INTRODUCTION TO REWRITING LOGIC AND MAUDE

Peter Ölveczky

(University of Oslo)

International School on Rewriting, Paris, July 4, 2019

CONTENT

Rewriting Logic

Maude

Maude Tutorial

- Equational Specifcations in Maude
- Rewrite Theories in Maude
- Object-based Specifications in Full Maude
- Example: Distributed Mutual Exclusion

Final Remarks, Exercises

REWRITING LOGIC

BACK IN THE 80-IES

- Term rewriting $\approx$ equational logic
  - "abstract data types"
  - "equational (KB-) completion"
  - OBJ2, CLEAR, Larch, ASF, ...
  - denotational semantics: algebra
  - termination + confluence

BACK IN THE 80-IES

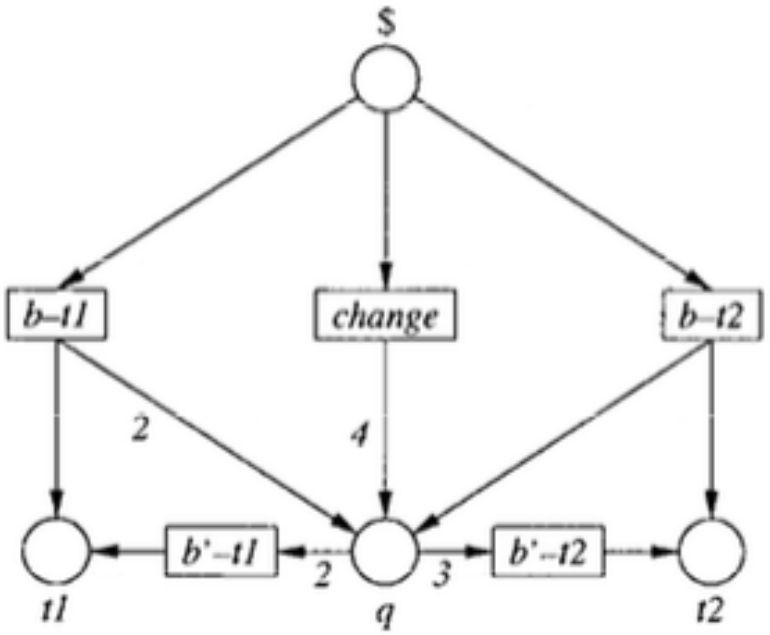
- Term rewriting $\approx$ equational logic
  - "abstract data types"
  - "equational (KB-) completion"
  - OBJ2, CLEAR, Larch, ASF, ...
  - denotational semantics: algebra
  - termination + confluence
- Concurrent/distributed systems?

LATE 80-IES

José Meseguer: rewriting simple model for concurrency

- naturally concurrent
- non-terminating and/or non-confluent

PETRI NETS AS REWRITE THEORIES




```
mod TICKET is
  subsorts Place < Marking .
  ops $,q,t1,t2 : -> Place .
  op _⊗_ : Marking Marking -> Marking
                [assoc comm id: λ] .
  rl b-t1 : $ => t1 ⊗ q ⊗ q .
  rl b-t2 : $ => t2 ⊗ q .
  rl change : $ => q ⊗ q ⊗ q ⊗ q .
  rl b'-t1 : q ⊗ q => t1 .
  rl b'-t2 : q ⊗ q ⊗ q => t2 .
endm
```

Fig. 7. A Petri net and its code in Maude.

REWRITING LOGIC

Theoretical Computer Science 96 (1992) 73–155
Elsevier

Conditional rewriting logic as a unified model of concurrency

José Meseguer*

SRI International, Menlo Park, CA 94025, USA, and Center for the Study of Language and Information, Stanford University, Stanford, CA 94305, USA

- equational specification defines data types
- rewrite rules define state change (transitions)

REWRITING LOGIC (CONT.)

- Rewrite theory (Σ, E, L, R):
  - (Σ, E) algebraic equational specification
  - L rule labels
  - R rewrite rules
    - applied "modulo E"

REWRITING LOGIC (CONT.)

- Rewrite theory (Σ, E, L, R):
  - (Σ, E) algebraic equational specification
  - L rule labels
  - R rewrite rules
    - applied "modulo E"
- Terms $[t]_E$ (of some sorts) $\approx$ states
- Sequent $[t_1]_E \longrightarrow [t_2]_E$
  - "possible to reach state $[t_2]_E$ from state $[t_1]_E$"

DEDUCTION RULES FOR REWRITING LOGIC

Reflexivity: $\mathcal{R} \vdash [t]_E \longrightarrow [t]_E$ holds for each term t

Congruence: If $\mathcal{R} \vdash [t_1]_E \longrightarrow [u_1]_E, \quad \ldots, \quad \mathcal{R} \vdash [t_n]_E \longrightarrow [u_n]_E$ hold, then $\mathcal{R} \vdash [f(t_1, \ldots, t_n)]_E \longrightarrow [f(u_1, \ldots, u_n)]_E$ holds for each function symbol f

Rule instantiation: If $\mathcal{R} \vdash [t_1]_E \longrightarrow [u_1]_E, \quad \ldots,$ $\mathcal{R} \vdash [t_n]_E \longrightarrow [u_n]_E$ hold, then $\mathcal{R} \vdash$ $[t(t_1/x_1, \ldots, t_n/x_n)]_E \longrightarrow [u(u_1/x_1, \ldots, u_n/x_n)]_E$ also holds for rule $l: \; t(x_1, \ldots, x_n) \longrightarrow u(x_1, \ldots, x_n)$

Transitivity: If $\mathcal{R} \vdash [t_1]_E \longrightarrow [t_2]_E$ and $\mathcal{R} \vdash [t_2]_E \longrightarrow [t_3]_E$ hold, then $\mathcal{R} \vdash [t_1]_E \longrightarrow [t_3]_E$ also holds

DEDUCTION RULES FOR REWRITING LOGIC

Reflexivity: $\mathcal{R} \vdash [t]_E \longrightarrow [t]_E$ holds for each term t

Congruence: If $\mathcal{R} \vdash [t_1]_E \longrightarrow [u_1]_E, \quad \ldots, \quad \mathcal{R} \vdash [t_n]_E \longrightarrow [u_n]_E$ hold, then $\mathcal{R} \vdash [f(t_1, \ldots, t_n)]_E \longrightarrow [f(u_1, \ldots, u_n)]_E$ holds for each function symbol f

Rule instantiation: If $\mathcal{R} \vdash [t_1]_E \longrightarrow [u_1]_E, \quad \ldots,$ $\mathcal{R} \vdash [t_n]_E \longrightarrow [u_n]_E$ hold, then $\mathcal{R} \vdash$ $[t(t_1/x_1, \ldots, t_n/x_n)]_E \longrightarrow [u(u_1/x_1, \ldots, u_n/x_n)]_E$ also holds for rule $l: \; t(x_1, \ldots, x_n) \longrightarrow u(x_1, \ldots, x_n)$

Transitivity: If $\mathcal{R} \vdash [t_1]_E \longrightarrow [t_2]_E$ and $\mathcal{R} \vdash [t_2]_E \longrightarrow [t_3]_E$ hold, then $\mathcal{R} \vdash [t_1]_E \longrightarrow [t_3]_E$ also holds

Definition

Concurrent step: Proved without Transitivity

SEMANTICS

- Model-theoretic semantics: Categories
  - objects: states $[t]_E$
  - arrows: $[t_1]_E \xrightarrow{\pi} [t_2]_E$

SEMANTICS

- Model-theoretic semantics: Categories
  - objects: states $[t]_E$
  - arrows: $[t_1]_E \xrightarrow{\pi} [t_2]_E$
- Meseguer:
  - shows existence of initial and free models
  - defines satisfaction
  - proves soundness and completeness

MAUDE

THE MAUDE TOOL

Maude

- Rewriting logic language/tool
  - SRI International and University of Illinois

THE MAUDE TOOL

Maude

- Rewriting logic language/tool
  - SRI International and University of Illinois
- Object-based modeling of distributed systems

THE MAUDE TOOL

Maude

- Rewriting logic language/tool
  - SRI International and University of Illinois
- Object-based modeling of distributed systems
- Explicit-state analysis methods
  - rewriting for simulation
  - seaech for reachability analysis
  - LTL and LTLR model checking

THE MAUDE TOOL

Maude

- Rewriting logic language/tool
  - SRI International and University of Illinois
- Object-based modeling of distributed systems
- Explicit-state analysis methods
  - rewriting for simulation
  - seaech for reachability analysis
  - LTL and LTLR model checking
- Symbolic methods being developed
  - terms with variables represent infinitely many states
  - narrowing
  - rewriting with SMT-solving

META-LEVEL PROGRAMMING IN MAUDE

- Maude module M represented as term $\overline{M}$ of sort `Module`
- Module transformations defined as Maude functions

$$f : \texttt{Module} \longrightarrow \texttt{Module}$$

- Powerful for defining tools in Maude

EXTENSIONS

- Real-time systems: Real-Time Muade

EXTENSIONS

- Real-time systems: Real-Time Muade
- Probabilistic systems:
  - use `random` to define your own probability distributions
  - connect "probabilistic rewrite theory" to statistical model checkers PVeStA and MultiVeStA
  - performance estimation ...

DISTRIBUTED PROGRAMMING

Distributed Maude sessions can communicate by Maude sockets

- connect to external environments
- from verified model to correct-by-construction distributed implementation

APPLICATIONS

- Security
  - found unknown address bar and status bar spoof attacks in web browsers
  - Maude-NPA: Cathy Meadows NLA Protocol Analyzer

APPLICATIONS

- Security
  - found unknown address bar and status bar spoof attacks in web browsers
  - Maude-NPA: Cathy Meadows NLA Protocol Analyzer
- Semantics for programming languages
  - C, Java, JVM, Scheme, EVM, ...
  - K framework (G. Rosu)
  - errors in electronic contracts on blockchain

APPLICATIONS

- Security
  - found unknown address bar and status bar spoof attacks in web browsers
  - Maude-NPA: Cathy Meadows NLA Protocol Analyzer
- Semantics for programming languages
  - C, Java, JVM, Scheme, EVM, ...
  - K framework (G. Rosu)
  - errors in electronic contracts on blockchain
- Semantics for modeling languages and frameworks (MOF, ...)
- Logical framework
  - automatically translate HOL libraries to NuPrl

APPLICATIONS

- Security
  - found unknown address bar and status bar spoof attacks in web browsers
  - Maude-NPA: Cathy Meadows NLA Protocol Analyzer
- Semantics for programming languages
  - C, Java, JVM, Scheme, EVM, ...
  - K framework (G. Rosu)
  - errors in electronic contracts on blockchain
- Semantics for modeling languages and frameworks (MOF, ...)
- Logical framework
  - automatically translate HOL libraries to NuPrl
- Network protocols and cloud computing
  - Apache Cassandra, Google's Megastore, ZooKeeper, ...

APPLICATIONS

- Security
  - found unknown address bar and status bar spoof attacks in web browsers
  - Maude-NPA: Cathy Meadows NLA Protocol Analyzer
- Semantics for programming languages
  - C, Java, JVM, Scheme, EVM, ...
  - K framework (G. Rosu)
  - errors in electronic contracts on blockchain
- Semantics for modeling languages and frameworks (MOF, ...)
- Logical framework
  - automatically translate HOL libraries to NuPrl
- Network protocols and cloud computing
  - Apache Cassandra, Google's Megastore, ZooKeeper, ...
- Biological systems
  - cell biology (Pathway logic)
  - brains (Alzheimer, Parkinson, ...)

MAUDE TUTORIAL

EQUATIONAL SPECIFICATION

- Equations must be (ground) confluent and terminating
- Maude computes normal form of expression

MAUDE EXAMPLE 1: NATURAL NUMBERS

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

  vars M N : Nat .  --- This is a comment

  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm
```

Maude specification a set of declarations

MAUDE EXECUTION

1. Start Maude
2. Read file into Maude:

   ```
   Maude> in nat-add.maude
   ```

3. Execute Maude:

   ```
   Maude> red s(s(0)) + s(0) .
   ```

MAUDE EXECUTION

1. Start Maude
2. Read file into Maude:

   ```
   Maude> in nat-add.maude
   ```

3. Execute Maude:

   ```
   Maude> red s(s(0)) + s(0) .

   result Nat: s(s(s(0)))
   ```

4. End session with `q` (or `quit`)

MODULE INCLUSION

`including` or `protecting`:

```
fmod NAT-MULT is
  protecting NAT-ADD .
  op _*_ : Nat Nat -> Nat .

  vars M N : Nat .

  eq 0 * M = 0 .
  eq s(M) * N = N + (M * N) .
endfm
```

EXAMPLE: LISTS

```
fmod NAT-LIST-CONSTR is
  protecting NAT-ADD .

  sort List .

  op nil : -> List [ctor] .
  op _ _ : List Nat -> List [ctor] .
endfm
```

EXAMPLE: LISTS

```
fmod NAT-LIST-CONSTR is
  protecting NAT-ADD .

  sort List .

  op nil : -> List [ctor] .
  op _ _ : List Nat -> List [ctor] .
endfm
```

List “1 2 3” represented by term

```
  nil s(0) s(s(0)) s(s(s(0)))
```

EXAMPLE: LISTS (CONT.)

```
fmod NAT-LIST is   protecting NAT-LIST-CONSTR .
  op length : List -> Nat .
  ops first last : List -> Nat .
  ops rest reverse : List -> List .

  vars N N' : Nat .        vars L L' : List .
  eq length(nil) = 0 .
  eq length(L N) = s(length(L)) .

  eq first(nil) = 0 .                  --- ??
  eq first(nil N) = N .
  eq first(L N N') = first(L N) .
endfm
```

BUILT-IN MODULES

Built-in modules

- `NAT` (unbounded natural numbers)
- `INT`
- `RAT`
- `BOOL`
- `STRING`
- `META-LEVEL`
- `...`

See file prelude.maude

SUBSORTS

Subsorts: `sorts s1 s2 .` `subsort s1 < s2 .`

every `s1` is also an `s2`

SUBSORTS

Subsorts: `sorts s1 s2 .` `subsort s1 < s2 .`

every `s1` is also an `s2`

- Inclusion: `subsort Nat < Int .`

SUBSORTS

Subsorts: `sorts s1 s2 .` `subsort s1 < s2 .`

every `s1` is also an `s2`

- Inclusion: `subsort Nat < Int .`
- Subdomain for partial functions:

```
sort NzNat .    subsort NzNat < Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> NzNat .

op _/_ : Nat NzNat -> Nat .
```

SUBSORTS (CONT.)

- Subdomain for partial functions:

```
sort NeList .   subsort NeList < List .
...
op __ : List Nat -> NeList [ctor] .

ops first last : NeList -> Nat .
op length : List -> Nat .
op rest : NeList -> List .
```

EQUATIONAL ATTRIBUTES

Binary function *f* can be declared

- associative: `op f : s s -> s [assoc] .`
- commutative: `op f : s s -> s [comm] .`
- have *t* as identity element: `op f : s s' -> s [id: t] .`

EQUATIONAL ATTRIBUTES

Binary function f can be declared

- associative: `op f : s s -> s [assoc] .`
- commutative: `op f : s s -> s [comm] .`
- have t as identity element: `op f : s s' -> s [id: t] .`

- deduction modulo these axioms

REVISITING LISTS: ASSOC AND ID

```
sorts List NeList .
subsorts Nat < NeList < List .

op nil : -> List [ctor] .
op __ : List List -> List [assoc id: nil ctor] .
op __ : NeList NeList -> NeList [assoc id: nil ctor] .
```

List “3 5 1” represented by term `3 5 1`

REVISITING LISTS: ASSOC AND ID

```
op length : List -> Nat .
ops first last : NeList -> Nat .
op rest : NeList -> List .

vars M N : Nat .    var L : List .

eq length(nil) = 0 .
eq length(N L) = 1 + length(L) .

eq first(N L) = N .        eq rest(N L) = L .
```

EXAMPLE: MERGE-SORT

```
op mergeSort : List -> List .
op merge : List List -> List [comm] .
```

EXAMPLE: MERGE-SORT

```
op mergeSort : List -> List .
op merge : List List -> List [comm] .

vars L L' : List .     vars NEL NEL' : NeList .
vars I J : Int .

eq mergeSort(nil) = nil .
eq mergeSort(I) = I .
```

EXAMPLE: MERGE-SORT

```
op mergeSort : List -> List .
op merge : List List -> List [comm] .

vars L L' : List .      vars NEL NEL' : NeList .
vars I J : Int .

eq mergeSort(nil) = nil .
eq mergeSort(I) = I .

ceq mergeSort(NEL NEL') =
        merge(mergeSort(NEL), mergeSort(NEL'))
    if length(NEL) == length(NEL')
       or  length(NEL) == s length(NEL') .
```

EXAMPLE: MERGE-SORT

```
op mergeSort : List -> List .
op merge : List List -> List [comm] .

vars L L' : List .      vars NEL NEL' : NeList .
vars I J : Int .

eq mergeSort(nil) = nil .
eq mergeSort(I) = I .

ceq mergeSort(NEL NEL') =
        merge(mergeSort(NEL), mergeSort(NEL'))
    if length(NEL) == length(NEL')
       or  length(NEL) == s length(NEL') .

eq merge(nil, L) = L .
ceq merge(I L, J L') = I merge(L, J L') if I <= J .
```

MULTISETS: ASSOC + ID + COMM

Multiset of `Msg` and `Object` elements:

```
sort Mset .    subsort Msg Object < Mset .

op none : -> Mset [ctor] .
op __ : Mset Mset -> Mset [ctor assoc comm id: none] .
```

REWRITE SPECIFICATIONS

- Model the life of person
- State `person(`*name*`,` *age*`,` *status*`)`
- Equation

```
ceq person(X, N, S) = person(X, N + 1, S)
    if N < 1000 .
```

unsuitable

REWRITE SPECIFICATIONS

- Model the life of person
- State `person(`*name*`,` *age*`,` *status*`)`
- Equation

```
ceq person(X, N, S) = person(X, N + 1, S)
    if N < 1000 .
```

unsuitable

- (Conditional) rewrite rule

```
crl [birthday] :
     person(X, N, S) => person(X, N + 1, S)
     if N < 1000 .
```

REWRITE RULES FOR STATE TRANSITIONS

```
sorts Status Person .

ops dead single married engaged separated : -> Status [ctor] .
op person : String Nat Status -> Person [ctor] .

var N : Nat .    var S : Status .    var P : String .

crl [birthday] :
    person(P, N, S) => person(P, N+1, S)  if N < 1000 .

rl [death] :  person(P, N, S) => person(P, N, dead) .

crl [engagement] :
    person(P, N, single) => person(P, N, engaged) if N > 15 .

rl [divorce] : person(P,N,married) => person(P,N,separated)
```

ANALYSIS IN MAUDE

- E-normal forms computed before applying rewrite rule
- Rewriting: simulation

Example

```
Maude> rew [10] person("Robert Fisk", 72, married) .
result Person: person("Robert Fisk", 82, married)
```

ANALYSIS IN MAUDE

- E-normal forms computed before applying rewrite rule
- Rewriting: simulation

Example

```
Maude> rew [10] person("Robert Fisk", 72, married) .
result Person: person("Robert Fisk", 82, married)

Maude> frew [5] person("Noam Chomsky", 90, married) .
result Person: person("Noam Chomsky", 95, married)
```

REACHABILITY ANALYSIS

Search all states (terms) reachable from the initial state:

`search` *`startterm`* *`arrow`* *`pattern`* [`such that` *`cond`*] .

arrow is either

`=>*` (reachable in 0 or more steps)

`=>!` ("final/deadlocked state")

pattern is a term (possibly with variables) and *cond* is a condition

- Limit number of answers: `search [`*n*`] ...`
- ... and number of rewrite steps: `search [`*n*`,`*d*`] ...`

REACHABILITY ANALYSIS

Example

Where will it all end?

```
Maude> search person("Peter", 50, single) =>! P:Person .
```

REACHABILITY ANALYSIS

Example

Where will it all end?

```
Maude> search person("Peter", 50, single) =>! P:Person .

Solution 1 (state 2900)
P:Person --> person("Peter", 1001, married)

No more solutions.
```

REACHABILITY ANALYSIS

Example

Where will it all end?

```
Maude> search person("Peter", 50, single) =>! P:Person .

Solution 1 (state 2900)
P:Person --> person("Peter", 1001, married)

No more solutions.
```

Can I become younger?

```
Maude> search person("Peter", 50, married) =>*
                  person("Peter", N, S)
               such that  N < 50 .
```

REACHABILITY ANALYSIS

Example

Where will it all end?

```
Maude> search person("Peter", 50, single) =>! P:Person .

Solution 1 (state 2900)
P:Person --> person("Peter", 1001, married)

No more solutions.
```

Can I become younger?

```
Maude> search person("Peter", 50, married) =>*
                  person("Peter", N, S)
               such that  N < 50 .

No solution.
```

OBJECT-ORIENTED SPECIFICATIONS

- Distributed system: state multiset of objects and messages
- Easy to define different forms of communication
  - asynchronous/synchronous
  - unicast/multicast/broadcast
  - ordered/unordered
  - ...

OO SPECIFICATIONS IN FULL MAUDE

Full Maude:

- Convenient syntax for OO

```
class C | att1 : s1, ..., attn : sn .
```

- Start: load `full-maude.maude`

OO SPECIFICATIONS IN FULL MAUDE

Full Maude:

- Convenient syntax for OO

```
class C | att1 : s1, ..., attn : sn .
```

- Start: load `full-maude.maude`
- Input enclosed by parentheses: `(omod ... endom)`

OBJECT-ORIENTED SPECIFICATIONS IN (FULL) MAUDE

Populations:

```
class Person | age : Nat, status : Status .

ops single divorced : -> Status [ctor] .
ops enganged married separated : Oid -> Status [ctor] .

msgs separate sepOK : Oid -> Msg .
```

OBJECT-ORIENTED SPECS IN (FULL) MAUDE II

A state in the system:

```
< "Emmanuel" : Person | age : 41, status : married("Brigitte") >
< "Hamlet" : Person | age : 20, status : single >
< "Ophelia" : Person | age : 16, status : single >
< "Brigitte" : Person | age : 66, status : separated("Emmanuel") >
separate("Emmanuel")
```

OBJECT-ORIENTED SPECS IN MAUDE: REWRITE RULES

```
vars N M : Nat .  vars P P' : Oid

crl [birthday] :
    < P : Person | age : N >  =>  < P : Person | age : N + 1 >
  if N < 1000 .

crl [engagement] :
    < P : Person | age : N, status : single >
    < P' : Person | age : M, status : single >
  =>
    < P : Person | status : engaged(P') >
    < P' : Person | status : engaged(P) >
  if N > 15 and M > 15 .
```

OBJECT-ORIENTED SPECS: MORE REWRITE RULES

```
rl [separate]
   < P : Person | status : married(P') >
 =>
   < P : Person | status : separated(P') >
   separate(P') .

rl [readSeparateRequest]
   separate(P)
   < P : Person | status : married(P') >
 =>
   < P : Person | status : separated(P') >
   ok(P') .
```

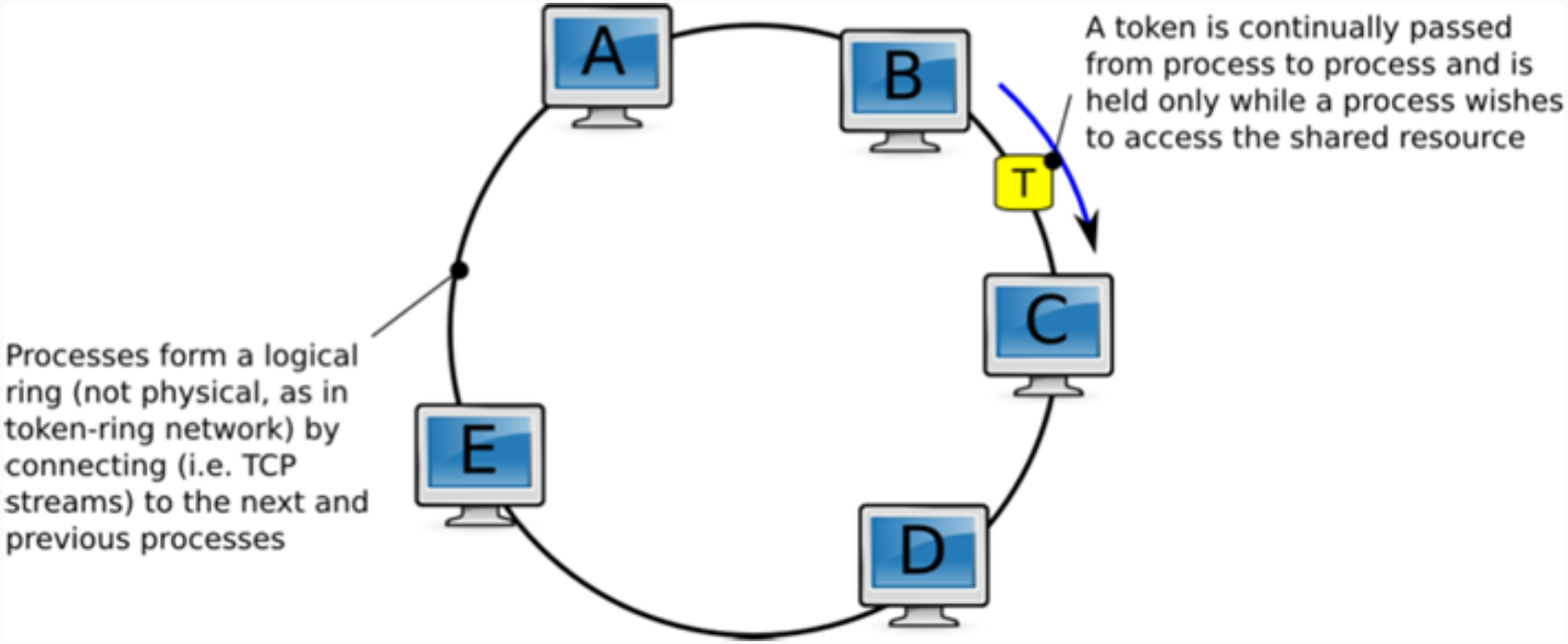
OBJECT-ORIENTED SPECS: MORE REWRITE RULES

```
rl [separate]
   < P : Person | status : married(P') >
 =>
   < P : Person | status : separated(P') >
   separate(P') .

rl [readSeparateRequest]
   separate(P)
   < P : Person | status : married(P') >
 =>
   < P : Person | status : separated(P') >
   ok(P') .

rl [death] :
  < P : Person | >  =>  none .     --- wrong; why?
```

EXAMPLE: TOKEN RING DISTRIBUTED MUTUAL EXCLUSION



```
load full-maude          --- start Full Maude

(omod TOKEN-RING-MUTEX is including MESSAGE-WRAPPER .

  class Node | state : MutexState, next : Oid .

  sort MutexState .
  ops outsideCS waitingCS insideCS : -> MutexState [ctor] .

  vars O O' O'' : Oid .

  op token : -> MsgContent [ctor] .
```

```
load full-maude        --- start Full Maude

(omod TOKEN-RING-MUTEX is including MESSAGE-WRAPPER .

  class Node | state : MutexState, next : Oid .

  sort MutexState .
  ops outsideCS waitingCS insideCS : -> MutexState [ctor] .

  vars O O' O'' : Oid .

  op token : -> MsgContent [ctor] .

  rl [wantToEnterCS] :
     < O : Node | state : outsideCS >
    =>
     < O : Node | state : waitingCS > .
```

```
rl [receiveAndPassOnToken] :
   (msg token from O to O')
   < O' : Node | state : outsideCS, next : O'' >
 =>
   < O' : Node | >
   (msg token from O' to O'') .
```

```
rl [receiveAndPassOnToken] :
   (msg token from O to O')
   < O' : Node | state : outsideCS, next : O'' >
 =>
   < O' : Node | >
   (msg token from O' to O'') .

rl [receiveTokenAndEnterCS] :
   (msg token from O to O')
   < O' : Node | state : waitingCS >
 =>
   < O' : Node | state : insideCS > .
```

```
  rl [receiveAndPassOnToken] :
     (msg token from O to O')
     < O' : Node | state : outsideCS, next : O'' >
   =>
     < O' : Node | >
     (msg token from O' to O'') .

  rl [receiveTokenAndEnterCS] :
     (msg token from O to O')
     < O' : Node | state : waitingCS >
   =>
     < O' : Node | state : insideCS > .

  rl [exitCS] :
     < O : Node | state : insideCS, next : O' >
   =>
     < O : Node | state : outsideCS >
     (msg token from O to O') .
endom)
```

INITIAL STATE

```
(omod TEST-TOKEN is
  including TOKEN-RING-MUTEX .

  ops a b c d : -> Oid [ctor] .
  op init : -> Configuration .

  eq init =
      (msg token from d to a)       --- token
      < a : Node | state : outsideCS, next : b >
      < b : Node | state : outsideCS, next : c >
      < c : Node | state : outsideCS, next : d >
      < d : Node | state : outsideCS, next : a > .
endom)
```

CHECKING MUTUAL EXCLUSION

```
Maude> (search [1] init =>*
                    C:Configuration
                    < O:Oid : Node | state : insideCS >
                    < O':Oid : Node | state : insideCS > .)
```

CHECKING MUTUAL EXCLUSION

```
Maude> (search [1] init =>*
                    C:Configuration
                    < O:Oid : Node | state : insideCS >
                    < O':Oid : Node | state : insideCS > .)

No solution.
```

LTL MODEL CHECKING

- Explicit-state model checker
- Atomic propositions: terms of sort `Prop`
  - parametric propositions
- Must define labeling function `_|=_ : State Prop -> Bool`
- Logical operators `~`, `->`, `/\`, `\/`, `True`, ...
- Temporal operators `[]`, `<>`, `U`, ...

MODEL CHECKING MUTUAL EXCLUSION

```
load model-checker

...

(omod MODEL-CHECK-TOKEN is
  including MODEL-CHECKER + TEST-TOKEN .

  subsort Configuration < State .  --- States are configurations
```

MODEL CHECKING MUTUAL EXCLUSION

```
load model-checker

...

(omod MODEL-CHECK-TOKEN is
  including MODEL-CHECKER + TEST-TOKEN .

  subsort Configuration < State .  --- States are configurations

  var O : Oid .   var REST : Configuration .   var MS : MutexState

  op executingInCS : Oid -> Prop [ctor] .
```

MODEL CHECKING MUTUAL EXCLUSION

```
load model-checker

...

(omod MODEL-CHECK-TOKEN is
  including MODEL-CHECKER + TEST-TOKEN .

  subsort Configuration < State .  --- States are configurations

  var O : Oid .   var REST : Configuration .   var MS : MutexState

  op executingInCS : Oid -> Prop [ctor] .

  eq REST < O : Node | state : MS >  |=  executingInCS(O)
      = (MS == insideCS) .
endom)
```

```
Maude> (red modelCheck(init,
                       ([] <> executingInCS(a))
                    /\ ([] <> executingInCS(b))) .)
```

```
Maude> (red modelCheck(init,
                       ([] <> executingInCS(a))
                    /\ ([] <> executingInCS(b))) .)

result ModelCheckResult :
  counterexample(...)
```

MODEL CHECKING WITH FAIRNESS ASSUMPTION

Add justice fairness assumption of wanting to enter CS:

```
ops outside waiting : Oid -> Prop [ctor] .
op just : Oid -> Formula .
```

MODEL CHECKING WITH FAIRNESS ASSUMPTION

Add justice fairness assumption of wanting to enter CS:

```
ops outside waiting : Oid -> Prop [ctor] .
op just : Oid -> Formula .

eq REST < O : Node | state : MS >  |=  outside(O)
    = (MS == outsideCS) .
```

MODEL CHECKING WITH FAIRNESS ASSUMPTION

Add justice fairness assumption of wanting to enter CS:

```
ops outside waiting : Oid -> Prop [ctor] .
op just : Oid -> Formula .

eq REST < O : Node | state : MS >  |=  outside(O)
    = (MS == outsideCS) .

eq REST < O : Node | state : MS >  |=  waiting(O)
    = (MS == waitingCS) .
```

MODEL CHECKING WITH FAIRNESS ASSUMPTION

Add justice fairness assumption of wanting to enter CS:

```
ops outside waiting : Oid -> Prop [ctor] .
op just : Oid -> Formula .

eq REST < O : Node | state : MS >  |=  outside(O)
    = (MS == outsideCS) .

eq REST < O : Node | state : MS >  |=  waiting(O)
    = (MS == waitingCS) .

eq just(O) = (<> [] outside(O)) -> ([] <> waiting(O)) .
```

MODEL CHECKING WITH FAIRNESS ASSUMPTION (CONT.)

```
Maude> (red modelCheck(init,
             (just(a) /\ just(b)) ->
                   (    ([] <> executingInCS(a))
                    /\ ([] <> executingInCS(b)))) .)
```

MODEL CHECKING WITH FAIRNESS ASSUMPTION (CONT.)

```
Maude> (red modelCheck(init,
             (just(a) /\ just(b)) ->
                   (    ([] <> executingInCS(a))
                    /\ ([] <> executingInCS(b)))) .)

result Bool :
  true
```

FINAL REMARKS, EXERCISES

TAKE AWAY FROM TALK

- Rewriting logic:
  - equational specification defines data types
  - rewrite rules define transitions

TAKE AWAY FROM TALK

- Rewriting logic:
  - equational specification defines data types
  - rewrite rules define transitions
- Simple and intuitive
- Expressive and general

TAKE AWAY FROM TALK

- Rewriting logic:
  - equational specification defines data types
  - rewrite rules define transitions
- Simple and intuitive
- Expressive and general
- Natural model of concurrent objects

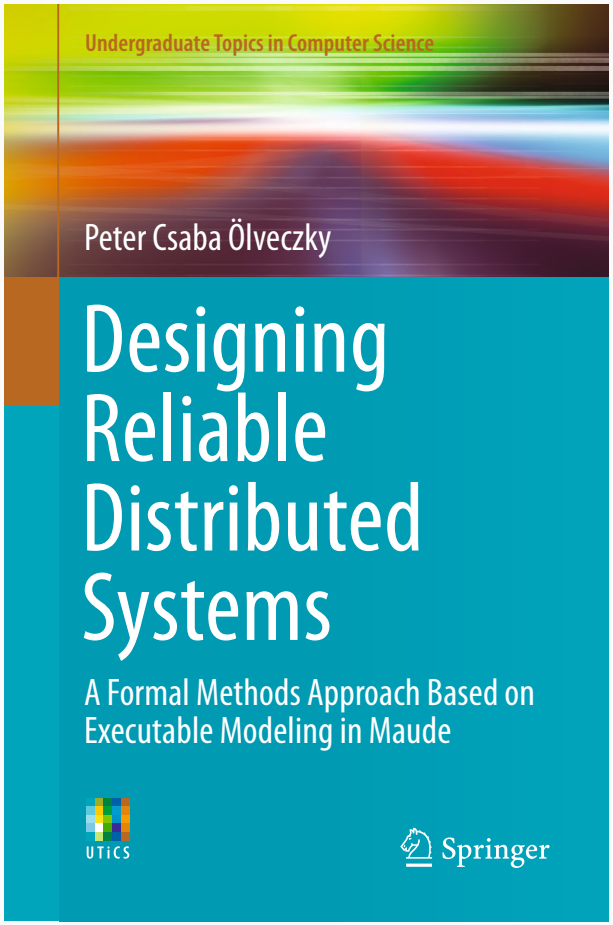
TAKE AWAY FROM TALK

- Rewriting logic:
  - equational specification defines data types
  - rewrite rules define transitions
- Simple and intuitive
- Expressive and general
- Natural model of concurrent objects
- Maude: high-performance tool
- State-of-the-art applications
- Simulation and explicit-state model checking
  - state space explosion
- Symbolic analysis methods developed

FURTHER READING

- M. Clavel et al.: All About Maude. Springer LNCS 4350, 2007
- J. Meseguer: Twenty years of rewriting logic. J. Log. Algebr. Program. 81(7-8), 2012
- J. Meseguer: Symbolic Reasoning Methods in Rewriting Logic and Maude. WoLLIC 2018, Springer LNCS 10944, 2018
- J. Meseguer and G. Rosu: The rewriting logic semantics project: A progress report. Inf. Comput. 231, 2013
- J. Meseguer and G. Rosu: The rewriting logic semantics project. Theor. Comput. Sci. 373(3), 2007
- G. Rosu: $\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools. Dependable Software Systems Engineering 2017, NATO Science for Peace and Security Series – D: Information and Communication Security 50, IOS Press, 2017

EXERCISES



Exercises 9.1 (use built-in `BOOL` module with sort `Bool`), 20, 23, 26, 37, 122.1, 145, (186; with simplification: a node does not use queues, but only reads requests when it can grant access), and 188.

ONE MORE EXERCISE

Define atomic proposition `someoneInCS` (some node is executing in critical section) and use LTL model checking to show that–even without fairness assumptions–nodes will execute inside critical section infinitely often.