R)) => leftDrop(N,L) [narrowing] .
endm
}
```

Ex2 - Constrained Horn Clauses for Program Verification TPLP 2022




```
Maude> {fold} vu-narrow {delay, filter}
  not (minLeafDepth(T) > (minLD(leftDrop(N,T)) + N))  =>* false .

No solution.
rewrites: 19 in 1ms cpu (1ms real) (12541 rewrites/second)
```

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.
- Narrowing-based Theorem Prover NuITP
- Deductive Model Checking DMCheck

Thank you!

More information in the Maude webpage.