Software Specification and Verification in Rewriting Logic: Lecture 1

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Sequential vs. Concurrent Programs

Programs come in many different languages and styles. This in fact impacts both the level of difficulty and the verification techniques suitable in each case.

A first useful distinctions is sequential vs. concurrent:

- sequential programs run on sequential computers, and for each input either yield an answer or loop;
- concurrent programs run simultaneously on different processors and may yield many different answers, or no answer at all, in the sense of being reactive systems constantly interacting with their environment.

although not identical, this distinction is closely related to that of deterministic vs. nondeterministic programs.

Imperative vs. Declarative

A second useful distinction is imperative vs. declarative:

- imperative programs are those of most conventional languages; they involve commands changing the state of the machine to perform a task;

- declarative programs give a mathematical axiomatization of a problem, as opposed to low-level instructions on how to solve it; they can be based on different logical systems.

Of course, the sequential vs. concurrent and the imperative vs. declarative are orthogonal distinctions: all four combinations are possible.

The Declarative Advantage

For program reasoning and verification purposes, declarative programs have the important advantage of being already a piece of mathematics. Specifically:

- a declarative program P in a language based on a given logic is typically a logical theory in that logic.
- the properties that we want to verify are satisfied by P can be stated in another theory Q; and
- the satisfaction relation that needs to be verified is a semantic implication relation $P \models Q$ stating that any model of P is also a model of Q.

The Imperative Program Verification Game

By contrast, imperative programs are not expressed in the language of mathematics, but in a conventional programming language like C, C^{++}, Java, or whatever, with all kinds of idiosyncrasies.

Thefore, the first thing that we crucially need to do in order to reason about programs in an imperative programming language $\mathcal{L}$ is to define the mathematical semantics of $\mathcal{L}$.

This we can always do in informal mathematics, but for tool assistance purposes it is advantageous to axiomatize the semantics of $\mathcal{L}$ as a logical theory $T_{\mathcal{L}}$ in a logic.

The Imperative Program Verification Game (II)

Then, given a program P in $\mathcal{L}$, the properties we wish to verify about P can typically be expressed as a logical theory $Q(P)$, involving somehow the text of P.

In the imperative case the satisfaction relation can again be understood as a semantic implication between two theories, namely the axiomatization of the language, and the desired properties: $T_{\mathcal{L}} \models Q(P)$.

Traditionally, a "logic of programs" such as Hoare's logic is used, with triples of the form $\{A\}P\{B\}$ with P the program and A, B formulas. In fact, the traditional approach can be seamlessly integrated with the above one (see lecture notes in the web page for CS 476 at the Univ. of Illinois CS Dept. for a detailed justification of this claim).

The Equational/Rewriting Logic Framework

A very good and nontrivial question is what logic to use as the framework logic for program specification and verification. There are many choices with different tradeoffs.

In these lectures we will use equational logic to axiomatize the semantics of declarative sequential programs, and rewriting logic to axiomatize the semantics of (declarative or imperative) concurrent programs.

To axiomatize the properties satisfied by such programs we will allow more expressive logics, such as full first-order logic, or even temporal logic (for concurrent programs).

The Equational/Rewriting Logic Framework (II)

The above choice has the following advantages:

1. suitable subsets of equational and rewriting logic are efficiently executable, giving rise, respectively, to a declarative sequential functional language, and a declarative concurrent language;
2. equational logic is very well suited to give executable axiomatizations of imperative sequential languages;
3. rewriting logic is likewise very well suited to give executable axiomatization of imperative concurrent languages;
4. therefore, we can specify all the four kinds of programs in an executable way within the combined framework.

Initiality and Induction

Yet another key advantage is that equational and rewriting logic theories have initial models. That is, theories in these logics have an intended or standard model, (also called initial) which is the one corresponding to our computational intuitions.

inductive reasoning principles, such as the different induction schemes, are then sound principles to infer other properties satisfied by the standard model of a theory.

The two crucial satisfaction relations for declarative, resp. imperative, program verification, namely, $P \models Q$, resp. $T_{\mathcal{L}} \models Q(P)$, should be understood as inductive satisfaction relations, corresponding to the initial model of P, resp. $T_{\mathcal{L}}$.

Maude

Maude is a declarative language and high-performance interpreter based on rewriting logic that is very well suited for concurrent specification and programming.

Since equational logic is a sublogic of rewriting logic, Maude has a functional programming sublanguage.

We will use Maude and its tools in these lectures to experiment with and verify both sequential (functional) and concurrent declarative programs, and also imperative concurrent programs.

Membership Equational Logic

Membership equational logic generalizes order-sorted equational logic. This generalization is explained in Section 11 of,

> J. Meseguer, "Membership Algebra as a Logical Framework for Equational Specification," pp. 18–61, Springer LNCS 1376, 1998.

where membership equational logic is also proposed, and it is shown to naturally extend order-sorted equational logic in a conservative way.

Membership Equational Logic (II)

We can informally describe the main additions involved in this logic extension by saying that:

- for each connected component C in the poset of sorts $(S, \leq)$ we add a new "error supersort" $k(C)$ at the top of the component, called the kind of the component.

- we "lift" all operators to kinds; that is, for each $f : s_1 \ldots s_n \longrightarrow s$, with $k(C_i)$ the kind of s_i and $k(C)$ the kind of s we add, $f : k(C_1) \ldots k(C_n) \longrightarrow k(C)$.

- we add for each sort $s \in S$ (but not for the kinds) a membership predicate, $_ : s$, which is interpreted as holding of an element a in an algebra A iff $a \in A_s$.

Membership Equational Logic (II)

Terms that have a kind, but do not have a sort in S are thought of as error, or undefined, terms. Membership equational logic gives us a general way of dealing with partiality within the total context provided by the kinds.

A theory (Σ, E) can, in addition to equations, also have memberships, and both equations and memberships can be conditonal, with the condition a conjunction of other equations and memberships. That is, we have axioms of the form,

$$(\forall X)\, t = t' \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$

$$(\forall X)\, t : s \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$

Membership Equational Logic (III)

The inference rules then extend those of (order-sorted) equational logic in a natural way, and are the following:

1. **Reflexivity**.

$$\frac{}{E \vdash (\forall X)\ t = t}$$

2. **Symmetry**.

$$\frac{E \vdash (\forall X)\ t = t'}{E \vdash (\forall X)\ t' = t}$$

3. **Transitivity**.

$$\frac{E \vdash (\forall X)\ t = t' \qquad E \vdash (\forall X)\ t' = t''}{E \vdash (\forall X)\ t = t''}$$

4. **Congruence**.

$$\frac{E \vdash (\forall X)\ t_1 = t'_1 \quad \ldots \quad E \vdash (\forall X)\ t_n = t'_n}{E \vdash (\forall X)\ f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}$$

where we assume that $f : k(C_1) \ldots k(C_n) \to k(C)$ is in Σ, and the terms $t_i, t'_i \in T_\Sigma(X)_{k(C_i)}$, $1 \leq i \leq n$.

5. **Membership**.

$$\frac{E \vdash (\forall X)\ t = t' \qquad E \vdash (\forall X)\ t : s}{E \vdash (\forall X)\ t' : s}$$

6. **Modus ponens**. Given a sentence

$$(\forall X)\ t = t' \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$

$$(\text{resp. } (\forall X)\ t : s \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m)$$

in the set E of axioms, and given an assignment $\theta : X \to T_\Sigma(Y)$, then

$$\frac{E \vdash (\forall Y)\ \overline{\theta}(u_i) = \overline{\theta}(v_i)\ \ 1 \leq i \leq n \qquad E \vdash (\forall Y)\ \overline{\theta}(w_j) : s_j\ \ 1 \leq j \leq m}{E \vdash (\forall Y)\ \overline{\theta}(t) = \overline{\theta}(t') \quad (\text{resp. } (\forall Y)\ \overline{\theta}(t) : s)}$$

Soundness, Completeness, and Initial Models

These inference rules are sound and complete.

Furthermore, any membership equational theory (Σ, E) has an initial algebra $T_{\Sigma/E}$, defined as a quotient of the term algebra T_Σ by:

- $t \equiv_E t' \quad \Leftrightarrow \quad E \vdash (\forall\emptyset)\ t = t'$
- $[t]_{\equiv_E} \in T_{\Sigma/E,s} \quad \Leftrightarrow \quad E \vdash (\forall\emptyset)\ t : s$

Simplification, Confluence, and Termination

As explained in,

> A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, "Specification and Proof in Membership Equational Logic," Theoretical Computer Science, 326:35–132, 2000,

all the results about equational simplification, confluence and termination extend in a natural way to membership equational logic theories.

Maude Functional Modules

Membership equational theories with initial algebra semantics can be specified as functional modules in Maude. The following module specifies palindrome lists.

```
fmod PALINDROME is protecting QID .
  sorts Pal List .
  subsorts Qid < Pal < List .
  op nil : -> Pal [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  ops rev : List -> List .
  vars I : Qid .
  var P : Pal .
  var L : List .
  mb I P I : Pal .
  eq rev(nil) = nil .
  eq rev(I L) = rev(L) I .
endfm
```

Maude Functional Modules (II)

`PALINDROME`'s axioms are confluent and terminating, (modulo associativity and identity) so that we can simplify expression with the `reduce` command.

```
reduce in PALINDROME : 'f 'o 'o 'o 'o 'f .
result Pal: 'f 'o 'o 'o 'o 'f

reduce in PALINDROME : rev('f 'o 'o 'o 'o 'f) == 'f 'o 'o 'o 'o 'f .
result Bool: true
```

Verification of Maude Functional Modules

We are now ready to begin discussing program verification for deterministic declarative programs, and, more specifically, for functional modules in Maude.

Notice that such functional modules are of the form $\texttt{fmod}(\Sigma, E \cup A)\texttt{endfm}$, where we assume E confluent and terminating modulo A. Their mathematical semantics is given by the initial algebra $T_{\Sigma/E \cup A}$.

Their operational semantics is given by equational simplification with E modulo A. Both semantics coincide in the so-called canonical term algebra (whose elements are simplified expressions) since we have the Σ-isomorphism,

$$T_{\Sigma/E \cup A} \cong Can_{\Sigma,E/A}.$$

Verification of Maude Functional Modules (II)

What are properties of a module $\texttt{fmod}(\Sigma, E \cup A)\texttt{endfm}$?

They are sentences φ, perhaps in equational logic, or, more generally, in first-order logic, in the language of a signature containing Σ.

When do we say that the above module satisfies property φ?

When we have,

$$T_{\Sigma/E \cup A} \models \varphi.$$

How do we verify such properties?

A Simple Example: Associativity of Addition

Consider the module,

```
fmod NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

A property φ satisfied by this module is the associativity of addition, that is, the equation,

$$(\forall \mathtt{N}, \mathtt{M}, \mathtt{L})\ \ \mathtt{N\ +\ (M\ +\ L)} = \mathtt{(N\ +\ M)\ +\ L}.$$

Need More than Equational Deduction

Associativity is not a property satisfied by all models of the equations E in NAT. Consider, for example, the initial model obtained by adding a nonstandard number a,

```
fmod NON-STANDARD-NAT is
  sort Nat .
  ops 0 a : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

Since it has the same equations E, this initial model satisfies E, but it does not satisfy associativity, since `a +(a + a)` $\neq$ `(a + a)+ a`. In fact, no equations apply to either side.

Inductive Properties

The point is that associativity is an inductive property of natural number addition; that is, one satisfied by the initial model of E, but not in general by other models of E.

What we need are inductive proof methods based on a more powerful proof system $\vdash_{ind}$, satisfying the soundness requirement,

$$E \cup A \vdash_{ind} \phi \quad \Rightarrow \quad T_{\Sigma/E\cup A} \models \phi.$$

Also, it should prove all that equational deduction can prove and more. That is, for formulas φ that are equations it should satisfy,

$$E \cup A \vdash \phi \quad \Rightarrow \quad E \cup A \vdash_{ind} \phi.$$

Inductive Properties (II)

Because of Gödel's Incompleteness Theorem we cannot hope to have completeness of inductive inference, that is, to have an equivalence

$$E \cup A \vdash_{ind} \phi \quad \Leftrightarrow \quad T_{\Sigma/E \cup A} \models \phi.$$

The structural induction inference system that we will use generalizes the usual proofs by natural number induction. In fact, in our example of associativity of natural number addition it actually specializes to the usual proof method by natural number induction.

Machine-Assisted Proof with Maude's ITP

Maude's ITP is an inductive theorem prover supporting proof by induction in Maude modules.

It is a program written entirely in Maude by Manuel Clavel in which one can:

- enter a module, together with a property we want to prove in that module, and
- give commands, corresponding to proof steps, to prove that property

For example, we enter the associativity of addition goal (stored, say, in a file `nat-assoc`) as follows

Machine-Assisted Proof with Maude's ITP (II)

```
loop init .

(goal
fmod NAT is
including BOOL .
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .
vars N M L : Nat .
eq N + 0 = N .
eq N + s(M) = s(N + M) .
endfm
|-ind {N ; M ; L}((N + (M + L)) = ((N + M) + L)) .)
```

Machine-Assisted Proof with Maude's ITP (III)

The tool then responds as follows, indicating that it is ready to prove the goal (numbered 1):

```
Maude> in nat-assoc



=================================
1
=================================



|-ind { N ; M ; L } ( N + ( M + L ) = ( N + M ) + L )
Maude>
```

Machine-Assisted Proof with Maude's ITP (VI)

We can then try prove goal 1 by induction on `L` giving the command `(ind (1) on L .)` and get the subgoals,

```
Maude> (ind (1) on L .)


+++++++++++++++++++++++++++++++++


=================================
1 . 1
=================================

|-ind { N:1 } ( { N ; M } ( N + ( M + N:1 ) = ( N + M ) + N:1 ) ==> { N ; M } (
    N + ( M + s ( N:1 ) ) = ( N + M ) + s ( N:1 ) ) )
=================================
1 . 2
=================================

|-ind { N ; M } ( N + ( M + 0 ) = ( N + M ) + 0 )
Maude>
```

Machine-Assisted Proof with Maude's ITP (VI)

We can then try prove the "base case" subgoal (1 . 2) by simplification, giving the command (simp (1 . 2).), which succeeds, leaving only goal (1 . 1) unproved

```
Maude> (simp (1 . 2). )

+++++++++++++++++++++++++++++++++

=================================
1 . 1
=================================


|-ind { N:1 } ( { N ; M } ( N + ( M + N:1 ) = ( N + M ) + N:1 ) ==> { N ; M } (
    N + ( M + s ( N:1 ) ) = ( N + M ) + s ( N:1 ) ) )
Maude>
```

Machine-Assisted Proof with Maude's ITP (V)

Finally, we can simplify the "induction step" subgoal with the command `(simp (1 . 1). )`, which succeeds and proves the theorem.

```
Maude> (simp (1 . 1). )

++++++++++++++++++++++++++++++++++

q.e.d

Maude>
```

Machine-Assisted Proof with Maude's ITP (VI)

The ITP has also a more powerful `ind+` command, which takes a step of induction and then automatically tries to simplify all the subgoals generated by that step. In this example, we can "blow away" the entire theorem (goal 1).

```
Maude> in nat-assoc

=================================
1
=================================

|-ind { N ; M ; L } ( N + ( M + L ) = ( N + M ) + L )
Maude> (ind+ (1) on L .)

+++++++++++++++++++++++++++++++++

q.e.d

Maude>
```

List Induction

So far, we have only used natural number induction. What about induction on other data structures? For example, what about list induction?

Consider, for example, the following module defining a list "append" operator `_@_` in terms of a list "cons" operator `_*_` for lists of Booleans,

```
fmod LIST-OF-BOOL is
  including BOOL .
  sort List .
  op nil : -> List [ctor] .
  op _*_ : Bool List -> List [ctor] .
  op _@_ : List List -> List .
  var B : Bool .
  vars L P Q : List .
  eq nil @ L = L .
  eq (B * L) @ P = (B * (L @ P)) .
endfm
```

Proving Append Associative

```
loop init .
(goal
fmod LIST-OF-BOOL is
including BOOL .
sort List .
op nil : -> List [ctor] .
op _*_ : Bool List -> List [ctor] .
op _@_ : List List -> List .
var B : Bool .
vars L P Q : List .
eq nil @ L = L .
eq (B * L) @ P = (B * (L @ P)) .
endfm
|-ind {L ; P ; Q}(((L @ P) @ Q) = (L @ (P @ Q))) .)
```

Proving Append Associative (II)

```
Maude> in append-assoc

=================================
1
=================================

|-ind { L ; P ; Q } ( ( L @ P ) @ Q = L @ ( P @ Q ) )
Maude> (ind+ (1) on L .)

+++++++++++++++++++++++++++++++++

q.e.d

Maude>
```

Using Lemmas

Life is not always as easy. Often, attempts at simplification do not succeed. However, they suggest lemmas to be proved. Trying to prove commutativity of addition suggests two lemmas that do the trick.

```
(goal
fmod NAT is
including BOOL .
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .
vars N M L : Nat .
eq N + 0 = N .
eq N + s(M) = s(N + M) .
endfm
|-ind {N ; M}((N + M) = (M + N)) .)
```

Structural Inductions and Other ITP Commands

The `ind` proof command corresponds to a structural induction inference step. For any membership equational theory it uses the constants, constructors and memberships in the module for the base case and the induction step.

Besides `simp`, and `lem`, other ITP proof commands include:

- `vrt` (proof in variety)
- `cns` (constants lemma)
- `split` and `split+` (reasoning by cases)
- `imp` (implication elimination)

Other Equational Reasoning Maude Tools

Besides the ITP tool, the following Maude tools, developed in joint work with Francisco Durán, Salvador Lucas, and Joe Hendrix, can be used to prove certain properties of equational specifications:

- *Church-Rosser Checker* (CRC): checks confluence assuming termination;
- *Maude Termination Tool* (MTT): checks termination of Maude specifications by theory transformations and calls to standard termination tools.
- *Sufficient Complenteness Checker* (SCC): checks that enough equations have been given to compute all the defined functions.

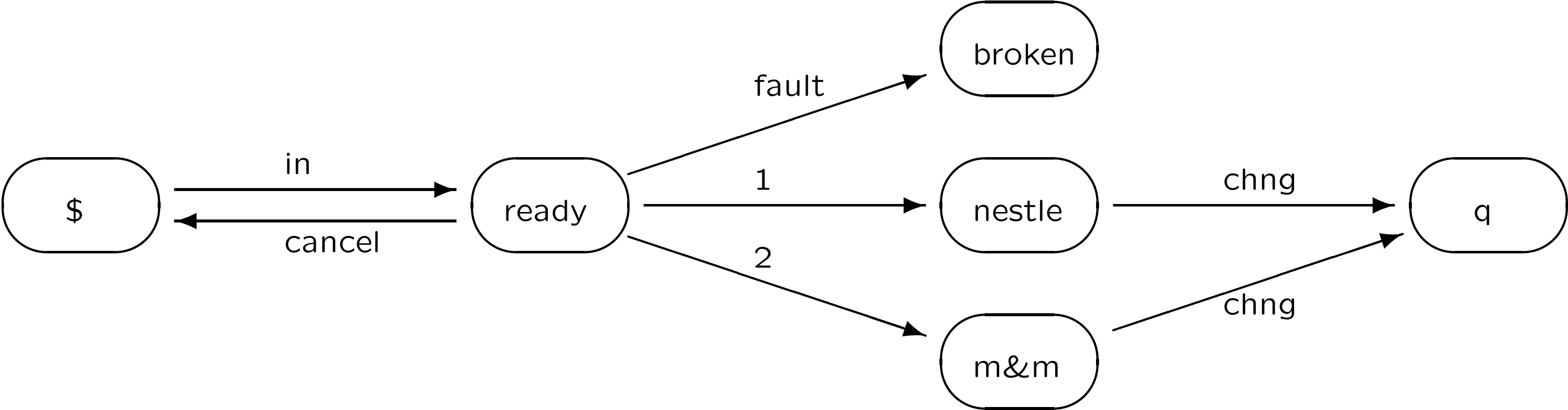
Software Specification and Verification in Rewriting Logic: Lecture 2

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Concurrency vs. Nondeterminism: Automata

We can motivate concurrency by its absence. The point is that we can have systems that are nondeterministic, but are not concurrent. Consider the following faulty automaton to buy candy:



Concurrency vs. Nondeterminism (II)

Although in the standard terminology this would be called a deterministic automaton (because each labeled transition from each state leads to a single next state) in reality it is still nondeterministic, in the sense that its computations are not confluent, and therefore completely different outcomes are possible.

For example, from the `ready` state the transitions `fault` and `1` lead to completely different states that can never be reconciled in a common subsequent state.

Concurrency vs. Nondeterminism (III)

So, the automaton is in this sense nondeterminisitc, yet it is strictly sequential, in the sense that, although at each state the automaton may be able to take several transitions, it can only take one transition at a time.

Since the intuitive notion of concurrency is that several transitions can happen simultaneously, we can conclude by saying the our automaton, although it exhibits a form of nondeterminism, has no concurrency whatsoever.

Automata as Rewrite Theories

In Maude we can specify such an automaton as,

```
mod CANDY-AUTOMATON is
  sort State .
  ops $ ready broken nestle m&m q : -> State .
  rl [in] : $ => ready .
  rl [cancel] : ready => $ .
  rl [1] : ready => nestle .
  rl [2] : ready => m&m .
  rl [fault] : ready => broken .
  rl [chng] : nestle => q .
  rl [chng] : m&m => q .
endm
```

Automata as Rewrite Theories

The above axioms are rewrite rules, but they do not have an equational interpretation. They are not understood as equations, but as transitions, that in general cannot be reversed.

This is just a simple example of a rewrite theory. In Maude such rewrite theories are declared in system modules, with keywords, `mod ... endm`.

The `rewrite` Command

Maude can execute such rewrite theories with the `rewrite` command (can be abbreviated to `rew`). For example,

```
Maude> rew $ .
rewrite in CANDY-AUTOMATON : $ .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result State: q
```

The `rewrite` command applies the rule in a fair way (all rules are given a chance) until termination, and gives one result.

The `rewrite` Command (II)

In this example, fairness saves us from nontermination, but in general we can esily have nonterminating computations.

For this reason the `rewrite` command can be given a numeric argument stating the maximum number of rewrite steps. For example,

The `rewrite` Command (III)

```
Maude> set trace on .
Maude> rew [3] $ .
rewrite [3] in CANDY-AUTOMATON : $ .
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
*********** rule
rl [cancel]: ready => $ .
empty substitution
ready ---> $
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result State: ready
```

The `search` Command

Of course, since we are in a nondeterministic situation, the `rewrite` command gives us one possible behavior among many.

To systematically explore all behaviors from an initial state we can use the `search` command, which takes two terms: a ground term which is our initial state, and a term, possibly with variables, which describes our desired target state.

Maude then does a breadth first search to try to reach the desired target state. For example, to find the terminating states from the `$` state we can give the command (where the "`!`" in `=>!` specifies that the target state must be a terminating state),

The `search` Command (II)

```
Maude> search $ =>! X:State .
search in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken

Solution 2 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

We can then inspect the search graph by giving the command,

The `search` Command (III)

```
Maude> show search graph .
state 0, State: $
arc 0 ===> state 1 (rl [in]: $ => ready .)

state 1, State: ready
arc 0 ===> state 0 (rl [cancel]: ready => $ .)
arc 1 ===> state 2 (rl [1]: ready => nestle .)
arc 2 ===> state 3 (rl [2]: ready => m&m .)
arc 3 ===> state 4 (rl [fault]: ready => broken .)

state 2, State: nestle
arc 0 ===> state 5 (rl [chng]: nestle => q .)

state 3, State: m&m
arc 0 ===> state 5 (rl [chng]: m&m => q .)

state 4, State: broken
state 5, State: q
```

The `search` Command (IV)

We can then ask for the shortest path to any state in the state graph (for example, state 5) by giving the command,

```
Maude> show path 5 .
state 0, State: $
===[ rl [in]: $ => ready . ]===>
state 1, State: ready
===[ rl [1]: ready => nestle . ]===>
state 2, State: nestle
===[ rl [chng]: nestle => q . ]===>
state 5, State: q
```

The `search` Command (V)

Similarly, we can search for target terms reachable by one rewrite step, one or more, or zero or more steps by typing (respectively):

- `search` t `=>` t' `.`
- `search` t `=>+` t' `.`
- `search` t `=>*` t' `.`

Furthermore, we can restrict any of those searches by giving an equational condition on the target term. For example, all terminating states reachable from `$` other than `broken` can be found by the command,

The `search` Command (VI)

```
Maude> search $ =>! X:State such that X:State =/= broken .
search in CANDY-AUTOMATON : $ =>! X:State
such that X:State =/= broken = true .

Solution 1 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

The search Command (VII)

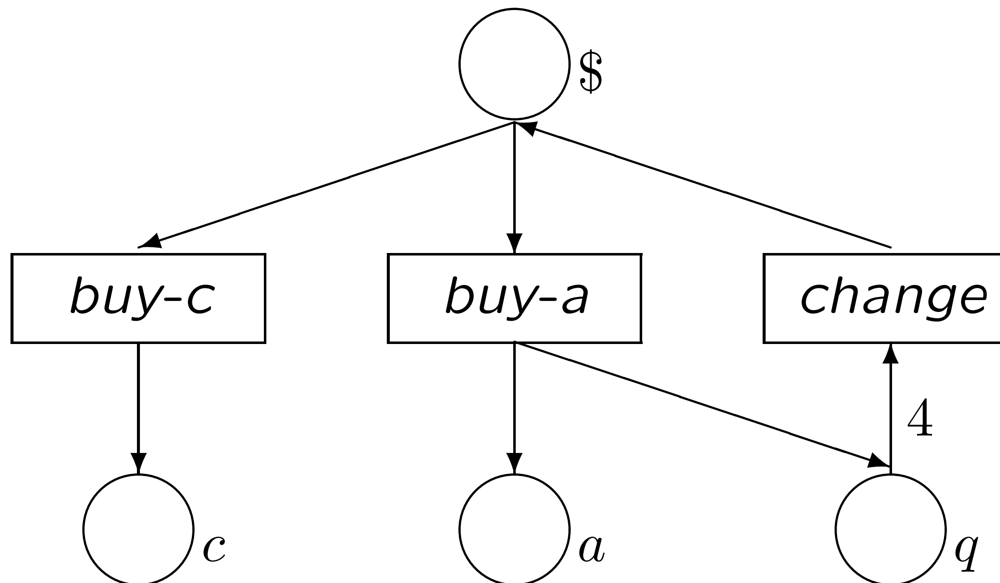
Of course, in general there can be an infinite number of solutions to a given search. Therefore, a search can be restricted by giving as an extra parameter in brackets the number of solutions (i.e., target terms that are instances of the pattern and satisfy the condition) we want:

```
search [1] in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken
```

Petri Nets

So far so good, but we have not yet seen any concurrency. Among the simplest concurrent system examples we have the concurrent automata called Petri nets. Consider for example the picture,



Petri Nets (II)

The previous picture represents a concurrent machine to buy cakes and apples; a cake costs a dollar and an apple three quarters.

Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar.

The machine is concurrent because we can push several buttons at once, provided enough resources exist in the corresponding slots, which are called places

Petri Nets (III)

For example, if we have one dollar in the $\$$ place, and four quarters in the q place, we can simultaneously push the *buy-a* and *change* buttons, and the machine returns, also simultaneously, one dollar in $\$$, one apple in a, and one quarter in q.

That is, we can achieve the concurrent computation,

$$\textit{buy-a} \;\; \textit{change} : \$ \, q \, q \, q \, q \longrightarrow a \, q \, \$.$$

Petri Nets (IV)

This has a straightforward expression as a rewrite theory (system module) as follows:

```
mod PETRI-MACHINE is
  sort Marking .
  ops null $ c a q : -> Marking .
  op _ _ : Marking Marking -> Marking [assoc comm id: null] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [chng] : q q q q => $ .
endm
```

Petri Nets (V)

That is, we view the distributed state of the system as a multiset of places, called a marking, with identity for multiset union the empty multiset `null`.

We then view a transition as a rewite rule from one (pre-)marking to another (post-)marking.

Petri Nets (VI)

The rewrite rule can be applied modulo associativity, commutativity and identity to the distributed state iff its pre-marking is a submultiset of that state.

Furthermore, if the distributed state contains the union of several such presets, then several transitions can fire concurrently.

For example, from `$ $ $` we can get in one concurrent step to `c c a q` by pushing twice (concurently!) the `buy-c` button and once the `buy-a` button.

Petri Nets (VII)

We can of course ask and get answers to questions about the behaviors possible in this system. For example, if I have a dollar and three quarters, can I get a cake and an apple?

```
Maude> search $ q q q =>+ c a M:Marking .
search in PETRI-MACHINE : $ q q q =>+ c a M:Marking .

Solution 1 (state 4)
states: 5 in 0ms cpu (0ms real)
M:Marking --> null
```

Another Rewrite Theory

Here is a simple rewrite theory. It consists of a single rewrite rule that allows choosing a submultiset in a multiset of elements.

```
mod CHOICE is
  sort MSet .
  ops a b c d e f g : -> MSet .
  op __ : MSet MSet -> MSet [assoc comm] .
  rl [choice] : X:MSet Y:MSet => Y:MSet .
endm
```

A Simple Rewrite Theory (II)

We can ask for all terminating computations, which correspond exactly to choosing the different elements of a given multiset,

```
Maude> search a b a c b c =>! X:MSet .
search in CHOICE : a b a c b c =>! X:MSet .

Solution 1 (state 23)
states: 26 in 0ms cpu (0ms real)
X:MSet --> c

Solution 2 (state 24)
states: 26 in 0ms cpu (0ms real)
X:MSet --> b

Solution 3 (state 25)
states: 26 in 0ms cpu (0ms real)
X:MSet --> a
```

Rewrite Theories in General

In general, a rewrite theory is a 4-tuple, $\mathcal{R} = (\Sigma, E, \Omega, R)$, where:

- (Σ, E) is a membership equational theory
- $\Omega \subseteq \Sigma$ is a subsignature
- R is a set of (universally quantified) labeled conditional rewrite rules of the form,

$$l : t \longrightarrow t' \ \Leftarrow \ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w_k').$$

Rewrite Theories in General (II)

The new requirement not discussed before is the subsignature $\Omega \subseteq \Sigma$. In all our previous examples we had $\Omega = \Sigma$, and this requirement was not needed.

The operators in $\Sigma - \Omega$ are called frozen operators, because they freeze the rewriting computations in the sense that no rewrite can take place below a frozen symbol.

We can illustrate frozen operators with the following example extending the `CHOICE` rewrite theory,

CHOICE-CARD

```
mod CHOICE-CARD is
  protecting INT .
  sorts Elt MSet .
  subsorts Elt < MSet .
  ops a b c d e f g : -> Elt .
  op __ : MSet MSet -> MSet [assoc comm] .
  op card : MSet -> Int [frozen] .
  eq card(X:Elt) = 1 .
  eq card(X:Elt M:MSet) = 1 + card(M:MSet) .
  rl [choice] : X:MSet Y:MSet => Y:MSet .
endm
```

`CHOICE-CARD` **(II)**

It does not make much sense to rewrite below the cardinality function `card`, because then the multiset whose cadinality we wish to determine becomes a moving target.

If `card` had not been declared frozen, then the rewrites, `a b c` $\longrightarrow$ `b c` $\longrightarrow$ `c` would induce rewrites, $3 \longrightarrow 2 \longrightarrow 1$, which seems bizarre.

The point is that we think of the kind `[MSet]` as the state kind in this example, whereas `[Int]` is the data kind. By declaring `card` frozen, we restrict rewrites to the state kind, where they belong.

Rewriting Logic in General

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \Omega, R)$, the sentences that it proves are universally quantified sentences of the form, $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_{\Sigma,E}(X)_k$, for some kind k, which are obtained by finite application of the following rules of deduction:

- **Reflexivity**. For each $t \in T_{\Sigma}(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$
- **Equality**.
$$\frac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X) u = u' \quad E \vdash (\forall X) v = v'}{(\forall X)\ u' \longrightarrow v'}$$
- **Congruence**. For each $f : k_1 \ldots k_n \longrightarrow k$ in Ω, with $t_i, t'_i \in T_{\Sigma}(X)_{k_i}$,
$$\frac{(\forall X)\ t_1 \longrightarrow t'_1 \quad \ldots \quad (\forall X)\ t_n \longrightarrow t'_n}{(\forall X)\ f(t_1, \ldots, t_n) \longrightarrow f(t'_1, \ldots, t'_n)}$$

- **Replacement**. For each finite substitution $\theta : X \longrightarrow T_\Sigma(Y)$, and for each rule in R of the form,

$$l : (\forall X)\, t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w'_k)$$

$$\frac{(\bigwedge_i \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j \theta(v_j) : s_j) \wedge (\bigwedge_k \theta(w_k) \longrightarrow \theta(w'_k))}{\theta(t) \longrightarrow \theta(t')}$$

- **Nested Replacement**. For each finite substitution $\theta : X \longrightarrow T_\Sigma(Y)$, with, say, $X = \{x_1, \ldots, x_n\}$, and $\theta(x_l) = p_l$, $1 \leq l \leq n$, and for each rule in R of the form,

$$l : (\forall X)\, t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w'_k)$$

with $t, t' \in T_\Omega(X)_k$ for some $k \in K$,

$$\frac{(\bigwedge_l p_l \longrightarrow p'_l) \wedge (\bigwedge_i \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j \theta(v_j) : s_j) \wedge (\bigwedge_k \theta(w_k) \longrightarrow \theta(w'_k}{\theta(t) \longrightarrow \theta'(t')}$$

where $\theta'(x_l) = p'_l,\ 1 \leq l \leq n$.

- **Transitivity**

$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

Comments on the Rules

Note that we have two replacement rules: a **Replacement** rule that does not involve rewrites in the substitution, and a **Nested Replacement** rule that does.

The introduction of two different rules is necessary because the terms t or t' could contain frozen operators, and in that case we want to disallow nested rewrites. Consequently, the **Nested Replacement** rule imposes the restriction $t, t' \in T_\Omega(X)_k$.

Comments on the Rules (II)

Of course, whenever we have $\Omega = \Sigma$, the **Replacement** rule becomes a special case of **Nested Replacement**.

Note, finally, that form the provability point of view the **Nested Replacement** rule is redundant, in that any proof $\mathcal{R} \vdash (\forall X)\, t \longrightarrow t'$ can be transformed into a proof of the same sentence not involving **Nested Replacement**. However, from a concurrency semantics perspective **Nested Replacement** isn't redundant, since it allows greater concurrency in computations than **Replacement** alone.

Concurrent Objects in Rewriting Logic

Rewriting logic can model very naturally many different kinds of concurrent systems. We have, for example, seen that Petri nets can be naturally formalized as rewrite theories. The same is true for many other models of concurrency such as CCS, the π-calculus, dataflow, real-time models, and so on.

One of the most useful and important classes of concurrent systems is that of concurrent object systems, made out of concurrent objects, which encapsulate their own local state and can interact with other objects in a variety of ways, including both synchronous interaction, and asynchronous communication by message passing.

Concurrent Objects in Rewriting Logic (II)

It is of course possibe to represent a concurrent object system as a rewrite theory with different modeling styles and adopting different notational conventions.

What follows is a particular style of representation that has proved useful and expressive in practice, and that is supported by Full Maude's object-oriented modules.

Concurrent Objects in Rewriting Logic (III)

To model a concurrent object system as a rewrite theory, we have to explain two things:

1. how the distributed states of such a system are equationally axiomatized and modeled by the initial algebra of an equational theory (Σ, E), and

2. how the concurrent interactions between objects are axiomatized by rewrite rules.

We first explain how the distributed states are equationally axiomatized.

Configurations

Let us consider the key state-building operations in Σ and the equations E axiomatizing the distributed states of concurrent object systems. The concurrent state of an object-oriented system, often called a configuration, has typically the structure of a multiset made up of objects and messages.

Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax (i.e. juxtaposition) as,

$$_\ _ : \mathbf{Conf} \times \mathbf{Conf} \longrightarrow \mathbf{Conf}.$$

Configurations (II)

The operator _ _ is declared to satisfy the structural laws of associativity and commutativity and to have identity `null`. Objects and messages are singleton multiset configurations, and belong to subsorts

$$\textbf{Object}\ \textbf{Msg} < \textbf{Conf},$$

so that more complex configurations are generated out of them by multiset union.

Configurations (III)

An object in a given state is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's attribute identifiers, and the v_i's are the corresponding values.

The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator $_ , _$ which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial.

Configurations (IV)

The value of each attribute shouldn't be arbitrary: it should have an appropriate sort, dictated by the nature of the attribute. Therefore, in Full Maude object classes can be declared in class declarations of the form,

$$\texttt{class}\ C \mid a_1 : s_1, \ldots, a_n : s_n\ .$$

where C is the class name, and s_i is the sort required for attribute a_i.

We can illustrate such class declarations by considering three classes of objects, `Buffer`, `Sender`, and `Receiver`.

Configurations (IV)

A buffer stores a list of integers in its `q` attribute. Lists of integers are built using an associative list concatenation operator, `_._` with identity `nil`, and integers are regarded as lists of length one. The name of the object reading from the buffer is stored in its `reader` attribute; such names belong to a sort `Oid` of object identifiers. Therefore, the class declaration for buffers is,

```
class Buffer | q : IntList, reader: Oid .
```

The sender and receiver objects store an integer in a `cell` attribute that can also be empty (`mt`) and have also a counter (`cnt`) attribute. The sender stores also the name of the receiver in an additional `receiver` attribute.

Configurations (V)

The class declarations are:

```
class Sender | cell: Int?, cnt: Int, receiver: Oid .
class Receiver | cell: Int?, cnt: Int .
```

where `Int?` is a supersort of `Int` having a new constant `mt`.

In Full Maude one can also give subclass declarations, with `subclass` syntax (similar to that of `subsort`) so that all the attributes and rewrite rules of a superclass are inherited by a subclass, which can have additional attributes and rules of its own.

Configurations (VI)

The messages sent by a sender object have the form,

```
(to Z : E from (Y,N))
```

where Z is the name of the receiver, E is the number sent, Y is the name of the sender, and N is the value of its counter at the time of the sending.

The syntax of messages is user-definable; it can be declared in Full Maude by message operator declarations. In our example by:

```
msg (to _ : _ from (_,_)) : Oid Int Oid Int -> Msg .
```

Object Rewrite Rules

We come now to explain (2): how the concurrent interactions between objects are axiomatized by rewrite rules.

The associativity and commutativity of a configuration's multiset structure make it very fluid. We can think of it as "soup" in which objects and messages float, so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind.

In general, the rewrite rules in R describing the dynamics of an object-oriented system can have the form,

Object Rewrite Rules (II)

$$
\begin{aligned}
r: \quad & M_1 \ldots M_n \, \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\
& \qquad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
& \qquad M'_1 \ldots M'_q \\
& \qquad \mathit{if} \ C
\end{aligned}
$$

where r is the label, the Ms are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition.

Object Rewrite Rules (III)

That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created.

If two or more objects appear in the lefthand side, we call the rule synchronous, because it forces those objects to jointly participate in the transition. If there is only one object and at most one message in the lefthand side, we call the rule asynchronous.

Object Rewrite Rules (IV)

Three typical rewrite rules involving objects in the `Buffer`, `Sender`, and `Receiver` classes are,

```
rl [read] : < X : Buffer | q: L . E, reader: Y >
                  < Y : Sender | cell: mt, cnt: N >
               => < X : Buffer | q: L, reader: Y >
                  < Y : Sender | cell: E, cnt: N + 1 >

rl [send] : < Y : Sender | cell: E, cnt: N, receiver: Z >
   => < Y : Sender | cell: mt, cnt: N > (to Z : E from (Y,N))

rl [receive] : < Z : Receiver | cell: mt, cnt: N >
                   (to Z : E from (Y,N))
                  => < Z : Receiver | cell: E, cnt: N + 1 >
```

where `E` and `N` range over `Int`, `L` over `IntList`, `X`, `Y`, `Z` over `Oid`, and `L.E` is a list with last element `E`.

Object Rewrite Rules (V)

Notice that the `read` rule is synchronous and the `send` and `receive` rules asynchronous.

Of course, these rules are applied modulo the associativity and commutativity of the multiset union operator, and therefore allow both object synchronization and message sending and receiving events anywhere in the configuration, regardless of the position of the objects and messages.

We can then consider the rewrite theory $\mathcal{R} = (\Sigma, E, \Omega, R)$ axiomatizing the object system with these three object classes, with R the three rules above (and perhaps other rules, such as one for the receiver to write its contents into another buffer object, that are omitted) and with Ω containing at least the multiset union operator.

Software Specification and Verification in Rewriting Logic: Lecture 3

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Verification of Declarative Concurrent Programs

We are now ready to discuss the subject of verification of declarative concurrent programs, and, more specifically, the verification of properties of Maude system modules, that is, of declarative concurrent programs that are rewrite theories.

There are two levels of specification involved: (1) a system specification level, provided by the rewrite theory and yielding an initial model for our program; and (2) a property specification level, given by some property (or properties) φ that we want to prove about our program. To say that our program satisfies the property φ then means exactly to say that its initial model does.

Verification of Declarative Concurrent Programs (II)

The question then becomes, which language shall we use to express the properties φ that we want to prove hold in the model $T_{Reach(\mathcal{R})}$? That is, how should we express relevant properties φ?

One possibility is to use the first-order language $FOL(Reach(\mathcal{R}))$ associated to the theory $Reach(\mathcal{R})$. But not all properties of interest are expressible in $FOL(Reach(\mathcal{R}))$: properties related to the infinite behavior of a system typically are not expressible in $FOL(Reach(\mathcal{R}))$.

For such properties we can use some kind of temporal logic. We will give particular attention to linear temporal logic (LTL) because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures.

Kripke Structures

The simplest models of LTL are called Kripke structures.

A binary relation $R \subseteq A \times A$ on a set A is called total iff for each $a \in A$ there is at least one $a' \in A$ such that $(a, a') \in R$. If R isn't total, it can be made total by defining, $R^{\bullet} = R \cup \{(a, a) \in A^2 \mid \nexists a' \in A \; (a, a') \in R\}$.

Definition. A *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set, called the set of states, $\rightarrow_{\mathcal{A}}$ 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.

Kripke Structures (II)

Note that the labeling function $L : A \longrightarrow \mathcal{P}(AP)$ specifies which propositions hold in which state. This of course is equivalent to specifying the semantics of each proposition p as a unary predicate on A:

$$A_p = \{a \in A \mid p \in L(a)\}$$

and conversely,

$$L(a) = \{p \in AP \mid a \in A_p\}$$

Propositional LTL

Given a set AP of atomic propositions, we define the propositional linear temporal logic $LTL(AP)$ inductively as follows:

- **Atomic Propositions**. $\top \in AP$; if $p \in AP$, then $p \in LTL(AP)$.
- **Next Operator**. If $\varphi \in LTL(AP)$, then $\bigcirc\varphi \in LTL(AP)$.
- **Until Operator**. If $\varphi, \psi \in LTL(AP)$, then $\varphi \, \mathcal{U} \, \psi \in LTL(AP)$.
- **Boolean Connectives**. If $\varphi, \psi \in LTL(AP)$, then the formulas $\neg\varphi$, and $\varphi \vee \psi$ are in $LTL(AP)$.

Paths in $\mathcal{A}$, and Models of $LTL(AP)$

Given a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$, the set $Path(\mathcal{A})$ of its paths is the set of functions of the form,

$$\pi : \mathbb{N} \longrightarrow A$$

such that for each $n \in \mathbb{N}$ we have,

$$\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$$

The models of the logic $LTL(AP)$ are the different Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ that have AP as their set of atomic propositions; that is, such that $L : A \longrightarrow \mathcal{P}(AP)$.

The Semantics of $LTL(AP)$

The binary satisfaction relation,

$$\mathcal{A} \models_{LTL} \varphi$$

by definition, holds iff for all $a \in A$ the ternary satisfaction relation,

$$\mathcal{A}, a \models_{LTL} \varphi$$

holds, which, again by definition, holds iff for all assignments $a \in A$, and all paths $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$, the quaternary satisfaction relation holds,

$$\mathcal{A}, a, \pi \models_{LTL} \varphi.$$

The Semantics of $LTL(AP)$ (II)

So, in the end we can boil everything down to defining the quaternary satisfaction relation

$$\mathcal{A}, a, \pi \models_{LTL} \varphi.$$

with $a \in A$, and $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$. We now proceed to giving the definition of this quaternary satisfaction relation in the usual inductive way:

- We always have, $\mathcal{A}, a, \pi \models_{LTL} \top$.

- For $p \in AP$,

$$\mathcal{A}, a, \pi \models_{LTL} p \quad \Leftrightarrow \quad p \in L(a).$$

The Semantics of $LTL(AP)$ (III)

- For $\bigcirc\varphi \in LTL(A)$,

$$\mathcal{A}, a, \pi \models_{LTL} \bigcirc\varphi \quad \Leftrightarrow \quad \mathcal{A}, \pi(1), s; \pi \models_{LTL} \varphi.$$

- For $\varphi\,\mathcal{U}\,\psi \in LTL(\mathcal{A})$,

$$\mathcal{A}, a, \pi \models_{LTL} \varphi\,\mathcal{U}\,\psi \quad \Leftrightarrow$$

$$\Leftrightarrow \quad (\exists n \in \mathbb{N})\ ((\mathcal{A}, \pi(n), s^n; \pi \models_{LTL} \psi)\wedge$$

$$\wedge((\forall m \in \mathbb{N})\ m < n\ \Rightarrow \mathcal{A}, \pi(m), s^m; \pi \models_{LTL} \varphi)).$$

The Semantics of $LTL(AP)$ (IV)

- For $\neg\varphi \in LTL(AP)$,

$$\mathcal{A}, a, \pi \models_{LTL} \neg\varphi \quad \Leftrightarrow \quad \mathcal{A}, a, \pi \not\models_{LTL} \varphi.$$

- For $\varphi \vee \psi \in LTL(AP)$,

$$\mathcal{A}, a, \pi \models_{LTL} \varphi \vee \psi \quad \Leftrightarrow$$

$$\Leftrightarrow \quad \mathcal{A}, a, \pi \models_{LTL} \varphi \text{ or } \mathcal{A}, a, \pi \models_{LTL} \psi.$$

Other $LTL(AP)$ Connectives

Other LTL connectives can be defined in term of the above minimal set of connectives as follows:

Other Boolean Connectives:

$$\bot = \neg\top$$

$$\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$$

$$\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$$

Other $LTL(AP)$ Connectives (II)

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: $\varphi \leadsto \psi = \Box(\varphi \rightarrow (\Diamond\psi))$

LTL Properties of Rewrite Theories

How can we associate LTL properties to a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$? We just need to make explicit two things: (1) the intended kind k of states in the signature Σ; and (2) the relevant state predicates.

In general, the state predicates need not be part of the system specification $\mathcal{R}$. They are typically part of the property specification.

LTL Properties of Rewrite Theories (II)

We can assume that state predicates have been defined by means of equations D in an equational theory $(\Sigma', E \cup D)$ extending (Σ, E) as a subtheory in protecting[a] mode.

We may also assume that the syntax defining the state predicates consists of a subsignature $\Pi \subseteq \Sigma'$ of function symbols, with each $p \in \Pi$ a different state predicate symbol that can be parameterized, that is, p need not be a constant, but can in general be an operator $p : s_1 \ldots s_n \longrightarrow Prop$.

[a] By definition, being protecting means that the unique Σ-homomorphism $h : T_{\Sigma/E} \longrightarrow T_{\Sigma'/E \cup D}|_{\Sigma}$ ensured by the initiality of $T_{\Sigma/E}$ restricts for each sort s in Σ to a bijective function $h_s : T_{\Sigma/E,s} \longrightarrow T_{\Sigma'/E \cup D,s}$.

LTL Properties of Rewrite Theories (III)

It is also useful to assume that, if k is the kind of states, the *semantics* of the state predicates Π is defined with the help of an operator,

$$_ \models _ : k \; [Prop] \longrightarrow [Bool]$$

in the signature Σ' (with $[Prop]$ and $[Bool]$ the kinds of $Prop$ and $Bool$) and by the equations $D \cup E$. Specifically, given a term t of kind k denoting a state, and a (possibly parametric) state predicate $p(u_1, \ldots, u_n)$, with $u_1, \ldots, u_n$ ground terms, we say that the state predicate $p(u_1, \ldots, u_n)$ holds in the state $[t]$ iff,

$$E \cup D \; \vdash \; (\forall \, \emptyset) \; t \models p(u_1, \ldots, u_n) = true.$$

LTL Properties of Rewrite Theories (IV)

In practice we typically want more. We want the equality $t \models p(u_1, \ldots, u_1) = true$ to be decidable. This can be achieved by making sure that $D \cup E$ is a set of confluent, sort-decreasing, and terminating equations and memberships (perhaps modulo some axioms).

Note that, since the equations are assumed confluent, sort-decreasing, and terminating, only the case when the predicate holds needs be specified by the equations $D \cup E$: when a ground expression $t \models p(u_1, \ldots, u_n)$ cannot be simplified to $true$, then by definition the predicate does not hold.

LTL Properties of Rewrite Theories (V)

We are now ready to associate to a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ (with a selected kind of states and with state predicates Π) a Kripke structure whose atomic predicates are specified by the set

$$AP_{\Pi} = \{\theta(p) \mid p \in \Pi,\ \theta \text{ ground substitution}\},$$

where by convention we use the simplified notation $\theta(p)$ to denote the ground term $\theta(p(x_1, \ldots, x_n))$. This defines a labeling function L_{Π} on the set of states $T_{\Sigma/E,k}$ assigning to each $[t] \in T_{\Sigma/E,k}$ the set of atomic propositions,

$$L_{\Pi}([t]) = \{\theta(p) \in AP_{\Pi} \mid (E \cup D) \vdash\ (\forall\, \emptyset)\ t \models \theta(p) = \mathit{true}\}.$$

LTL Properties of Rewrite Theories (VI)

The Kripke structure we are interested in is then,

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

with $(\rightarrow^1_{\mathcal{R}})^\bullet$ the total relation extending the one-step $\mathcal{R}$-rewriting relation $\rightarrow^1_{\mathcal{R}}$ for states of kind k.

By definition, given a formula $\varphi \in LTL(AP_\Pi)$, the system specified by $\mathcal{R}$ (with a selected kind k of states and with state predicates Π) satisfies φ beginning at an initial state $[t] \in T_{\Sigma/E,k}$ iff,

$$\mathcal{K}(\mathcal{R},k)_\Pi, [t] \models_{LTL} \varphi.$$

The Maude Model Checker

Maude support on-the-fly LTL model checking for initial states t of a rewrite theory $\mathcal{R}$ such that the set of all states reachable from t is finite.

Note the many rewrite theories of interest may have an infinite number of states, yet the states reachable from any given initial state may still be finite.

For example, rewriting logic biological models of the cell typically satisfy the above property. This is so essentially because of the conservation of matter property in chemical reactions, together with the physical limits on the amount of cell material membranes can hold inside (which limits their exchange of materials with their environment). However, the number of cell models can be infinite.

The Maude Model Checker (II)

Given a rewrite theory $\mathcal{R}$ satisfying the assumptions already mentioned when defining the logic $LTL(\mathcal{R})_{State}$ and specified in Maude by a system module, say `M`, and given an initial state, say `init` of sort $\texttt{State}_{\texttt{M}}$, we can model check different LTL properties beginning at this initial state by doing the following:

- defining a new module, say `CHECK-M`, that includes the modules `M` and `MODEL-CHECKER` as submodules (we can include other submodules as well if we wish, for example to introduce auxiliary data types and functions);

The Maude Model Checker (III)

- giving a subsort declaration, `subsort` $\mathtt{State}_{\mathrm{M}}$ `< State .`, where `State` is one of the key sorts in the module `MODEL-CHECKER` (this declaration can be omitted if $\mathtt{State}_{\mathrm{M}} = \mathtt{State}$);

- defining the syntax of the state predicates we wish to use by means of constants and operators of sort `Prop` (a subsort of the sort `Formula` (i.e., LTL formulas) in the module `MODEL-CHECKER`); we can define parameterless state predicates as constants of sort `Prop`, and parameterized state predicates by operators from the sorts of their parameters to the `Prop` sort.

The Maude Model Checker (IV)

- defining the semantics of the state predicates by means of equations involving the operator

```
op _|=_ : State Prop -> Result [special ... ] .
```

in `MODEL-CHECKER`. The sort `Result` is a supersort of `Bool`. We define the semantics of each state predicate, say a parameterized state predicate `p`, by giving (possibly conditional) equations of the form:

```
ceq exp1 |= p(u11,...,un1) = true if C1 .
...
ceq expk |= p(u1k,...,unk) = true if Ck .
```

where:

- the `expi`, $1 \leq i \leq k$, are patterns of sort $\texttt{State}_{\texttt{M}}$, that is, terms, possibly with variables, and involving only constructors, so that any of their instances by simplified ground terms cannot be further simplified;
- the terms `p(u1i,...,uni)`, $1 \leq i \leq k$ are likewise patterns of sort `Prop`;
- each condition `Ci`, $1 \leq i \leq k$, is a conjunction of equalities and memberships; such conditions may involve auxiliary functions, either imported from auxiliary modules, or defined by additional equations in our module `CHECK-M`.

The Maude Model Checker (V)

- once the semantics of each of the state predicates has been defined, we are then ready, given an initial state `init`, to model check any LTL formula, say `form`, involving such predicates; such LTL formulas are ground terms of sort `Formula` in `CHECK-M`; we do so by giving the Maude command,

  ```
  red init |= form .
  ```

  assuming, as already mentioned, that the set of reachable states is finite (in fact, finite enough to fit into memory in those cases when the on-the-fly model checking procedure has to search all of it).

The Maude Model Checker (VI)

Two things can then happen: if the property `form` holds we get the result `true`; if it doesn't, we get a counterexample, expressed with the syntax,

```
op counterExample : TransitionList TransitionList -> Result [ctor] .
```

This is because any counterexample to an LTL formula can be expressed as a path of transitions followed by a cycle; therefore the first argument of the above constructor is the path leading to the cycle, and the second is the cycle itself. Each transition is represented as a pair, consisting of a state and a rule label.

Note that we have defined the syntax and semantics of the state predicates in such a way that their state argument is left implicit, appearing as the first argument of _|=_.

The Maude Model Checker (VII)

For example, a parameterized state predicate `p` with parameters of sorts `S1, ... Sm` is defined by an operator,

```
op p : S1 ... Sm -> Prop .
```

instead than by an operator

```
op p : State S1 ... Sm -> Prop .
```

Note that the semantic equations in the first syntax,

```
ceq exp1 |= p(u11,...,un1) = true if C1 .
...
ceq expk |= p(u1k,...,unk) = true if Ck .
```

The Maude Model Checker (VIII)

correspond exactly to the equations,

```
ceq p(exp1,u11,...,un1) = true if C1 .
...
ceq p(expk,u1k,...,unk) = true if Ck .
```

in the second syntax. This observation helps clarify a further nice feature about how state predicates are specified in the LTL model checker, namely that only the positive cases have to be specified; that is, if a state predicate ground expression of the form `exp |= p(w1,...,wk)` (equivalent to `p(exp,w1,...,expk)` in the second syntax) cannot be simplified to `true`, then it is assumed to be false.

The LTL Syntax

The model checker's LTL syntax is defined by the following functional module LTL imported by MODEL-CHECKER

```
fmod LTL is sort Formula .
  *** primitive LTL operators
  ops True False : -> Formula [ctor] .
  op ~_ : Formula -> Formula [ctor prec 53] .
  op _/\_ : Formula Formula -> Formula [comm ctor gather (E e) prec 55] .
  op _\/_ : Formula Formula -> Formula [comm ctor gather (E e) prec 59] .
  op O_ : Formula -> Formula [ctor prec 53] .
  op _U_ : Formula Formula -> Formula [ctor prec 65] .
  op _R_ : Formula Formula -> Formula [ctor prec 65] .

  *** defined LTL operators
  op _->_ : Formula Formula -> Formula [gather (e E) prec 61] .
  op _<->_ : Formula Formula -> Formula [prec 61].
  op <>_ : Formula -> Formula [prec 53] .
  op []_ : Formula -> Formula [prec 53] .
```

```
  op _W_ : Formula Formula -> Formula [prec 65] .
  op _|->_ : Formula Formula -> Formula [prec 65] . *** leads-to

  vars f g : Formula .
  eq f -> g = ~ f \/ g .
  eq f <-> g = (f -> g) /\ (g -> f) .
  eq <> f = True U f .
  eq [] f = False R f .
  eq f W g = (f U g) \/ [] f .
  eq f |-> g = [](f -> (<> g)) .

  *** negative normal form
  eq ~ True = False .
  eq ~ False = True .
  eq ~ ~ f = f .
  eq ~ (f \/ g) = ~ f /\ ~ g .
  eq ~ (f /\ g) = ~ f \/ ~ g .
  eq ~ O f = O ~ f .
  eq ~(f U g) = (~ f) R (~ g) .
  eq ~(f R g) = (~ f) U (~ g) .
endfm
```

The LTL Syntax (II)

Note that the equations in this module do two things:

- express all defined LTL operators in terms of the basic operators `True`, `False`, negation, conjunction, disjunction, next, `O`, until, `U`, and release, `R`

- transform the LTL formula using only those basic operators into an equivalent one in negative nomal form, that is, the negations are pushed all the way down into the state predicates

It is for this reason that we also need `False` and the conjunction and release operators (dual to `True`, disjunction, and until) among our basic operators.

Tableau Generation

The negation of the LTL formula that we wish to model check is put in negative normal form and is used to generate a tableau from it, in the sense of Clark-Grumberg-Peled's "Model Checking" Section 6.7. Specifically, the LTL model checker expresses that tableau as a Büchi automaton in the way explained in Clark-Grumberg-Peled's "Model Checking" Section 9.4–5. The LTL model checker then searches on-the-fly the product of the tableau for the negated formula and the reachability model for the module `M` to find a counterexample.

The user can optionally import also a module `LTL-SIMPLIFIER` in `MODEL-CHECKER` that tries to further rearrange and simplify the negative normal form to generate a smaller Büchi automaton from it.

A Mutual Exclusion Example

The use of the Maude LTL model checker can be illustrated with a mutual exclusion example.

```
(omod MUTEX is sort Mode .
  ops a b : -> Oid .
  msgs * $ : -> Msg .
  ops wait critical : -> Mode .
  class Proc | mode : Mode .
  rl [a-enter] : $ < a : Proc | mode : wait > =>
                            < a : Proc | mode : critical > .
  rl [b-enter] : * < b : Proc | mode : wait > =>
                            < b : Proc | mode : critical > .
  rl [a-exit] : < a : Proc | mode : critical > =>
                            < a : Proc | mode : wait > * .
  rl [b-exit] : < b : Proc | mode : critical > =>
                            < b : Proc | mode : wait > $ .
endom)
```

A Mutual Exclusion Example (II)

We define two initial states, and parametrized state predicates `crit` and `wait` and define their semantics in the module,

```
mod MUTEX-CHECK is inc MUTEX . inc MODEL-CHECKER .
subsort Configuration < State .
ops crit wait : Oid -> Prop .
ops initial1 initial2 : -> Configuration .

var o : Oid .
var C : Configuration .

eq  < o : Proc | mode : critical > C |= crit(o) = true .
eq  < o : Proc | mode : wait  > C |= wait(o) = true .

eq initial1 = $ < a : Proc | mode : wait > < b : Proc | mode : wait > .
eq initial2 = * < a : Proc | mode : wait > < b : Proc | mode : wait > .
endm
```

A Mutual Exclusion Example (III)

We can then model check the mutual exclusion property for MUTEX with these intial states by submitting to Maude the following:

```
==========================================
reduce in MUTEX-CHECK : initial1 |= []~ (crit(a) /\ crit(b)) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= []~ (crit(a) /\ crit(b)) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
```

A Mutual Exclusion Example (IV)

The strong liveness property that waiting infinitely often implies acquiring the resource infinitely often:

```
reduce in MUTEX-CHECK : initial1 |= []<> wait(a) -> []<> crit(a) .
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial1 |= []<> wait(b) -> []<> crit(b) .
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= []<> wait(a) -> []<> crit(a) .
rewrites: 28 in 10ms cpu (10ms real) (2800 rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= []<> wait(b) -> []<> crit(b) .
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

A Mutual Exclusion Example (V)

Each process enters its critical section infinitely often:

```
==========================================
reduce in MUTEX-CHECK : initial1 |= True |-> crit(a) .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial1 |= True |-> crit(b) .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= True |-> crit(a) .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= True |-> crit(b) .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
```

A Mutual Exclusion Example (VI)

Both processes will simultaneously be in their wait section infinitely often:

```
==========================================
reduce in MUTEX-CHECK : initial1 |= True |-> wait(a) /\ wait(b) .
rewrites: 23 in 10ms cpu (10ms real) (2300 rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= True |-> wait(a) /\ wait(b) .
rewrites: 23 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
```

A Mutual Exclusion Example (VII)

If one process enters its critical section, then two steps later the other process will do so too:

```
reduce in MUTEX-CHECK : initial1 |= crit(a) -> O O crit(b) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= crit(a) -> O O crit(b) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial1 |= crit(b) -> O O crit(a) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in MUTEX-CHECK : initial2 |= crit(b) -> O O crit(a) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Dekker's Mutex Algorithm

One of the earliest correct solutions to the mutual exclusion problem was given by Dekker with his algorithm. The algorithm assumes processes that execute concurrently on a shared memory machine and communicate with each other through shared variables.

There are two processes, `p1` and `p2`. Process 1 sets a Boolean variable `c1` to 1 to indicate that it wishes to enter its critical section. Process `p2` does the same with variable `c2`. If one process, after setting its variable to 1 finds that the variable of its competitor is 0, then it enters its critical section rightaway. In case of a tie (both variables set to 1) the tie is broken using a variable `turn` that takes values in $\{1, 2\}$.

Dekker's Mutex Algorithm (II)

The code of process 1 is as follows,

```
repeat
  c1 := 1 ;
  while c2 = 1 do
    if turn = 2 then
      c1 := 0 ;
      while turn = 2 do skip od ;
      c1 := 1
    fi
  od ;
  crit ;
  turn := 2 ;
  c1 := 0 ;
  rem
forever .
```

Dekker's Mutex Algorithm (III)

The code of process 2 is entirely symmetric:

```
repeat
  c2 := 1 ;
  while c1 = 1 do
    if turn = 1 then
      c2 := 0 ;
      while turn = 1 do skip od ;
      c2 := 1
    fi
  od ;
  crit ;
  turn := 1 ;
  c2 := 0 ;
  rem
forever .
```

The Semantics of Dekker's Algorithm

To subject Dekker's algorithm to a model checking analysis we first need somehow to specify precisely the semantics of the parallel language in which it is written.

This can be done by specifying such a semantics as a rewrite theory. The approach presented here is the one by Steven Eker, who defines in Maude the semantics of a simple parallel language expressive enough to write Dekker's algorithm in it.

First, a model of the memory itself has to be developed; then the syntax of the programs used by the processes is defined. All this can be done in functional modules `MEMORY`, `TESTS`, and `SEQUENTIAL`.

The Semantics of Dekker's Algorithm (II)

```
fmod MEMORY is
  inc INT .
  inc QID .

  sorts Memory .
  op none : -> Memory .
  op __ : Memory Memory -> Memory [assoc comm id: none] .
  op [_,_] : Qid Int -> Memory .
endfm
```

The Semantics of Dekker's Algorithm (III)

```
fmod TESTS is
  inc MEMORY .

  sort Test .
  op _=_ : Qid Int -> Test .
  op eval : Test Memory -> Bool .

  var Q : Qid .
  var M : Memory .
  vars N N' : Int .

  eq eval(Q = N, [Q, N'] M) = N == N' .
endfm
```

The Semantics of Dekker's Algorithm (IV)

```
fmod SEQUENTIAL is
  inc TESTS .

  sorts UserStatement Program .
  subsort UserStatement < Program .
  op skip : -> Program .
  op _;_ : Program Program -> Program [prec 61 assoc id: skip] .
  op _:=_ : Qid Int -> Program .
  op if_then_fi : Test Program -> Program .
  op while_do_od : Test Program -> Program .
  op repeat_forever : Program -> Program .
endfm
```

The Semantics of Dekker's Algorithm (V)

Using the above modules, we can then define our simple parallel language in a system module `PARALLEL`. The global state is a triple consisting of:

1. a "soup" (set) of processes;
2. the shared memory; and
3. a process identifier recording the last process that touched the memory or, in any event, performed some computation.

Processes themselves are pairs having a process identifier and a program.

The Semantics of Dekker's Algorithm (VI)

```
mod PARALLEL is
  inc SEQUENTIAL .
  inc TESTS .

  sorts Pid Process Soup MachineState .
  subsort Process < Soup .
  op [_,_] : Pid Program -> Process .
  op empty : -> Soup .
  op _|_ : Soup Soup -> Soup [prec 61 assoc comm id: empty] .
  op {_,_,_} : Soup Memory Pid -> MachineState .

  vars P R : Program .
  var S : Soup .
  var U : UserStatement .
  vars I J : Pid .
  var M : Memory .
  var Q : Qid .
  vars N X : Int .
  var T : Test .
```

The Semantics of Dekker's Algorithm (VII)

The language's operational semantics is then given by just five rules.

```
rl {[I, U ; R] | S, M, J} => {[I, R] | S, M, I} .

rl {[I, (Q := N) ; R] | S, [Q, X] M, J} =>
      {[I, R] | S, [Q, N] M, I} .

rl {[I, if T then P fi ; R] | S, M, J} =>
      {[I, if eval(T, M) then P else skip fi ; R] | S, M, I} .

rl {[I, while T do P od ; R] | S, M, J} =>
      {[I, if eval(T, M) then (P ; while T do P od) else skip fi ; R]
       | S, M, I} .

rl {[I, repeat P forever ; R] | S, M, J} =>
      {[I, P ; repeat P forever ; R] | S, M, I} .
```

The Semantics of Dekker's Algorithm (VIII)

We can then define the two processes for Dekker's algorithm and the desired initial state in the following module extending `PARALLEL`:

```
mod DEKKER is
  inc PARALLEL .
  subsort Int < Pid .
  ops crit rem : -> UserStatement .
  ops p1 p2 : -> Program .
  op initialMem : -> Memory .
  op initial : -> MachineState .
```

```
eq p1 =
      repeat
        'c1 := 1 ;
        while 'c2 = 1 do
          if 'turn = 2 then
            'c1 := 0 ;
            while 'turn = 2 do skip od ;
            'c1 := 1
          fi
        od ;
        crit ;
        'turn := 2 ;
        'c1 := 0 ;
        rem
      forever .
```

```
  eq p2 =
        repeat
          'c2 := 1 ;
          while 'c1 = 1 do
            if 'turn = 1 then
              'c2 := 0 ;
              while 'turn = 1 do skip od ;
              'c2 := 1
            fi
          od ;
          crit ;
          'turn := 1 ;
          'c2 := 0 ;
          rem
        forever .

  eq initialMem = ['c1, 0] ['c2, 0] ['turn, 1] .
  eq initial = { [1, p1] | [2, p2], initialMem, 0 } .
endm
```

Model Checking Dekker's Algorithm

We need to define two state predicates parameterized by the process id: `enterCrit`, when the process is about to enter its critical section, and `exec`, when the process has just executed.

```
mod CHECK is inc DEKKER . inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .  *** optional
  subsort MachineState < State .
  ops enterCrit exec : Pid -> Prop .
  var M : Memory .
  vars R : Program .
  var S : Soup .
  vars I J : Pid .

  eq {[I, crit ; R] | S, M, J} |= enterCrit(I) = true .
  eq {S, M, J} |= exec(J) = true .
endm
```

Model Checking Dekker's Algorithm (II)

The mutual exclusion property is satisfied:

```
==========================================
reduce in CHECK : initial |= []~ (enterCrit(1) /\ enterCrit(2)) .
ModelChecker: Property automaton has 2 states.
ModelSymbol: Examined 245 system states.
rewrites: 1052 in 30ms cpu (30ms real) (35066 rewrites/second)
result Bool: true
==========================================
```

Model Checking Dekker's Algorithm (III)

But the strong liveness property that executing infinitely often implies entering one's critical section infinitely often fails, as witnessed by the counterexample,

```
reduce in CHECK : initial |= []<> exec(1) -> []<> enterCrit(1) .
ModelChecker: Property automaton has 3 states.
ModelSymbol: Examined 16 system states.
rewrites: 148 in 10ms cpu (10ms real) (14800 rewrites/second)
result Result: counterExample(

 {{[1,repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn
   = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ;
   'turn := 2 ; 'c1 := 0 ; rem forever] | [2,repeat 'c2 := 1 ; while 'c1 = 1
   do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od
   ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,0] ['c2,0] ['turn,1],0},
   unlabeled}
```

```
{{[1,'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;
  while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
  ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;
  while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
  ; rem forever] | [2,repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn = 1 then
  'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1
  ; 'c2 := 0 ; rem forever],['c1,0] ['c2,0] ['turn,1],1},unlabeled}

{{[1,
  while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ;
  'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem ; repeat 'c1 := 1 ;
  while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ;
  'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [2,repeat
  'c2 := 1 ; while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do
  skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],[
  'c1,1] ['c2,0] ['turn,1],1},unlabeled}

{{[1,while 'c2 = 1 do if 'turn = 2
  then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn
  := 2 ; 'c1 := 0 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2
  then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn
```

:= 2 ; 'c1 := 0 ; rem forever] | [2,'c2 := 1 ; while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2,0] ['turn,1],2}, unlabeled}

{{[1,while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [2,while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2, 1] ['turn,1],2},unlabeled}

{{[1,if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;

```
  while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
  ; rem forever] | [2,while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while
  'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem
  ; repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while
  'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem
  forever],['c1,1] ['c2,1] ['turn,1],1},unlabeled}

{{[1,while 'c2 = 1 do if
  'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ;
  crit ; 'turn := 2 ; 'c1 := 0 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if
  'turn = 2 then 'c1 := 0 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ;
  crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [2,while 'c1 = 1 do if 'turn
  = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ;
  'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn =
  1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi od ; crit ;
  'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2,1] ['turn,1],1},
  unlabeled}

{{[1,while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn =
  2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem ; repeat
  'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ; while 'turn = 2 do
```

```
   skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem forever] | [
   2,if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 fi ;
   while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
   'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ;
   while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
   'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2,
   1] ['turn,1],2},unlabeled}

  {{[1,while 'c2 = 1 do if 'turn = 2 then 'c1 := 0
   ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 :=
   0 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;
   while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
   ; rem forever] | [2,'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 ;
   while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
   'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ;
   while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
   'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2,
   1] ['turn,1],2},unlabeled},
*** end of the path

*** remaining transitions in the loop
```

```
{{[1,if 'turn = 2 then 'c1 := 0 ; while 'turn =
 2 do skip od ; 'c1 := 1 fi ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;
 while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;
 while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
 ; rem forever] | [2,'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 ;
 while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ;
 while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2,
 1] ['turn,1],1},unlabeled}
{{[1,while 'c2 = 1 do if 'turn = 2 then 'c1 := 0
 ; while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 :=
 0 ; rem ; repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then 'c1 := 0 ;
 while 'turn = 2 do skip od ; 'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0
 ; rem forever] | [2,'c2 := 0 ; while 'turn = 1 do skip od ; 'c2 := 1 ;
 while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem ; repeat 'c2 := 1 ;
 while 'c1 = 1 do if 'turn = 1 then 'c2 := 0 ; while 'turn = 1 do skip od ;
 'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem forever],['c1,1] ['c2,
 1] ['turn,1],1},unlabeled})
```

Model Checking Dekker's Algorithm (IV)

However, the weaker liveness property that if both `p1` and `p2` execute infinitely often then both enter their critical sections infinitley often is true:

```
==========================================
reduce in CHECK : initial |= []<> exec(1) /\ []<> exec(2) -> []<> enterCrit(1)
    /\ []<> enterCrit(2) .
ModelChecker: Property automaton has 7 states.
ModelSymbol: Examined 245 system states.
rewrites: 1502 in 60ms cpu (60ms real) (25033 rewrites/second)
result Bool: true
```

By Way of Conclusion

In these lectures we have explored a general approach to software specification and verification based on equational logic (for deterministic programs) and on rewriting logic (for concurrent programs).

The approach, including the use of equational, rewriting logic, inductive, and temporal logic proof techniques and associated Maude tools has been shown applicable to the verification of:

- functional programs,
- imperative-concurrent programs, and
- rewriting logic concurrent programs.

By Way of Conclusion (II)

For the applicability to imperative sequential programs see CS 376 lectures at UIUC. The general pattern emerging from this approach is a distinction between:

- a system specification level, carried out in equational logic for deterministic systems, and in rewriting logic for concurrent ones. This level provides Maude executable specifications that can be symbolically simulated and analyzed to discover many bugs.
- a property specification level, in which properties expressed in, for example, first-order logic (for deterministic systems) and temporal logic (for concurrent ones) can be established with the aid of tools such as Maude's ITP and LTL Model Checker.

By Way of Conclusion (III)

The lectures have shown some advantages of using Maude as a declarative programming paradigm, both for functional programming, and for declarative concurrent programming.

Since programs in this approach a theories in equational or rewritign logic, programming can be done at a high level of abstraction, and it is considerably easier to verify such programs than to verify conventional ones.

All this can be done with high performance. Maude's semicompiled interpreter can perform a rewrite step in about 150 machine clock cycles; and a prototype Maude compiler in about 40 clock cycles. Also, it is possible and easy to execute Maude 2.3 in a distributed way.

Where to Go from Here

More extensive and detailed lecture notes on these ideas can be found in the web pages of the CS 476 course at the CS Department of the University of Illinois at Urbana-Champaign.

The Maude system, its documentation, proof tools, case studies, and many papers on membership equational logic, rewriting logic, and Maude can be obtained, free of charge, at:

`http://maude.csl.sri.com`