Software Specification and Verification in Rewriting Logic*

José Meseguer
Computer Science Department
University of Illinois at Urbana-Champaign

A mi madre, Fuensanta Guaita de Meseguer, en su 90 cumpleaños

Abstract

One can distinguish two specification levels: a *system specification* level, in which the computational system of interest is specified; and a *property specification* level, in which the relevant properties are specified. These lectures present an approach to *executable* system specification based on equational logic for deterministic systems and on rewriting logic for concurrent systems that is seamlessly integrated with a property specification level using first-order, inductive, and temporal logics. This integration is directly supported by formal verification tools in the formal environment of the Maude rewriting logic language. We show how this approach and the supporting tools can be applied to the specification and verification of a wide variety of programs, that can be either declarative or imperative, and either deterministic or concurrent.

Contents

*Research supported by ONR Grant N00014-02-1-0715, NSF Grant CCR-0234524, and by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130.

1 Introduction

These lecture notes summarize my present approach to both research and teaching in Formal Methods. They give a snapshot of a specific research program in this area, one shared with a good number of other colleagues [31], and give an account of *one approach* among many others. No attempt is made to make comparisons with other approaches: to do them justice would require a different work, and a style different from that of a small set of lectures. However, lack of explicit comparisons does not preclude judging, and being judged by, the relevant technical merits. Which are those merits in the present case? I think that they center around the fact of being a *unified and unifying*, yet eminently practical, approach. This seems a useful feature, given the always threatening *centrifugal forces* in scientific research. Specifically:

- Equational logic and its extension into rewriting logic allow a unified approach to the specification of sequential/deterministic, and concurrent/nondeterministic systems.
- Since under appropriate assumptions both these logics are efficiently executable, *declarative programming* and *executable formal specification* are naturally integrated and unified, avoiding the usual gap between specifications and programs.
- Furthermore, the proposed approach provides a seamless integration between an executable *system specification* level, carried out in equational and rewriting logic, and a *property specification* level using first-order, inductive, and temporal logics.
- This integration is directly supported by *formal tools* in Maude's formal environment. Declarative programming and executable formal specification are supported in Maude itself [6], and inductive theorem proving, model checking, and other formal analyses in either Maude or its associated formal tools [7, 13, 11, 10, 17].
- The approach is applicable to the verification and analysis of *many different kinds of programs*, which can be written in either declarative or imperative languages, and that can be either sequential or concurrent.

1.1 A Taxonomy of Programming Languages

Programs can be written in many different languages, having quite different styles. This in fact impacts the verification techniques suitable in each case, as well as the level of difficulty and effort involved in the verification task. A first useful distinction 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, in the sense that sequential programs are typically deterministic, whereas concurrent programs are often nondeterministic. Being independent of implementation details, the notion of a deterministic program is more general and abstract than that of a sequential program. Indeed, a deterministic program can be implemented on a sequential machine or on a parallel one.

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. One can always blur the above distinctions. This can be done in many ways. For example, some so-called declarative languages have plenty of *extra-logical features* which seriously compromise the possibility of reasoning mathematically about them in terms of their underlying logic. In such a case one should be honest and admit that the presumed declarative nature may at best belong to a language subset. Furthermore, there is always the Quixotic and amusing possibility of declaring that *everything is a logic!* including, say, C^{++}, arriving at a toothless notion of "logic". The possibilities for confusion and obscurantism are indeed endless; but such verbal games are for the most part a waste of time and will not occupy us in these lectures. For a general *mathematical* notion of logic and general axiomatic requirements for declarative languages see [32].

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* of P that we want to verify 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.

By contrast, imperative programs are *not* expressed in the language of mathematics. They are expressed in conventional programming languages like C, C^{++}, Java, and so on. Therefore, 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 a *mathematical semantics* for $\mathcal{L}$. This we can always do using the language of mathematics, perhaps in a semi-formal way. But for tool assistance purposes it is advantageous to formally axiomatize the semantics of $\mathcal{L}$ as a logical theory $T_{\mathcal{L}}$ in some logic. 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: $T_{\mathcal{L}} \models Q(P)$.

1.2 System Specification vs. Property Specification

Typically, there are two things that must be mathematically specified for a program verification task to be mathematically meaningful at all:

1. the *computational system* of interest, which could for example be a *program*, a *hardware system*, a *distributed algorithm*, an entire *programming language*, a *hardware description language*, and so on; and
2. the relevant *properties* that we want to verify about such a computational system.

It is therefore often useful to distinguish *two specification levels*: a *system specification* level, in which the computational system of interest is specified; and a *property specification* level, in which the relevant properties are specified. Although sometimes the specifications of a system and of its properties could take place in the *same* formalism, this *need not be so*: often it is more useful and flexible to specify the system and its properties in *different* (yet related) formalisms.

1.3 The Equational/Rewriting Logic Framework

A nontrivial question is *what logic* to use as the *framework logic* for system specification. There are many choices with different tradeoffs. In these lectures we will use *equational logic* to axiomatize the semantics of declarative deterministic programs, and *rewriting logic* to axiomatize the semantics of (declarative or imperative) concurrent programs. This choice of system specification logics 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* axiomatizations of imperative concurrent languages;
4. therefore, we can specify the four kinds of program styles in the cartesian product:

$$\{imperative, declarative\} \times \{sequential, concurrent\}$$

   in an executable way within the combined framework.

The $(imperative, sequential)$ case will not be treated explicitly in these lectures, due to space limitations. For a good textbook showing how equational logic can be used to give semantics and to reason about such programs, and discussing inductive proof techniques see [22]. For a treatment of this case that also addresses Hoare logic and is in complete continuity with these lectures see [37].

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 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. of $T_{\mathcal{L}}$.

The *property specification* level is a related but different matter. To specify program properties we will allow more expressive logics, such as full first-order logic (enriched with *induction principles* to reason about initial models) or even temporal logic (for concurrent programs).

1.4 Maude

Maude [6] is a declarative language and high-performance interpreter based on *rewriting logic* [33, 3] 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 that generalizes the OBJ3 language [21]. We will use Maude and its tools in these lectures to experiment with and verify both deterministic (functional) and concurrent declarative programs, and also imperative concurrent programs.

2 Specification and Verification of Deterministic Declarative Programs

We illustrate Maude's equational style of declarative functional programming, explain its underlying logic, and discuss verification of inductive program properties using Maude's inductive theorem prover.

2.1 Maude Functional Modules

Maude *functional modules* provide an equational style of functional programming. A functional module is an *equational theory* whose equations are used from left to right as *simplification rules* to reach a *fully reduced* or *canonical form*, provided the simplification process terminates. Furthermore, the equations are assumed to be *confluent*[1], so that such a canonical form, if it exists, is unique, thus ensuring *determinism*.

Maude supports equational simplification *modulo* any combination of *associative* (specified with the keyword `assoc`), and/or *commutative* (keyword: `comm`) operators, which can have also an *identity* element (keyword: `id:`). This means that equational simplification takes place not just between terms, but between *equivalence classes of terms* modulo such equational axioms.

Furthermore, the equational theories can be *order-sorted* [20]. That is, we can declare both *sorts* (or types) and *subsorts*, that is, set-theoretic sort inclusions. The operators can then be *subsort overloaded*. That is, an operator can have several different typings, related in the subsort ordering. Finally, operator syntax is *user-definable*. It can be prefix, postfix, infix, or any "mixfix" combination, with underscores (_) indicating argument positions. We can illustrate all these features with the following module for lists, that imports the predefined `QID` module of quoted identifiers.

```
fmod LIST is
  protecting QID .
  sorts List NeList .
  subsorts Qid < NeList < List .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op __ : List NeList -> NeList [ctor ditto] .
  op __ : NeList List -> NeList [ctor ditto] .
  op head : NeList -> Qid .
  op tail : NeList -> List .
  op rev : List -> List .
  var I : Qid .
  var L : List .
  eq head(I L) = I .
  eq tail(I L) = L .
  eq rev(nil) = nil .
  eq rev(I L) = rev(L) I .
endfm
```

The keywords declaring each entity are the obvious abbreviations; for example, `eq` begins the declaration of an equation. The `protecting` keyword is a *semantic statement* (indeed, a proof obligation) stating that the sort of quoted identifiers is not altered by the new sorts, operations, and equations. In particular, the `head` of a nonempty list can always be simplified to a quoted identifier. Note that list concatenation (specified here with "empty" juxtaposition syntax) has been declared associative and with `nil` as its identity element. Note also that it is subsort overloaded, with smaller typings defining the subsort `NeList` of nonempty lists (the `ditto` keywords makes it unnecessary to repeat the `assoc` and `id:` declarations). Finally, note that both `nil` and list concatenation are *constructors* (keyword: `ctor`) whereas the functions `head`, `tail`, and `rev` are *defined functions*, so that each *ground term* (that is, a term without variables) is simplified by the equations to a constructor term. In Maude we can evaluate expressions in a functional module with the `reduce` command. For example,

```
reduce in LIST : head('a 'b 'c 'd) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Qid: 'a
==========================================
reduce in LIST : tail('a 'b 'c 'd) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NeList: 'b 'c 'd
==========================================
reduce in LIST : rev('a 'b 'c 'd) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result NeList: 'd 'c 'b 'a
```

The order-sorted type structure allows defining functions such as `head` and `tail` that would be *partial*—or would require an ad-hoc, awkward definition—if `NeList` did not exist because we had only a many-sorted type structure.

[1] The equations are called *confluent* if whenever a term t can be simplified to two different terms u and v using the equations from left to right, then u and v can both be further simplified to a common term w.

Maude's equational logic is even more expressive. Besides allowing order-sorted equational specifications, it also allows *membership axioms* to specify very sophisticated subsorts by giving *semantic conditions* under which a term has a given sort. In general, such membership axioms (specified with the keyword `mb` when unconditional) can be *conditional*, with equations and memberships in their condition (and are then specified with the keyword `cmb`). We can illustrate *membership equational logic* specifications by means of the following `PALINDROME` module, which extends the above `LIST` module.

```
fmod PALINDROME is
  protecting LIST .
  sorts NePal Pal .
  subsorts Qid < NePal < Pal NeList < List .
  op nil : -> Pal [ctor] .
  var I : Qid .
  var P : Pal .
  mb I P I : NePal .
endfm
```

Palindromes are defined as the terms of sort `Pal`, with `NePal` the subsort of nonempty palindromes. According to the module's (initial algebra) definition, a list is a palindrome iff it is either

1. the `nil` list, or
2. a quoted identifier, or
3. a quoted identifier `I` concatenated with a smaller palindrome `P`, concatenated again with `I`,

and is a nonempty palindrome in the subcases 2–3. We can then evaluate expressions in this module such as the following:

```
reduce in PALINDROME : 'a 'b 'c 'd 'c 'b 'a .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result NePal: 'a 'b 'c 'd 'c 'b 'a
==========================================
reduce in PALINDROME : rev('a 'b 'c 'd 'c 'b 'a) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result NePal: 'a 'b 'c 'd 'c 'b 'a
```

2.2 Membership Equational Logic

The above module `PALINDROME` is a theory in *membership equational logic* [35, 2], a logic that generalizes *order-sorted* equational logic by allowing also membership axioms such as the one in `PALINDROME`.

A membership equational logic *signature* is a 3-tuple $\Sigma = (K, \Sigma, S)$, where:

- K is a set of *kinds*
- Σ is a $K^* \times K$-indexed family of function symbols $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$, where we denote each $f \in \Sigma_{k_1\ldots k_n,k}$ by $f : k_1 \ldots k_n \longrightarrow k$;
- $S = \{S_k\}$ is a disjoint K-indexed family of sets, whose elements are called *sorts*.

A Σ-*atomic formula* is either a Σ-*equation* $t = t'$, or a Σ-*membership* $t : s$, with $t, t' \in T_\Sigma(X)_k$, and $s \in S_k$.

A *membership equational theory* is a pair (Σ, E), with Σ a membership equational signature and E a set of *conditional* Σ-equations and Σ-memberships of the form,

$$\begin{array}{rcl} (\forall X)\; t = t' & \Leftarrow & u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m \\ (\forall X)\; t : s & \Leftarrow & u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m \end{array}$$

that is, the sentences in E are universally quantified *Horn clauses* whose atoms are Σ-atoms.

Note that, strictly speaking, in a membership equational logic signature Σ operators are not subsort overloaded, and subsort relations are not specified as such. Also, variables are declared at the kind level, not at the sort level. By contrast, in our PALINDROME module and in its LIST submodule we have specified several subsort relations and subsort overloaded operations, and variables have been given sorts. In fact, kinds as such have not been specified. All these differences are just inessential (but helpful) *notational conventions*. It is simply that, for convenience and to stress the close relationship with order-sorted equational logic, which can be regarded as a special case, membership equational theories are specified in Maude using the following *syntactic sugar conventions*:

- *Subsorts*. Given sorts $s, s' \in S_k$, the declaration $s < s'$ is syntactic sugar for the conditional membership $(\forall x : k)\ x : s' \Leftarrow x : s$.
- *Kinds* are not declared explicitly. They are *inferred by the system* and are identified with the *connected components* of the poset of sorts $(S, \leq)$, that is, with the equivalence classes of the smallest equivalence relation containing the subsort inclusion order $<$.
- *Operations*. If $f \in \Sigma_{k_1 \ldots k_n, k}$, and $s_1 \in S_{k_1}, \ldots, s_n \in S_{k_n}$, $s \in S_k$, then the declaration $f : s_1 \ldots s_n \longrightarrow s$ is syntactic sugar for $(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s \Leftarrow x_1 : s_1 \wedge \ldots \wedge x_n : s_n$.
- *Variables*. $(\forall x : s, X)\ a \Leftarrow a_1 \wedge \ldots \wedge a_n$ is syntactic sugar for the Σ-sentence $(\forall x : k, X)\ a \Leftarrow a_1 \wedge \ldots \wedge a_n \wedge x : s$, where $s \in S_k$ and a and the a_j are Σ-atoms. Then, $f : s_1 \ldots s_n \longrightarrow s$ is equivalent to $(\forall x_1 : s_1, \ldots, x_n : s_n)\ f(x_1, \ldots, x_n) : s$.

For example, the declarations,

```
subsort  NePal < Pal .
op __ : List List -> List [ctor assoc id: nil] .
var P : Pal .
var I : Qid .
mb I P I : NePal .
```

are syntactic sugar for declarations,

```
vars X Y I P : [List] .
cmb X : Pal if X : NePal .
op __ : [List] [List] -> [List] [assoc id: nil] .
cmb X Y : List if X : List /\ Y : List .
cmb I P I : NePal if I : Qid /\ P : Pal .
```

where `[List]` denotes the kind corresponding to the connected component of sorts `[List]` = {`Qid`, `NeList`, `NePal`, `Pal`, `NeList`}. Since we understand `[List]` as an equivalence class of sorts, the choice of a representative sort is immaterial, e.g., `[List]`=`[Pal]`. Maude 2.0 also allows on-the-fly declaration of variables inside expressions, without any need for a separate `var` declaration; for example, we can introduce in this way variables such as `X:Pal` and `Y:[List]` in expressions such as `rev(X:Pal)` and `tail(Y:[List])`.

Terms that *have a kind* but *do not have a sort* in S are thought of as *error*, or *undefined*, terms. For example, `head(nil)` is one such term. Membership equational logic gives us a general way of dealing with *partiality* within the total context provided by the kinds.

For (Σ, E) a membership equational theory, the *inference rules* to deduce other equations and memberships 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 $f : k_1 \ldots k_n \to k$ is in Σ, and the terms $t_i, t'_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$.

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
$$\begin{array}{rcl} (\forall X)\ t = t' & \Leftarrow & u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m \\ (\text{resp. } (\forall X)\ t : s & \Leftarrow & u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m) \end{array}$$
in the set E of axioms, and given a substitution[2] $\theta : X \to T_\Sigma(Y)$, then
$$\frac{E \vdash (\forall Y)\ \overline{\theta}(u_i) = \overline{\theta}(v_i)\ \ 1 \leq i \leq n \qquad E \vdash (\forall Y)\ \overline{\theta}(w_j) : s_j\ \ 1 \leq j \leq m}{E \vdash (\forall Y)\ \overline{\theta}(t) = \overline{\theta}(t') \quad (\text{resp. } (\forall Y)\ \overline{\theta}(t) : s)}$$

A model of a membership equational signature $\Sigma = (K, \Sigma, S)$ is called a Σ-*algebra*. A Σ-algebra A consists of a K-indexed family of sets $A = \{A_k\}_{k \in K}$, together with:

- for each $f : k_1 \ldots k_n \longrightarrow k$ in Σ a function $f_A : A_{k_1} \times \ldots \times A_{k_n} \longrightarrow A_k$; and
- for each $k \in K$ and each $s \in S_k$ a subset $A_s \subseteq A_k$.

Given two Σ-algebras A and B, a Σ-*homomorphism* is a K-indexed family of functions $h = \{h_k : A_k \longrightarrow B_k\}_{k \in K}$, denoted $h : A \longrightarrow B$, preserving the operations Σ and the subsorts, that is, such that:

- for each $f : k_1 \ldots k_n \longrightarrow k$ in Σ and each $(a_1, \ldots a_n) \in A_{k_1} \times \ldots \times A_{k_n}$ we have, $h_k(f_A(a_1, \ldots a_n)) = f_B(h_{k_1}(a_1), \ldots h_{k_n}(a_n))$; and
- for each $k \in K$ and each $s \in S_k$, $h_k(A_s) \subseteq B_s$.

The *models* of a membership equational theory (Σ, E) are those Σ-algebras that satisfy the axioms E. The above inference rules are *sound and complete* [35]. That is, an equation $(\forall X)\ t = t'$ or a membership $(\forall X)\ t : s$ are *provable* from (Σ, E) according to the above rules iff they are satisfied by all (Σ, E) models. Furthermore, any membership equational theory (Σ, E) has an *initial algebra*[3] $T_{\Sigma/E}$ [35] which, assuming Σ unambiguous, is defined as a quotient of the term algebra T_Σ by:

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

The paper [2] extends in a natural way to membership equational logic theories the usual results about *equational simplification*, *confluence*, *termination*, and *sort-decreasingness*[4] well-known for order-sorted theories. Under such assumptions, a membership equational theory can be used as a *declarative functional program* which is *deterministic* thanks to the confluence property and can be executed by equational simplification using the equations from left to right. Maude supports efficient functional evaluation by simplification as we have illustrated with the `LIST` and `PALINDROME` modules.

[2]By a *substitution* we mean a K-indexed family of functions $\theta : X \to T_\Sigma(Y)$ assigning to each variable x, say of kind k, in X a term $\theta(x) \in T_\Sigma(Y)_k$

[3]By definition, $T_{\Sigma/E}$ is *initial* among all (Σ, E) models iff for each (Σ, E) model A there exists a unique Σ-homomorphism $h : T_{\Sigma/E} \longrightarrow A$.

[4]Intuitively —assuming confluence, termination, and a syntactic condition on Σ called (pre-)regularity [20, 2]— if the axioms E are sort-decreasing, then, the canonical form $can_E(t)$ of a term t contains the most precise sort information about t, in the sense that we can compute the *smallest* sort s possible for t by repeated application to $can_E(t)$ of a more specialized version of the above inference rules in which steps of equality are only simplification steps taken from left to right. For a more precise account see [2].

2.3 Verification of Functional Module Properties

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, sort-decreasing, 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* $Can_{\Sigma,E/A}$, whose elements are A-equivalence classes of terms fully simplified by the equations E. Since any term t can be simplified by E to an A-equivalence class $can_{E/A}(t)$ that cannot be simplified any further, we have the Σ-isomorphism (see, for example, Lecture 5 in [37]):

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

Different *properties* of the module $\texttt{fmod}(\Sigma, E \cup A)\texttt{endfm}$ can be expressed a *sentences* φ, perhaps in equational logic, or, more generally, in the *first-order logic language* of a signature containing Σ. When does the above module *satisfy* property φ? When we have,

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

How do we *verify* such properties? Let us look at an example. Consider the module,

```
fmod PEANO-NAT is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  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,

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

We need more than just equational deduction to verify this property. Indeed, associativity is *not* a property satisfied by *all models* of the equations E in PEANO-NAT. Therefore, by the soundness and completeness of membership equational logic [35], it cannot be deduced equationally from the PEANO-NAT equations. Consider, for example, the initial model obtained by adding a *nonstandard number* a,

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

Since it has the same equations E as PEANO-NAT, this initial model obviously satisfies the equations E, but it does not satisfy associativity, since we have,

$$T_{\Sigma_{\texttt{NON-STANDARD-NAT}}/E} \models \texttt{a +(a + a)} \neq \texttt{(a + a)+ a}.$$

This fact is easy to see using the isomorphism,

$$T_{\Sigma_{\texttt{NON-STANDARD-NAT}}/E} \;\cong\; Can_{\Sigma_{\texttt{NON-STANDARD-NAT}}/E},$$

since the equations E are confluent and terminating, and *no equations in E can simplify either side* of the inequality. Therefore we have, $can_E(\texttt{a +(a + a)}) = \texttt{a +(a + a)}$, and $can_E(\texttt{(a + a)+ a}) = \texttt{(a + a)+ a}$.

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*,

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

Furthermore, $\vdash_{ind}$ should prove all that equational deduction can prove and more. That is, for formulas φ that are equations it should satisfy,

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

Because of Gödel's *Incompleteness Theorem*, in general 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 in these lectures generalizes to general membership equational logic specifications the usual *proofs by natural number induction*. In fact, in our example of associativity of natural number addition structural induction actually *specializes* to the usual proof method by natural number induction.

2.4 Machine-Assisted Proof with Maude's ITP

Maude's ITP [7] is an *inductive theorem prover* supporting proof by induction in Maude modules. It is a program written entirely in Maude by Manuel Clavel that extends the Full Maude module algebra [6] (another program written entirely in Maude by Francisco Durán) and works as follows. After both Full Maude and the ITP have been loaded into Maude one:

- enters a module (enclosed in parentheses) that is loaded into the Full Maude database;
- enters a goal to be proved, also enclosed in parentheses and beginning with the name of the module where it has to be proved; and
- gives commands, corresponding to proof steps, to prove that property.

For example, after we enter the module `PEANO-NAT` into Full Maude enclosed in parentheses, we can present to the ITP the associativity of addition goal by giving this goal a name (`main`) and stating the equation to be proved as follows,

```
Maude> (goal main : PEANO-NAT |- {N:Natural ; M:Natural ;  L:Natural}
       ((N:Natural + (M:Natural + L:Natural)) =
                             ((N:Natural + M:Natural) + L:Natural)) .)
```

where each variable is qualified by its sort, and `{N:Natural ; M:Natural ; L:Natural}` denotes universal quantification on the listed variables. The ITP then responds by labeling that goal `main.0` and prompting us to give more proof commands.

```
=================================
main.0
=================================


|-{N:Natural ; M:Natural ; L:Natural}(N:Natural +(M:Natural + L:Natural)=
                                  (N:Natural + M:Natural)+ L:Natural)

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

Maude>
```

The user can then instruct the ITP to do induction on the variable `L:Natural` by giving the command `(ind (main.0) on L:Natural .)`, which then splits that goal into subgoals `main.1.0` (base case) and `main.2.0` (induction step).

```
Maude> (ind (main.0) on L:Natural .)

=================================
main.1.0
=================================


|-{N:Natural ; M:Natural}(N:Natural +(M:Natural + 0)=
                          (N:Natural + M:Natural)+ 0)
=================================
main.2.0
=================================


|-{V#0:Natural}({N:Natural ; M:Natural}
    (N:Natural +(M:Natural + V#0:Natural)=
                (N:Natural + M:Natural)+ V#0:Natural)==>
    {N:Natural ; M:Natural}(N:Natural +(M:Natural + s(V#0:Natural))=
                              (N:Natural + M:Natural)+ s(V#0:Natural)))

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

Maude>
```

We can then try to *simplify* subgoal `main.1.0` *automatically*, by using the ITP's `auto` tactic[5] as follows:

```
Maude> (auto (main.1.0) .)

=================================
main.2.0
=================================


|-{V#0:Natural}
   ({N:Natural ; M:Natural}(N:Natural +(M:Natural + V#0:Natural)=
                           (N:Natural + M:Natural)+ V#0:Natural)==>
    {N:Natural ; M:Natural}(N:Natural +(M:Natural + s(V#0:Natural))=
                             (N:Natural + M:Natural)+ s(V#0:Natural)))

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

Maude>
```

Similarly, we can now try to simplify automatically the remaining `main.2.0` subgoal, which succeeds and finishes the proof of our original goal.

```
Maude> (auto (main.2.0) .)

q.e.d
```

So far we have only used *natural number induction*. More generally, the ITP can perform proofs by *structural induction* on any membership equational logic specification. Let us illustrate this general capability by means of our *palindrome* example. Palindromes are words that read exactly the same way from left to right and from right to left. That is, the equation `rev(P:Pal)= P:Pal` holds inductively. Can we prove it? Actually, this goal cannot be proved without extra assumptions. We need to prove first an *auxiliary lemma* in the `LIST` submodule, namely that for `L1` and `L2` lists we have, `rev(L1 L2) = rev(L2) rev(L1)`. This can done by entering `LIST` enclosed in parentheses and then naming that lemma with a label (say `rev`) and entering it as a goal as follows:

[5]This tactic takes automatically the obvious steps, such as applying equations to try to simplify both sides of the equality, and, in the case of a universally quantified implication goal, transforming the quantified variables into constants, and adding the equations in the transformed hypothesis to the module to try to prove the transformed consequence.

```
Maude> (goal rev : LIST |- {L1:List ; L2:List}
       (rev(L1:List L2:List) = (rev(L2:List) rev(L1:List))) .)

=================================
rev.0
=================================


|-{L1:List ; L2:List}(rev(L1:List L2:List)= rev(L2:List)rev(L1:List))

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

Maude>
```

We can prove this lemma by inducting on the `L1:List` variable, which splits `rev.0` into five subgoals:

```
Maude> (ind (rev.0) on L1:List .)

=================================
rev.1.0
=================================


|-{L2:List}(rev(nil L2:List)= rev(L2:List)rev(nil))
=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                         rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                             rev(L2:List)rev(V#0:NeList V#0:List)))
=================================
rev.4.0
=================================


|-{V#0:List ; V#1:List}
  ({L2:List}(rev(V#1:List L2:List)= rev(L2:List)rev(V#1:List))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#1:List)L2:List)=
                           rev(L2:List)rev(V#0:List V#1:List)))
=================================
rev.5.0
=================================


|-{L2:List ; V#0:Qid}(rev(V#0:Qid L2:List)= rev(L2:List)rev(V#0:Qid))

+++++++++++++++++++++++++++++++++
```

```
Maude>
```

We can simplify automatically the `rev.1.0` and `rev.5.0` subgoals:

```
Maude> (auto (rev.1.0) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                          rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                              rev(L2:List)rev(V#0:NeList V#0:List)))
=================================
rev.4.0
=================================


|-{V#0:List ; V#1:List}
  ({L2:List}(rev(V#1:List L2:List)= rev(L2:List)rev(V#1:List))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#1:List)L2:List)=
                            rev(L2:List)rev(V#0:List V#1:List)))
=================================
rev.5.0
=================================


|-{L2:List ; V#0:Qid}(rev(V#0:Qid L2:List)= rev(L2:List)rev(V#0:Qid))

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

Maude> (auto (rev.5.0) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}({L2:List}(rev(V#0:NeList L2:List)=
                              rev(L2:List)rev(V#0:NeList))&
    {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
    {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                              rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}({L2:List}(rev(V#0:NeList L2:List)=
```

```
                              rev(L2:List)rev(V#0:NeList))&
    {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
    {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                              rev(L2:List)rev(V#0:NeList V#0:List)))
=================================
rev.4.0
=================================


|-{V#0:List ; V#1:List}({L2:List}(rev(V#1:List L2:List)=
                            rev(L2:List)rev(V#1:List))&
    {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
    {L2:List}(rev((V#0:List V#1:List)L2:List)=
                            rev(L2:List)rev(V#0:List V#1:List)))

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

Maude>
```

Note that the three remaining subgoals `rev.2.0`, `rev.3.0`, and `rev.4.0`, are all "induction hypothesis case" first-order formulae with a very similar structure involving several levels of universal quantification, conjunction, and implication. We need to "unpack" these formulas to get to the underlying equations in the hypothesis and the conclusion. To get rid of the outermost quantification layer in `rev.4.0` we apply the "lemma of constants" to turn the corresponding variables into costants as follows:

```
Maude> (cns (rev.4.0) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                          rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                              rev(L2:List)rev(V#0:NeList V#0:List)))
=================================
rev.4.0
=================================


|-{L2:List}(rev(V#1*List L2:List)= rev(L2:List)rev(V#1*List))&
  {L2:List}(rev(V#0*List L2:List)= rev(L2:List)rev(V#0*List))==>
  {L2:List}(rev((V#0*List V#1*List)L2:List)=
                             rev(L2:List)rev(V#0*List V#1*List))

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

Maude>
```

Note that now in `rev.4.0` the variables formerly quantifying the outermost level have been replaced by constants[6] of the corresponding sorts. For example, `V#0:List` has been turned into the constant `V#0*List`. We can now perform a step of "implication elimination" on `rev.4.0` by giving the command,

```
Maude> (imp (rev.4.0) .

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                          rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                              rev(L2:List)rev(V#0:NeList V#0:List)))
=================================
rev.4.0
=================================


|-{L2:List}(rev((V#0*List V#1*List)L2:List)= rev(L2:List)rev(V#0*List V#1*List))

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

Maude>
```

which has the effect of adding the hypotheses of the implication as additional axioms to the current module for `rev.4.0` and leaving the conclusion of the implication as the remaining subgoal `rev.4.0`. We can now eliminate the remaining universal quantification in `rev.4.0` by applying again the "lemma of constants":

```
Maude> (cns (rev.4.0) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                          rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================
```

[6]Each goal to be proved has an associated *module*, which is not necessarily the original one (`LIST` in this example). The associated module can be *modified* by a `cns` command, that adds to it additional *constants* (as in the last step) or (as in the next step) by an `imp` command, that adds the equations or memberships in an implication's condition as new hypotheses. These modules reside in the Full Maude database and can be interrogated by the user to inspect their axioms at each stage if necessary.

```
|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                             rev(L2:List)rev(V#0:NeList V#0:List)))
================================
rev.4.0
================================


|- rev((V#0*List V#1*List)L2*List)= rev(L2*List)rev(V#0*List V#1*List)

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

Maude>
```

One of the consequences of adding new equations to a module is that the resulting module—which was originally (ground) confluent, sort-decreasing, and terminating—may easily fail to remain so. In such cases, as now for `rev.4.0`, a simple goal such as the above equation—that might be easily simplified automatically by simplifying both sides of the equation if the module had those good executability properties—has to be solved by *applying the equations carefully in a user-directed way.* For this, the first thing we do is to *declare all equations in the module associated to* `rev.4.0` as *non-executable* by giving the command `(set*-in (rev.4.0) .)`. This command does not alter the goal, but has the effect of adding to each of the axioms in the module associated to `rev.4.0` the `nonexec` attribute, so that they are no longer used automatically as simplification rules but have to be applied under user guidance by giving an `apply` command specifying: (1) which axioms to use and in which direction (by giving the correspoding label, prefixed by '`-`' if from right to left), (2) where to apply them (whether to the left or to the right term of the equation on the goal, and at what subtem position), and (3) with which substitution. For example, we can apply the induction hypothesis (which has been labeled internally by `hyp`) from left to right in three different parts of the equation in the goal: first at the top (position `0`) of the lefthand side; then at the first immediate subterm (position `1.0`) of the resulting lefthand side; and finally at the second immediate subterm (position `2.0`) of the righthand side (note the `apply-r` variant for the righthand side):

```
Maude> (apply hyp to (rev.4.0) at (0)
              with ((L2:List <- (V#1*List L2*List))) .)

================================
rev.2.0
================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                         rev(L2:List)rev(V#0:List V#0:NeList)))
================================
rev.3.0
================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                             rev(L2:List)rev(V#0:NeList V#0:List)))
================================
rev.4.0
```

```
=================================


|- rev(V#1*List L2*List)rev(V#0*List)= rev(L2*List)rev(V#0*List V#1*List)

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

Maude> (apply hyp to (rev.4.0) at (1.0)
              with ((L2:List <- (L2*List))) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                          rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                              rev(L2:List)rev(V#0:NeList V#0:List)))
=================================
rev.4.0
=================================


|- rev(L2*List)rev(V#1*List)rev(V#0*List)= rev(L2*List)rev(V#0*List V#1*List)

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

Maude> (apply-r hyp to (rev.4.0) at (2.0)
                with ((L2:List <- (V#1*List))) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                          rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                              rev(L2:List)rev(V#0:NeList V#0:List)))
```

```
=================================
rev.4.0
=================================


|- rev(L2*List)rev(V#1*List)rev(V#0*List)= rev(L2*List)rev(V#1*List)rev(
    V#0*List)

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

Maude>
```

Note that now `rev.4.0` has become an *identity*. We can then *perform an identity inference step* to eliminate the `rev.4.0` subgoal.

```
Maude> (idt (rev.4.0) .)

=================================
rev.2.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:List V#0:NeList)L2:List)=
                         rev(L2:List)rev(V#0:List V#0:NeList)))
=================================
rev.3.0
=================================


|-{V#0:List ; V#0:NeList}
  ({L2:List}(rev(V#0:NeList L2:List)= rev(L2:List)rev(V#0:NeList))&
  {L2:List}(rev(V#0:List L2:List)= rev(L2:List)rev(V#0:List))==>
  {L2:List}(rev((V#0:NeList V#0:List)L2:List)=
                             rev(L2:List)rev(V#0:NeList V#0:List)))

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

Maude>
```

The proofs of subgoals `rev.2.0` and `rev.3.0` are entirely similar and are omitted. After proving those two subgoals, and therefore the lemma `rev`, we can enter the `PALINDROME` module in parentheses and state our original goal:

```
Maude> (goal main : PALINDROME |- {P:Pal}(rev(P:Pal) = P:Pal) .)

=================================
main.0
=================================


|-{P:Pal}(rev(P:Pal)= P:Pal)

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

Maude>
```

We can now induct on the `P:Pal` variable, which has the effect of splitting `main.0` into the following subgoals:

```
Maude> (ind (main.0) on P:Pal .)

=================================
main.1.0
=================================


|- rev(nil)= nil
=================================
main.2.0
=================================


|-{I:Qid ; I:Qid ; P:Pal}
  (rev(P:Pal)= P:Pal & rev(I:Qid)= I:Qid &
   rev(I:Qid)= I:Qid ==>
   rev(I:Qid P:Pal I:Qid)= I:Qid P:Pal I:Qid)
=================================
main.3.0
=================================


|-{V#0:Qid}(rev(V#0:Qid)= V#0:Qid)

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

Maude>
```

We can simplify automatically `main.1.0` and `main.3.0` by giving the commands `(auto (main.1.0) .)` and `(auto (main.3.0) .)`. After this we only have left the `main.2.0` subgoal.

```
=================================
main.2.0
=================================


|-{I:Qid ; I:Qid ; P:Pal}
  (rev(P:Pal)= P:Pal & rev(I:Qid)= I:Qid &
   rev(I:Qid)= I:Qid ==>
   rev(I:Qid P:Pal I:Qid)= I:Qid P:Pal I:Qid)

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

Maude>
```

After applying the "lemma of constants" and "implication elimination" with the commands `(cns (main.2.0) .)` and `(imp (main.2.0) .)` we get to the transformed subgoal:

```
=================================
main.2.0
=================================


|- rev(I*Qid P*Pal I*Qid)= I*Qid P*Pal I*Qid

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

Maude>
```

In trying to apply the `rev` lemma to this goal we again have a problem of nonexecutablity, which we solve by temporarily making the axioms of the module associated to `main.2.0` nonexecutable by giving the command `(set*-in (main.2.0) .)`. We can then apply the `rev` lemma as follows:

```
Maude> (apply rev to (main.2.0) at (0)
        with ((L2:List <- (I*Qid))) .)

=================================
main.2.0
=================================


|- rev(I*Qid)rev(I*Qid P*Pal)= I*Qid P*Pal I*Qid

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

Maude>
```

We can make the axioms in the module associated with `main.2.0` executable again by giving the command `(set-in (main.2.0) .)`. However, we still want to keep the lemma `rev` nonexecutable, which we achieve by giving the additional command `(set* rev in (main.2.0) .)`. Then, we can simplify both sides of the equation with the "rewrite command" `(rwr (main.2.0) .)` to get the identity:

```
Maude> (rwr (main.2.0) .)

=================================
main.2.0
=================================


|- I*Qid P*Pal I*Qid = I*Qid P*Pal I*Qid

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

Maude>
```

We can then prove our goal by giving the identity inference command:

```
Maude> (idt (main.2.0) .)

q.e.d
```

As illustrated by the above proofs of properties for `PEANO-NAT`, `LIST`, and `PALINDROME`, Maude's ITP supports a very general scheme of structural induction which depends on the form of the data *constructors* involved in each case: zero and successor for `PEANO-NAT`, and nil and concatenation for `LIST` and `PALINDROME`. More precisely, the structural induction associated to a membership equational logic specification is based on the *memberships* defining the data of each sort. Such memberships are of three kinds:

- operator declarations having the `ctor` attribute, which are syntactic sugar for a corresponding membership;
- subsort declarations, which are also memberships in disguise; and
- memberships defined explicitly as such in the Maude module.

These different memberships account for the different cases in which a goal is split when inducting on a variable of that sort. The general structural induction scheme thus associated to memberships and its soundness are described in detail in Section 4.1 of [1]. But what about declarations of operators that are *not* constructors (for example, natural number addition)? why can they be excluded from the induction scheme? The answer is that their exclusion is in fact an *implicit proof obligation* about the so-called *sufficient completeness* (which could be better described as *sufficient generation*) of the equations to simplify every ground term to a constructor term. In fact, the memberships associated to nonconstructor operator declarations are *inductive consequences* of the specification which can be also discharged by the ITP, by proving inductive goals about a related specification [7] (see also Lecture 9 in [37]).

Besides the ITP tool, the following Maude tools, written in Maude by Francisco Durán, can be used to prove certain properties of equational specifications:

- *Church-Rosser Checker* [13]: checks confluence assuming termination;
- *Termination Checker* [11]: checks termination of equations under different orders;
- *Knuth-Bendix Completion Tool* [11]: tries to complete an equational theory to make it confluent.

3 Declarative Concurrent Programs and Rewriting Logic

In declarative concurrent programming, the first question to ask is: what is a *suitable logic* to write concurrent programs in a declarative style? This is of course an *open-ended* question, in that a variety of answers are possible at present, and new answers may be proposed in the future. In these lectures, we will use *rewriting logic* [33] as a specific logic that is indeed suitable for concurrent declarative programming. This is in full harmony with our use of equational logic for deterministic declarative programming. In fact, rewriting logic *generalizes* equational logic in a natural way and makes possible a seamless extension from deterministic to possibly nondeterministic and concurrent declarative programming.

We first give a quite general definition of rewrite theories. We will further generalize this notion in Section 3.5. A *rewrite theory* $\mathcal{R}$ is a triple $\mathcal{R} = (\Sigma, E, R)$, with:

- (Σ, E) a membership equational theory, and
- R a set of *labeled conditional rewrite rules* of the form,

$$r : t \longrightarrow t' \ \Leftarrow\ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l'),$$

where r is a label, and there is a finite set X of variables such that $t, t' \in T_\Sigma(X)_k$ for some kind k, all the Σ-terms $u_i, u_i', v_j, w_k, w_l'$ in the condition are in $T_\Sigma(X)$, and the condition is a conjunction of *equations*, *memberships*, and *rewrites* (where w_l, w_l' must also have the same kind). There is *no* requirement that[7] $vars(t) = X$, and *no* assumptions of confluence or termination. The rule is called *unconditional* if the condition is empty.

Intuitively, the operational meaning of such a rule is that, if for a given substitution θ the corresponding substitution instances of the equations and memberships in the condition are provable from E, and, furthermore, it is possible using the rules in R to perform, possibly complex, rewrites $\theta(w_k) \longrightarrow \theta(w_k')$ for each k, then, it is possible to rewrite $\theta(t)$ to $\theta(t')$ in one step.

3.1 Maude System Modules

In Maude, rewrite theories are specified in *system modules*. In the same way that a functional module has essentially the form, `fmod` (Σ, E) `endfm`, with (Σ, E) a membership equational logic theory, a system module has essentially the form, `mod` (Σ, E, R) `endm`, with (Σ, E, R) a rewrite theory.

A conditional rewrite rule of the form, $l : t \longrightarrow t' \ \Leftarrow cond$ is specified in Maude with syntax,

`crl [`l`] : ` t `=>` t' `if` *cond* `.`

Similarly, an unconditional rewrite rule of the form, $l : t \longrightarrow t'$ is specified in Maude with syntax,

`rl [`l`] : ` t `=>` t' `.`

To motivate both rewriting logic as a formalism to specify and program concurrent systems, and Maude system modules as a concrete realization of this formalism, we will show how three important classes of systems, namely:

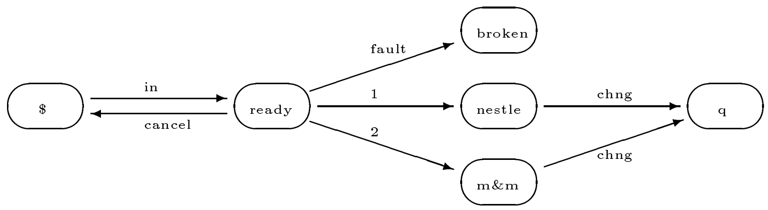
- automata, also called *labeled transition systems*,
- *Petri nets*, one of the simplest concurrency models, and
- *object-oriented* concurrent systems,

can be naturally expressed as rewrite theories and can be specified in Maude as system modules.

[7] We denote by $vars(t)$ the set of variables occurring in the term t.

3.2 Labeled Transition Systems

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:



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.

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 that our automaton, although it exhibits a form of nondeterminism, *has no concurrency* whatsoever.

We can specify such an automaton as the following system module:

```
mod CANDY-AUTOMATON is
  sort State .
  ops $ ready broken nestle m&m q : -> State [ctor] .
  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
```

Note that *rewrite rules* do *not* have an equational interpretation. They are not understood as equations, but as *transitions*, that in general *cannot be reversed*. This is why in a rewrite theory (Σ, E, R) the equations in E are *totally different* from the rules in R, since equations and rules have *totally different semantics*.

Operationally, the Maude system assumes that the equations in E are confluent, terminating, and sort decreasing modulo some axioms $A \subseteq E$, and will compute with such equations and also with the rules in R by rewriting, yet distinguishing *equation simplification* (the `reduce` command) from *rewriting with rules* (the `rewrite` command). Indeed, Maude can execute rewrite theories with the `rewrite` command (which 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 rules in a *top down* and *rule fair*[8] way (all rules are given a chance) until termination, and gives one result. In the above example, fairness saves us from nontermination, but in general we can easily have nonterminating computations.

For this reason the `rewrite` command can be given a numeric argument stating the *maximum number of rewrite steps*. For example (setting also the `trace` facility on),

[8] Maude 2.0 has also an `frewrite` command that is both *rule and position fair*, that is, all rules and all positions where a rule applies are eventually given a chance to rewrite.

```
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
```

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 Maude's `search` command, which takes two terms: a ground term, which is our initial state, and a term, possibly with variables, which specifies our desired target state. This second term should be a *pattern*, that is, a term involving only constructors, so that it becomes decidable by matching whether a given state simplified to canonical form with the equations E is an instance of it.

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),

```
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,

```
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
```

We can ask for the shortest path to any state in the state graph (for example, to 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
```

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,

```
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
```

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
```

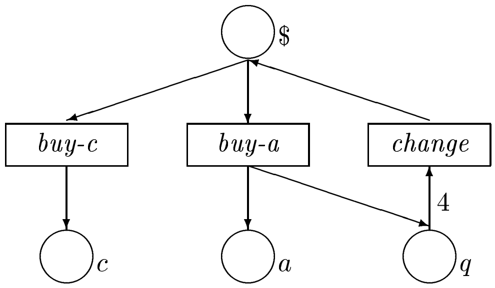
Our `CANDY-AUTOMATON` example is just a special instance of a general concept, namely, that of *automaton*, also called a *labeled transition system* (LTS), by which we mean a triple $A = (A, L, T)$ with:

- A a set, called the set of *states*,
- L a set, called the set of *labels*, and
- $T \subseteq A \times L \times A$ a subset, called the set of *labeled transitions.*

We have associated to our candy automaton the rewrite theory `CANDY-AUTOMATON`. This is an instance of *a general transformation*, that assign to each LTS A a rewrite theory $R(A) = (\Sigma_A, \emptyset, R_A)$, whose signature Σ_A has a single sort and has as constants the elements $x \in A$, whose set of equations is empty, and with R_A having a rewrite rule $l : x \longrightarrow y$ for each $(x, l, y) \in T$.

3.3 Petri Nets

So far so good, but we have not yet seen any concurrency. The simplest concurrent system examples are probably the *concurrent automata* called *Petri nets* after their proponent, Carl Adam Petri [41]. Consider for example the picture,



which 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, called *places*. 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*,

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

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

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

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 *rewrite rule* from one (pre-)marking to another (post-)marking. 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 simultaneously pushing the `buy-c` button (twice and concurrently!) and the `buy-a` button.

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
```

We can also interrogate the search graph,

```
Maude> show search graph .
state 0, Marking: $ q q q
arc 0 ===> state 1 (rl [buy-c]: $ => c .)
arc 1 ===> state 2 (rl [buy-a]: $ => a q .)

state 1, Marking: c q q q

state 2, Marking: a q q q q
```

```
arc 0 ===> state 3 (rl [chng]: q q q q => $ .)

state 3, Marking: $ a
arc 0 ===> state 4 (rl [buy-c]: $ => c .)
arc 1 ===> state 5 (rl [buy-a]: $ => a q .)

state 4, Marking: c a

state 5, Marking: a a q

Maude> show path 4 .
state 0, Marking: $ q q q
===[ rl [buy-a]: $ => a q . ]===>
state 2, Marking: a q q q q
===[ rl [chng]: q q q q => $ . ]===>
state 3, Marking: $ a
===[ rl [buy-c]: $ => c . ]===>
state 4, Marking: c a
```

Why was concurrency *impossible* in our `CANDY-AUTOMATON` example but becomes possible in our `PETRI-MACHINE` example? The problem with `CANDY-AUTOMATON`, as with any LTS having unstructured states, is that its states are *atomic*, and, having no smaller pieces, *cannot be distributed*. By contrast, a Petri net marking *is made out of smaller pieces*, namely its constituent places, and therefore *can be distributed*, so that several transitions can happen simultaneously.

Then what, is concurrency about multisets? Not necessarily. This is the very common, and cheap, fallacy of *taking the part for the whole*; for example, "Logic Programming = Prolog," or "Concurrency = Petri Nets". A more fair and open-minded answer is suggested by the rewriting logic motto (see [33]):

Concurrent Structure = *Algebraic Structure*.

That is, *any algebraic structure* in the set of states, other than atomic constants, even a single unary operator, will make the states potentially *distributed*, and will therefore allow concurrent transitions. Of course, the potential for concurrency may be frustrated by the specific transitions of a system *forcing a sequential execution*, but the potential is there if we use other transitions.

In summary, there are *as many possible styles of concurrent systems* as there are *signatures* Σ and equations E. For example: multiset concurrency, tree concurrency, graph concurrency, string concurrency, and many, many other possibilities and hybrid combinations.

In general, a *place-transition* Petri net N consists of:

- a set P of *places*; we then call *markings* to the elements in the free commutative monoid $M(P)$ of finite multisets of P;
- a labeled transition system $N = (M(P), L, T)$.

The general transformation associating a rewrite theory $R(N)$ to each Petri net N is then obvious. $R(N) = (\Sigma_N, E, R_N)$ has:

- Σ_N with a single sort, which we may call *Marking*, with constants the elements of P and a *null* constant, and with a binary operator $\_\,\_ : \mathit{Marking}\ \mathit{Marking} \longrightarrow \mathit{Marking}$,
- E the associativity and commutativity axioms for this binary operator, plus the identity axioms for *null*, which are specified in Maude with the keywords `[assoc comm id: null]`,
- R_N has a rewrite rule $l : m \longrightarrow m'$ for each $(m, l, m') \in T$.

3.4 Concurrent Objects in Rewriting Logic

Rewriting logic can naturally model 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 [33, 36], including, for example, CCS [44], the π-calculus [42], real-time models [40], 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.*

It is of course possible to *represent* a concurrent object system as a rewrite theory with different modeling styles and adopting different *notational conventions.* What follows is based on a particular style of representation [34] that has proved useful and expressive in practice, and that is supported by Full Maude's *object-oriented modules* [9, 6].

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.*

3.4.1 Configurations

We first explain (1): how the distributed states are equationally axiomatized. 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,

$$\_\ \_ : \mathit{Configuration}\ \ \mathit{Configuration} \longrightarrow \mathit{Configuration}.$$

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

$$\mathit{Object}\ \mathit{Message} < \mathit{Configuration}$$

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

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 the name of 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. The value of each attribute shouldn't be arbitrary: it should have an appropriate *sort*, dictated by the nature of the attribute. Therefore, *object classes* in Full Maude 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`. 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. 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.

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 .
```

3.4.2 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,

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

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.

That is, a number of objects and messages can come together and participate in a transition in which the messages are consumed, some new objects may be created, others may be destroyed, and others may 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*.

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`. 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 $\mathcal{R} = (\Sigma, E, R)$ 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).

3.5 Rewrite Theories in General

It is clear that *rewriting logic has the underlying equational logic as a parameter:* the more general the equational logic, the more general and expressive the resulting rewrite theories are. In particular, choosing membership equational logic as the underlying equational logic, as done in Maude, leads to highly expressive specifications. For example, we have seen that, when membership equational logic is the underlying equational logic, conditional rewrite rules can have the very expressive general form,

$$r : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l').$$

It has also become increasingly clear that *frozen arguments in operators*, that restrict the rewrites allowed below such arguments, are also very useful in practice. By the above simultaneous generalization of the underlying equational logic, the just mentioned general form of the conditional rewrite rules, and the presence of frozen arguments, we arrive at very expressive notions of: (1) rewrite theory, (2) rewriting logic's rules of deduction, and (3) the models of a rewrite theory. This generalization has been developed in [3], and is explained in what follows.

We can illustrate frozen operator arguments with the following nondeterministic choice example:

```
mod CHOICE is
  protecting INT .
  sorts Elt MSet .
  subsorts Elt < MSet .
  ops a b c d e f g : -> Elt [ctor] .
  op __ : MSet MSet -> MSet [ctor 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
```

It does not make much sense to rewrite below the cardinality function `card`, because then the multiset whose cardinality 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`'s single argument as `frozen`, we restrict rewrites to the state kind, where they belong.

This leads to the following general definition of a rewrite theory. A *rewrite theory* is a 4-tuple, $\mathcal{R} = (\Sigma, E, \phi, R)$, where:

- (Σ, E) is a membership equational theory, with, say, kinds K, sorts S, and operations Σ;
- $\phi : \Sigma \longrightarrow \mathcal{P}_{fin}(\mathbb{N})$ is a $K^* \times K$-indexed family of functions assigning to each $f : k_1 \ldots k_n \longrightarrow k$ in Σ the finite set $\phi(f) \subseteq \{1, \ldots, n\}$ of its *frozen argument positions*;
- R is a set of (universally quantified) labeled conditional rewrite rules of the form (with t, t' and the w_l, w_l' pairs of terms of same kind)

$$r : t \longrightarrow t' \;\Leftarrow\; (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l').$$

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, and given a Σ-term $t \in T_\Sigma(X)$, we call a variable $x \in vars(t)$ *frozen* in t iff there is a nonvariable position $\alpha \in \mathbb{N}^*$ such that $t/\alpha = f(u_1, \ldots, u_i, \ldots, u_n)$, with $i \in \phi(f)$, and $x \in vars(u_i)$. Otherwise, we call $x \in X$ *unfrozen.* Similarly, given Σ-terms $t, t' \in T_\Sigma(X)$, we call a variable $x \in X$ *unfrozen* in t and t' iff it is unfrozen in both t and t'.

3.6 Rewriting Logic in General

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, the sentences that it proves are universally quantified rewrites 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**. $\dfrac{(\forall X)\; u \longrightarrow v \quad E \vdash (\forall X) u = u' \quad E \vdash (\forall X) v = v'}{(\forall X)\; u' \longrightarrow v'}$

- **Congruence**. For each $f : k_1 \dots k_n \longrightarrow k$ in Σ, with $\{1, \dots, n\} - \phi(f) = \{j_1, \dots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,

$$\frac{(\forall X)\; t_{j_1} \longrightarrow t'_{j_1} \quad \dots \quad (\forall X)\; t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\; f(t_1, \dots, t_{j_1}, \dots, t_{j_m}, \dots, t_n) \longrightarrow f(t_1, \dots, t'_{j_1}, \dots, t'_{j_m}, \dots, t_n)}$$

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

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

with $Z = \{x_{j_1}, \dots, x_{j_m}\}$ the set of unfrozen variables in t and t', then,

$$\frac{\begin{array}{c}(\bigwedge_r (\forall Y)\; p_{j_r} \longrightarrow p'_{j_r}) \\ (\bigwedge_i (\forall Y)\; \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\; \theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\; \theta(w_k) \longrightarrow \theta(w'_k))\end{array}}{(\forall Y)\; \theta(t) \longrightarrow \theta'(t')}$$

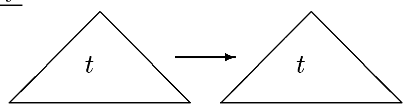
where for $x \in X - Z$, $\theta'(x) = \theta(x)$, and for $x_{j_r} \in Z$, $\theta'(x_{j_r}) = p'_{j_r}$, $1 \leq r \leq m$.

- **Transitivity**

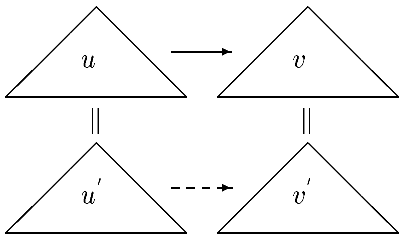
$$\frac{(\forall X)\; t_1 \longrightarrow t_2 \qquad (\forall X)\; t_2 \longrightarrow t_3}{(\forall X)\; t_1 \longrightarrow t_3}$$

We can visualize the above inference rules as follows:

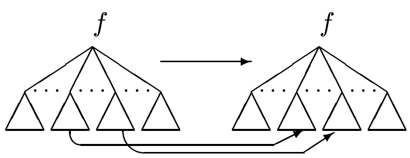
Reflexivity



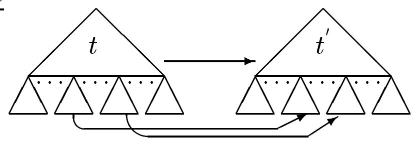
Equality



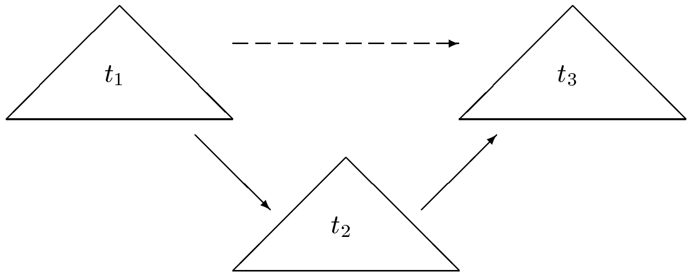
Congruence



Replacement



Transitivity



3.7 Computational and Logical Readings

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has two closely related, yet different, *readings*, one computational and another logical. *Computationally*, a rewrite theory specifies a *concurrent system* whose set of *states* is (a kind in) the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies a parameterized family of *concurrent transitions* in the system.

Rewriting logic's inference system allows us to infer *all* the possible finitary concurrent computations of a system specified as a rewrite theory $\mathcal{R}$ as follows:

- **Reflexivity** is just the possibility of having *idle* transitions.
- **Equality** means that states are equal *modulo* E.
- **Congruence** is a general form of *sideways parallelism*.
- **Replacement** combines an *atomic transition* at the top using a rule with *nested concurrency* in the substitution.
- **Transitivity** is *sequential composition*.

However, both the sideways parallelism allowed by the **Congruence** rule, and the nested concurrency allowed by the **Replacement** rule can be restricted in part by the presence of frozen operator positions and variables, as specified in those inference rules.

Logically, a rewrite theory specifies a *logic* whose set of *formulas* is (a kind in) the initial algebra $T_{\Sigma/E}$. Then, each rewrite rule specifies an *inference rule* in the logic. Then, from the logical point of view, what rewriting logic's rules of deduction allow us to do is to derive all the different *proofs* possible in the logic thus specified by a rewrite theory $\mathcal{R}$. In this sense, rewriting logic serves as a *metalogic*, whose inference rules (rewriting logic's rules of deduction) do not change, but can be applied to derive proofs in *any finitary logic*, by representing that logic's inference rules as the rewrite rules of a given rewrite theory (or, in general, of a class of rewrite theories).

For example, the *minimal logic of implication* can be specified as the rewrite theory,

```
mod MINIMALR is sorts SentConstant Formula Configuration .
subsorts SentConstant < Formula < Configuration .
op _→_ : Formula Formula -> Formula [ctor] .
op empty : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration [ctor assoc comm id: empty] .
vars A B C : Formula .
rl [ax.K] :       empty
           => ----------------
               A → (B → A) .
rl [ax.S] :                          empty
           => ------------------------------------------
              (A → B) → ((A →(B → C)) → (A → C)) .
rl [mp] :      (A → B)    A
           => ------------------
                         B .
endm
```

where we have taken advantage of Maude's convention for comments (that can be introduced with three or more dashes) to highlight how each inference rule is naturally represented as a rewrite rule.

The computational and logical readings are *not* mutually exclusive. Rather, the *same theory* can be regarded computationally, or logically, or *both!*, depending on one's point of view. For example, logically, our `PETRI-MACHINE` example is a *linear logic theory* [18] in disguise. It is just a matter of a slight change of syntax, replacing the empty syntax $\_\,\_$ by that of linear logic's *multiplicative conjunction* operator $\_ \otimes \_$.

The operator $\_\otimes\_$ can be viewed as a form of *resource-conscious* non-idempotent conjunction. Then, the state $a \otimes q \otimes q$ corresponds to having an apple *and* a quarter *and* a quarter, which is a strictly better situation than having an apple *and* a quarter (non-idempotence of $\otimes$).

Then, in order to get the tensor theory corresponding to `PETRI-MACHINE`, it is enough to change the arrows into turnstiles, getting the following axioms:

$$\begin{array}{ll} \textit{buy-c}: & \$ \vdash c \\ \textit{buy-a}: & \$ \vdash a \otimes q \\ \textit{change}: & q \otimes q \otimes q \otimes q \vdash \$ \end{array}$$

The point is that we have the following equivalences between these two readings:

$$\begin{array}{ccccc} \textit{State} & \longleftrightarrow & \textit{Term} & \longleftrightarrow & \textit{Proposition} \\ \\ \textit{Computation} & \longleftrightarrow & \textit{Rewriting} & \longleftrightarrow & \textit{Proof} \\ \\ \begin{array}{c}\textit{Distributed}\\ \textit{Structure}\end{array} & \longleftrightarrow & \begin{array}{c}\textit{Algebraic}\\ \textit{Structure}\end{array} & \longleftrightarrow & \begin{array}{c}\textit{Propositional}\\ \textit{Structure}\end{array} \end{array}$$

In particular, *concurrent computations* in our Petri net example *coincide* with linear logic *proofs*.

3.8 Reachability Models in General

Given a general rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, its $\mathcal{R}$*-reachability relation* $\rightarrow_{\mathcal{R}}$ (also called $\mathcal{R}$*-rewriting relation*, or $\mathcal{R}$*-provability relation*) is defined proof-theoretically, for each kind k in Σ and each $[t], [t'] \in T_{\Sigma/E,k}$, by the equivalence,

$$[t] \rightarrow_{\mathcal{R}} [t'] \quad \Leftrightarrow \quad \mathcal{R} \vdash (\forall \emptyset)\, t \longrightarrow t',$$

which by the **Equality** rule is independent of the choice of t, t'. Model-theoretically, $\mathcal{R}$*-reachability* can be defined as the family of relations, indexed by the kinds k in Σ, interpreting the sorts $Arrow_k$ in the initial model of the following membership equational theory $Reach(\mathcal{R})$. The signature, equations, and memberships of $Reach(\mathcal{R})$ contain the signature, equations, and memberships of (Σ, E), to which we add:

- for each kind k in Σ a new kind $[Pair_k]$ with four sorts: $Arrow^0_k$, $Arrow^1_k$, $Arrow_k$ and $Pair_k$, and subsort inclusions $Arrow^0_k\ Arrow^1_k < Arrow_k < Pair_k$, and with operators $(\_ \rightarrow \_) : k\ k \longrightarrow Pair_k$, and $s, t : Pair_k \longrightarrow k$, satisfying the equations, $s(x \rightarrow y) = x$, $t(x \rightarrow y) = y$, and $(s(z) \rightarrow t(z)) = z$;
- we lift each $f : k_1 \ldots k_n \longrightarrow k$ in Σ with $\{1, \ldots, n\} - \phi(f) = \{i_1, \ldots, i_m\} \neq \emptyset$ to, $f : [Pair_{k_1}] \ldots [Pair_{k_n}] \longrightarrow [Pair_k]$, and for each $1 \leq j \leq m$ give also an operator declaration $f : Arrow^0_{k_1} \ldots Arrow^1_{k_{i_j}} \ldots Arrow^0_{k_n} \longrightarrow Arrow^1_k$; we then give for each $1 \leq j \leq m$ an equation,
$$f((x_1 \rightarrow x_1), \ldots, (x_{i_j} \rightarrow y_{i_j}), \ldots, (x_n \rightarrow x_n)) =$$
$$f(x_1, \ldots, x_{i_j}, \ldots, x_n) \rightarrow f(x_1, \ldots, y_{i_j}, \ldots, x_n);$$
- we also define for each kind k an operator, $\_ ; \_ : [Pair_k]\ [Pair_k] \longrightarrow [Pair_k]$, and add the equation, $(x \rightarrow y); (y \rightarrow z) = (x \rightarrow z)$;

- for each kind k, we give the membership, $(x \to x) : Arrow_k^0$, and the conditional membership,

$$(x \to z) : Arrow_k \quad \Leftarrow \quad (x \to y) : Arrow_k \wedge (y \to z) : Arrow_k;$$

- for each rewrite rule in R,

$$r : (\forall X)\ t \longrightarrow t' \ \Leftarrow\ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l')$$

  with, say t, t' of kind k, and w_l, w_l' of kind k_l, we give the conditional membership,

$$(\forall X)\ (t \to t') : Arrow_k^1 \ \Leftarrow\ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l (w_l \to w_l') : Arrow_{k_l}).$$

Since we are interested in identifying *one-step rewrites*, in which only one subterm is rewritten, the signature of $Reach(\mathcal{R})$ contains the sort $Arrow_k^1$, and the axioms of $Reach(\mathcal{R})$ mimic the inference rules of rewriting logic in a somewhat *restricted* way. Specifically, $Reach(\mathcal{R})$ has axioms simulating:

- a restricted **Congruence** rule so that only one nonfrozen position can be rewritten at a time,
- a restricted **Replacement** rule so that no rewrites are allowed in the substitution.

However, this restricted inference system implicitly simulated by $Reach(\mathcal{R})$ can derive the same inferences $\mathcal{R} \vdash (\forall\emptyset)\ t \longrightarrow t'$ as the original one.

The key point of defining $Reach(\mathcal{R})$ is to have an *algebraic model* for the $\mathcal{R}$-reachability relation $\to_{\mathcal{R}}$, given by the interpretation of the sorts $Arrow_k$ in the initial model $T_{Reach(\mathcal{R})}$. Specifically, we have the following theorem:

Theorem. [3] For $\mathcal{R} = (\Sigma, E, \phi, R)$ a rewrite theory and $t, t' \in T_{\Sigma,k}$ we have the equivalences,

$$\mathcal{R} \vdash (\forall\emptyset)\ t \longrightarrow t' \quad \Leftrightarrow \quad Reach(\mathcal{R}) \vdash (\forall\emptyset)\ (t \to t') : Arrow_k$$

$$\Leftrightarrow \quad [(t \to t')] \in T_{Reach(\mathcal{R})_{Arrow_k}}.$$

Reachability models, as their name suggests, focus only on the reachability relation. Rewriting logic, in the general form we have just described, has much more general *true concurrency models*, which can also be described as the algebras of an appropriate membership equational theory [3]. Such true concurrency models, including initial and free models, generalize those in [33] and have for each kind a *category* of concurrent computations, where each computation is an *equivalence class of proofs* describing the same truly concurrent computation according to natural equations. In such models the **Equality** inference rule gives rise to identities, and the **Transitivity** inference rule corresponds to arrow composition in the category. True concurrency models are indeed more expressive, and contain reachability models as a degenerate special case. However, for reasoning about the temporal logic properties of a concurrent system specified by a rewrite theory it is enough to consider reachability models.

4 Linear Temporal Logic and Concurrent Program Verification

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*. We will also consider the verification of *imperative concurrent programs*.

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.

Specifically, we have considered the *reachability* initial model $T_{Reach(\mathcal{R})}$ of a rewrite theory $\mathcal{R}$. 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 such that $T_{Reach(\mathcal{R})} \models \varphi$? One possibility is to use the *first-order language* $FOL(Reach(\mathcal{R}))$ associated to the theory $Reach(\mathcal{R})$. In particular, given a rewrite theory $\mathcal{R}$, the *modal logic* $\mathcal{M}(\mathcal{R})$, expressing properties based on *necessity*, $\Box\varphi$, and *possibility*, $\Diamond\varphi$, can be regarded as a *sublanguage* of $FOL(Reach(\mathcal{R}))$ (for a careful explanation of modal logic and its relationship to first-order logic see [43]).

But not all properties of interest are expressible in $FOL(Reach(\mathcal{R}))$. For example, properties involving *fairness*, and other 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) [29, 5], because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures; however, much of what we say could be extended to more expressive temporal logics such as, for example, CTL* [5].

We first introduce Kripke structures and the LTL syntax and semantics. We then show how a rewrite theory with a chosen kind and some equationally defined predicates gives rise to a Kripke structure and has an associated temporal logic. We then discuss some verification techniques for LTL properties of rewrite theories and use them to verify concurrent declarative programs. Finally, we discuss the verification of LTL properties of imperative concurrent programs.

4.1 Kripke Structures and LTL

Kripke structures are the natural models for *propositional temporal logic*. Essentially, a Kripke structure is a (total) *unlabeled transition system* to which we have added a collection of unary state predicates on its set of states.

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.

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,

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

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

- **True**. $\top \in LTL(AP)$.
- **Atomic Propositions**. 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 $LTL(AP)$.

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 *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 $a \in A$ and all paths $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$, the *quaternary satisfaction relation*,

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

holds. 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 $p \in AP$,
$$\mathcal{A}, a, \pi \models_{LTL} p \quad \Leftrightarrow \quad p \in L(a).$$
- For $\bigcirc\varphi \in LTL(A)$,
$$\mathcal{A}, a, \pi \models_{LTL} \bigcirc\varphi \quad \Leftrightarrow \quad \mathcal{A}, \pi(1), s; \pi \models_{LTL} \varphi,$$
where $s : \mathbb{N} \longrightarrow \mathbb{N}$ is the successor function.
- For $\varphi\, \mathcal{U}\, \psi \in LTL(\mathcal{A})$,
$$\mathcal{A}, a, \pi \models_{LTL} \varphi\, \mathcal{U}\, \psi \quad \Leftrightarrow$$
$$(\exists n \in \mathbb{N})\ ((\mathcal{A}, \pi(n), s^n; \pi \models_{LTL} \psi) \wedge ((\forall m \in \mathbb{N})\ m < n\ \Rightarrow \mathcal{A}, \pi(m), s^m; \pi \models_{LTL} \varphi)).$$
- 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 $\varphi \vee \psi \in LTL(AP)$,
$$\mathcal{A}, a, \pi \models_{LTL} \varphi \vee \psi \quad \Leftrightarrow$$
$$\mathcal{A}, a, \pi \models_{LTL} \varphi \ \text{ or } \ \mathcal{A}, a, \pi \models_{LTL} \psi.$$

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

4.2 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*. We can assume that they have been defined by means of equations D in an equational theory $(\Sigma', E \cup D)$ extending (Σ, E) as a subtheory in *protecting*[9] 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 $p : s_1 \ldots s_n \longrightarrow Prop$. For example, a state predicate `inCrit`, testing whether or not an object is in its critical section, may depend on the object identifier and could be defined in Maude with syntax,

```
op inCrit : Oid -> Prop .
```

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 ground 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,

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

In practice 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 for $D \cup E$ of this form, *only the cases when a predicate holds* need be specified by $D \cup E$: when a ground expression $t \models p(u_1, \ldots, u_n)$ cannot be simplified to $true$, then by the above definition and the decidability assumptions on $D \cup E$ the predicate does *not* hold[10].

We are now ready to associate to a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ (with a selected kind k 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,

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

The Kripke structure we are interested in is then, $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi)$, where $(\rightarrow^1_{\mathcal{R}})^\bullet$ denotes the total relation extending the one-step $\mathcal{R}$-rewriting relation $\rightarrow^1_{\mathcal{R}}$ among states of kind k, that is,

$$[t] \rightarrow^1_{\mathcal{R}} [t'] \quad \Leftrightarrow \quad Reach(\mathcal{R}) \vdash (\forall\, \emptyset)\ (t \rightarrow t') : Arrow^1_k.$$

Of course, we have the LTL language $LTL(AP_\Pi)$. 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,

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

---

[9] 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 $h_s : T_{\Sigma/E,s} \longrightarrow T_{\Sigma'/E \cup D,s}$.

[10] This means that to specify the semantics of the state predicates it is enough to have in $D \cup E$ (possibly conditional) equations of the general form,

$$t \models p(v_1, \ldots, v_n) = true \quad \Leftarrow \quad C.$$

There is in principle no need to specify when a state predicate is *false*. However, if so desired one can specify both the true and false cases by giving (possibly conditional) equations of the more general form,

$$t \models p(v_1, \ldots, v_n) = bexp \quad \Leftarrow \quad C,$$

where *bexp* is an arbitrary Boolean expression.

4.3 Verifying LTL Properties of Declarative Concurrent Programs

For declarative concurrent programs written in Maude, we discuss both model checking of systems with finite sets of reachable states, and formal analysis of systems with infinite sets of reachable states.

4.3.1 Model Checking and Maude's LTL Model Checker

It is well-known that for any (effectively presented, i.e., computable) Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$, any state $a \in A$ such that the set

$$Reach_{\mathcal{A}}(a) = \{x \in A \mid \exists \pi \in Path(\mathcal{A})\ \exists n \in \mathbb{N}\ \ s.t.\ \pi(0) = a \wedge \pi(n) = x\}$$

of states *reachable* from a in $\mathcal{A}$ is *finite*, and any LTL formula $\varphi \in LTL(AP)$, there is a *decision procedure* that can *effectively decide* the satisfaction relation,

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

Furthermore, if $\mathcal{A}, a \not\models_{LTL} \varphi$ the decision procedure will exhibit a *counterexample*, that is, a path not satisfying φ.

A decision procedure of this kind is called a *model checking algorithm*, since it checks whether φ holds in the model $\mathcal{A}$ with initial state a. Detailed discussion of such algorithms for a variety of temporal logics such as LTL, CTL, CTL*, and the μ-calculus, can be found in the excellent textbook [5]. There are two main classes of model checking algorithms:

- *explicit-state* model checking algorithms, that explicitly search the state space of $\mathcal{A}$ to find a counterexample; and
- *symbolic model checking* algorithms, that use a symbolic Boolean representation (typically BDDs) of *sets of states* to compute the fixpoint of the transition relation, i.e., the set $Reach_{\mathcal{A}}(a)$.

Maude 2.0 supports on-the-fly, explicit-state LTL model checking for initial states $[t]$ of a system specified by a rewrite theory $\mathcal{R}$ such that the set of all states *reachable* from $[t]$ is *finite* [17]. Note that 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 [15, 16] 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.

Given a rewrite theory $\mathcal{R}$ (with a selected kind k of states and with state predicates Π) and given $\varphi \in LTL(AP_{\Pi})$, to decide the satisfaction,

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

from a state $[t]$ having a *finite* reachability set, the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ should be finitary (finite signature, and finite number of equations, memberships, and rules) and should satisfy the following *requirements*:

- (Σ, E) and its extension $(\Sigma', E \cup D)$ defining the relevant state predicates Π are confluent, sort-decreasing, and terminating (perhaps *modulo* some axioms $A \subseteq E$);
- any variable not appearing in the lefhand side of a rule in R should appear in the condition; and should first appear either in a "matching" (`:=`) condition (see [6]), or in the righthand side of a rewrite condition for which all lefthand side ground instances have finite reachability sets that can be searched in finite time; and
- the rules R are *coherent* [45] relative to the equations E; that is, if $t \rightarrow^1_{\mathcal{R}} t'$, then there is a t'' such that $can_E(t) \rightarrow^1_{\mathcal{R}} t''$, and $can_E(t') = can_E(t'')$ (again, perhaps modulo some axioms A).

Given a rewrite theory $\mathcal{R}$ satisfying the above assumptions and specified in Maude by a system module, say `M`, and given an initial state, say `init` of kind $[\mathtt{State}]_{\mathtt{M}}$, 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 Maude's predefined MODEL-CHECKER module as submodules;
- giving a *subsort declaration*, subsort $\texttt{State}_{\texttt{M}}$ < State ., where State is one of the key sorts in the MODEL-CHECKER module (this declaration can be omitted if $\texttt{State}_{\texttt{M}} = \texttt{State}$);
- defining the *syntax* of the *state predicates* Π 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;
- 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, and therefore [Result]=[Bool]. We define the semantics of each state predicate $p \in \Pi$ by (possibly conditional) equations according to the method explained in Section 4.2.

If our original system module M specified a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, then the above steps ensure that our module CHECK-M includes both the equational theory $(\Sigma', E \cup D)$ specifying the state predicates Π, and Maude's predefined MODEL-CHECKER module.

We are then ready, given an initial state init, to model check any LTL formula, say form, involving such predicates, that is, a formula in $LTL(AP_\Pi)$. Such LTL formulas are *ground terms* of sort Formula in CHECK-M. We do so by giving the Maude command,

```
reduce init |= form .
```

assuming, as already mentioned, that the set of reachable states is finite. Two things can then happen: if the property form holds, then 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, if an LTL formula φ is not satisfied by a finite Kripke structure, it is always possible to find a counterexample for φ having the form of a *path of transitions followed by a cycle*. 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.

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

```
fmod LTL is
  sorts Prop Formula .
  subsort Prop < 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] .   *** weak until
  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 equations in this module do two things: (1) they express all defined LTL operators in terms of the basic operators `True`, `False`, negation, conjunction, disjunction, next `O`, until `U`, and release `R`; and (2) they transform the LTL formula using only those basic operators into an equivalent one in *negative normal form*, that is, the negations are *pushed all the way down into the state predicates*.

Maude's LTL model checker performs on-the-fly explicit-state LTL model checking using the double depth first method of Holzmann et al. [25, 26]. As explained in [17], 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, expressed as a Büchi automaton. The LTL model checker then searches on-the-fly the synchronous product of the tableau for the negated formula and the reachability model for the module `M` to either establish the property or 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 Token Ring Example

The use of the Maude LTL model checker can be illustrated by verifying some basic properties of a *token ring architecture of* n *processes*, where n is a parameter. The properties in question are: mutual exclusion, which is achieved by each process passing on a token to the next process in the ring when it is done using the (implicit) shared resource; and a certain liveness property.

We first need a Full Maude[11] *theory*[12] specifying the choice of a nonzero natural number n.

```
(fth NZNAT* is
   protecting NAT .
   op * : -> NzNat .
 endfth)
```

Note that in this theory the interpretation of natural numbers is *fixed*, with its usual initial algebra semantics, thanks to the `protecting` keyword; however, the theory has many different models, namely, one for each choice of a nonzero number as the interpretation of the constant `*`.

We then specify natural numbers modulo n as the following parameterized module in Full Maude,

```
(fmod NAT/(X :: NZNAT*) is
   sort Nat/(X) .
```

[11] All inputs to Full Maude must be enclosed in parentheses.

[12] In Full Maude [14], as in OBJ3 [21], we can specify not only modules with initial (or free extension) semantics, but also *theories* with "loose" semantics; that is, any model of the axioms is in principle a possible model; however, such theories may have *initiality constraints* [19, 12] reducing the class of allowable models. For example, in the `NZNAT*` theory the `protecting NAT` declaration is an initiality constraint requiring that the restriction of the models to the subtheory `NAT` yields the standard natural numbers.

```
  op `[_`] : Nat -> Nat/(X) .
  op _+_ : Nat/(X) Nat/(X) -> Nat/(X) .
  op _*_ : Nat/(X) Nat/(X) -> Nat/(X) .
  vars N M : Nat .
  ceq [N] = [N rem *] if N >= * .
  eq [N] + [M] = [N + M] .
  eq [N] * [M] = [N * M] .
 endfm)
```

The token ring module parameterized by n is the following parameterized *object-oriented module*[13]:

```
(omod TOK-RING(X :: NZNAT*) is protecting NAT/(X) .
  sort Mode .   subsort Nat/(X) < Oid .
  ops wait crit : -> Mode .
  msg tok : Nat/(X) -> Msg .
  op init : -> Configuration .
  op make-init : Nat/(X) -> Configuration .
  class Proc | mode : Mode .
  var N : Nat .
  ceq init = tok([0]) make-init([N]) if s(N) := * .
  ceq make-init([s(N)])
    = < [s(N)] : Proc | mode : wait > make-init([N])
    if N < * .
  eq make-init([0]) = < [0] : Proc | mode : wait > .
  rl [enter] : tok([N]) < [N] : Proc | mode : wait >
    => < [N] : Proc | mode : crit > .
  rl [exit] : < [N] : Proc | mode : crit >
    => < [N] : Proc | mode : wait > tok([s(N)]) .
 endom)
```

We can consider two important properties about this module, beginning in the initial state `init`, namely:

1. *mutual exclusion*, that is, we never have two process simultaneously in `crit` mode, and
2. *guaranteed reentrance*, that is:
   - each process eventually reaches its critical section, and
   - it does so again after $2 \times n$ steps.

Given the parametric nature of our specification, there isn't a single LTL formula stating each property for all n. In fact, for each property there is a *different* formula for each n. However, in Full Maude we can specify once and for all these two properties by parametric equational formula definitions as follows:

```
(omod CHECK-TOK-RING(X :: NZNAT*) is
 inc TOK-RING(X) .
 inc MODEL-CHECKER .
 subsort Configuration < State .

 op inCrit : Nat/(X) -> Prop .
 op twoInCrit : -> Prop .

 var  N : Nat .
 vars A B : Nat/(X) .
 var  C : Configuration .
 var  F : Formula .

 eq < A : Proc | mode : crit > C |= inCrit(A) = true .
 eq < A : Proc | mode : crit > < B : Proc | mode : crit > C
```

[13]Full Maude allows specifying object-oriented systems with the syntactic conventions already discussed in Section 3.4. Such *object-oriented modules* are introduced with the `omod` keyword.

```
      |= twoInCrit = true .

  op guaranteedReentrance : -> Formula .
  op allProcessesReenter : Nat -> Formula .
  op nextIter_ : Formula -> Formula .
  op nextIterAux : Nat Formula -> Formula .

  ceq guaranteedReentrance = allProcessesReenter(N) if s(N) := * .

  eq allProcessesReenter(s(N))
    = (<> inCrit([s(N)])) /\
       [] (inCrit([s(N)]) -> (nextIter inCrit([s(N)]))) /\
      allProcessesReenter(N) .
  eq allProcessesReenter(0) = (<> inCrit([0])) /\
       [] (inCrit([0]) -> (nextIter inCrit([0]))) .

  eq nextIter F = nextIterAux(2 * *, F) .
  eq nextIterAux(s N, F) = O nextIterAux(N, F) .
  eq nextIterAux(0, F) = F .

 endom)
```

We cannot model check these properties directly in their *parameterized* form. However, for each nonzero value n we can model check the corresponding *instance* of these properties. For example, for $n = 5$ we define in Full Maude the *view*[14],

```
(view 5 from NZNAT* to NAT is
   op * to term 5 .
 endv)
```

Then we can model check the mutual exclusion property for 5 processes by instantiating the module CHECK-TOK-RING(X) with the view 5 and giving to Full Maude the command,

```
(red in CHECK-TOK-RING(5) : init |= [] ~ twoInCrit .)
reduce in CHECK-TOK-RING<5> :
result Bool : true
```

Similarly, we can model check the *guaranteed reentrance* property for $n = 5$ by instantiating the CHECK-TOK-RING(X) module with the view 5 and giving to Full Maude the command,

```
(red in CHECK-TOK-RING(5) : init |= [] guaranteedReentrance .)
reduce in CHECK-TOK-RING<5> :
  init |=[]guaranteedReentrance
result Bool : true
```

4.3.2 Formal Analysis of Infinite-State Declarative Concurrent Programs

As usual for model checkers, the Maude LTL model checker relies quite essentially on the assumption that the set of states reachable from an initial state is *finite*. Yet, many concurrent systems of interest fail to have this property. For example, just adding a few fault-tolerant features to an asynchronous communication protocol can easily bring us into an infinity of reachable states. Likewise, crypto protocols with powerful enough intruder models easily have infinite sets of reachable states.

What can we do in such cases? Is some kind of formal analysis still applicable? In practice, the most vital properties are often *safety properties*. For such properties, under reasonable assumptions, we can indeed do something quite practical, namely *breadth-first search for a counterexample*. This provides a

[14] Again, as in OBJ3 [21], in Full Maude [14] the user can instantiate a *parameter theory*, for example NZNAT*, by a *view*, that is, a theory interpretation translating the sorts and operations of the parameter theory into a target theory, for example NAT, and preserving the axioms in the translation. Note that, in this case, each choice of a different nonzero natural number n can be expressed as a different view n from NZNAT* to NAT.

semidecision procedure for the *failure* of the safety property. That is, if the safety property fails we will always *find this out after a finite number of steps* (in practice, of course, should be "finite enough" so as not to run out of memory!). Because of the infinite search involved, by this method *we can never know for sure* that the safety property holds; but, disregarding time and space limitations, we have a *complete method* for *finding safety bugs* in a specification or program.

But we have to be more precise. What are the "reasonable assumptions" about the rewrite theory $\mathcal{R}$ and the safety properties to be checked? As usual, $\mathcal{R} = (\Sigma, E, \phi, R)$ should have (Σ, E) confluent, terminating, and sort-decreasing. And the rules R should be coherent relative to E and should have no extra variables in their conditions and righthand sides except for variables introduced in either "matching" equations or in righthands of rewrite conditions. In short, $\mathcal{R} = (\Sigma, E, \phi, R)$ should be an *admissible* rewrite theory in the sense of [6], so that it can be executed in Maude. Furthermore, for each rewrite rule having a rewrite in its condition that rewrite condition's ground lefthand side instances should have finite reachability sets that can be searched in finite time.

Regarding the safety property, we assume a kind k of states, and a signature Π of predicate symbols whose semantics has been defined in a protecting extension $(\Sigma', E \cup D)$ of (Σ, E) according to the method described in Section 4.2. The property itself should be of the form $p \to \Box Q$, with:

- p an unparameterized state predicate for which we can effectively determine that the set of states satisfying it is *finite*, and we can enumerate those states, say, $\{init_1, \ldots, init_n\}$,
- Q an *arbitrary Boolean combination* of state predicates $\theta_1(q_1), \ldots, \theta_n(q_n) \in AP_\Pi$.

Let $\neg Q = bexp(\theta_1(q_1), \ldots, \theta_n(q_n))$. Since, by definition, a basic predicate $\theta_j(q_j)$ holds at a state $[t]$ iff $t \models \theta_j(q_j)$ can be simplified to *true*, we can then evaluate in Maude $t \models \neg Q$ to either `true` or `false` by evaluating the Boolean expression,

$$bexp((t \models \theta_1(q_1) \texttt{ == true}), \ldots, (t \models \theta_n(q_n) \texttt{ == true})).$$

Therefore, under the above assumptions about $\mathcal{R}$, p, and Q, our desired semidecision procedure for the failure of the safety property $p \to \Box Q$ can be implemented in Maude by running in parallel, or in a fair scheduling of n processes, the n (breadth-first) search commands, $1 \leq j \leq n$,

$$\texttt{search [1] } init_j \texttt{ =>* S s.t. } bexp((\texttt{S} \models \theta_1(q_1) \texttt{ == true}), \ldots, (\texttt{S} \models \theta_n(q_n) \texttt{ == true})) \texttt{ .}$$

Of course, this is *not* the only method of dealing with infinite-state systems. Two other alternative methods are: (1) *deductive proof*; and (2) *model checking an abstraction*, that is, associating to $\mathcal{R}$ a more abstract version $\mathcal{R}'$ whose sets of reachable states are finite and such that,

$$\mathcal{K}(\mathcal{R}', k)_\Pi, [t] \models_{LTL} \varphi \;\;\Rightarrow\;\; \mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models_{LTL} \varphi.$$

For LTL deductive proof methods see [29, 30, 4]. For an abstraction technique specifically suited for rewrite theories and supported by Maude tools see [38].

A Cryptographic Protocol Example

We can illustrate the power of this formal analysis technique for safety properties of infinite state systems by showing how it can be used to find subtle *attacks* on *cryptographic protocols*, including some that have been used extensively and have been considered secure for a long time. One such protocol is the 1978 Needham-Schroeder authentication protocol (NSPK) for which a subtle "man-in-the-middle" attack was found by G. Lowe in 1996 [27].

The goal of the NSPK Protocol is to provide *authentication* of two agents who want to be assured of each other's identity before they exchange safety-critical data. That is, an intruder should not be allowed to impersonate another agent. For this purpose, initiator and responder of a communication mutually authenticate each other. NSPK uses *public key cryptography*, i.e., each agent has a public key which can be accessed by all agents, and a secret key which is the inverse of the public key. Moreover, *nonces* are used in the protocol. Nonces are freshly generated, unguessable random numbers to be used in a single run of the protocol.

Here is a textbook-style simplified description of NSPK:

Message 1 $\quad A \to B : A.B.\{N_a, A\}_{PK(B)}$
Message 2 $\quad B \to A : B.A.\{N_a.N_b\}_{PK(A)}$
Message 3 $\quad A \to B : A.B.\{N_b\}_{PK(B)}$

This level of description is *ambiguous*, in that a fair amount of *implicit assumptions* are *left unspecified*. An object-oriented rewriting logic specification of the protocol in Maude (developed in joint work with G. Denker and C. Talcott [8]) makes these assumptions explicit, and allows our desired formal analysis.

We first specify key *algebraic properties* of the *cryptographic infrastructure* in a functional module.

```
fmod DATATYPES is
  sorts Key Field Nonce Principal Run Role EstabComm .
  subsort Nonce Principal Key < Field .
  op keypair : Key Key -> Bool [comm] .
  op ped : Key Field -> Field . *** encryption function
  op n : Principal Nat -> Nonce .
  ceq ped(sk,ped(pk,f)) = f  if keypair(sk,pk) .
  ...
endfm
```

The protocol itself is specified as follows (fragment):

```
class Agent | e\_com: EstabCom, sec\_key: Key, role\_i: Run, role\_r: Run,
              d\_com: FieldSet cnt: Nat .

msg from\_to\_send\_ : Principal Principal Field  -> Message .

vars A B P : Principal . vars RI RR : Run . vars NI : Nonce . ...
rl [BeginRun] :
  < A : Agent | role\_i: RI, d\_com: B U S, cnt: J >
  => < A : Agent | role\_i: RI U (n(A,J),B,mtfield), d\_com: S, cnt: J + 1 >
     from(A)to(B)send(ped(pk(B),{n(A,J),A})) .

crl [Message1Rec] :
  < B : Agent | sec\_key: SKB, role\_i: RI, role\_r: RR, cnt: J >
  from(A)to(B)send(ped(PKB,{F,A}))
  => < B : Agent | role\_r: RR U (n(B,J),A,F), cnt: J + 1 >
     from(B)to(A)send(ped(pk(A),{F.n(B,J)}))
  if keypair(SKB,PKB) and not(F in RR) .

crl [Message2RecCorrect] :
  < A : Agent | sec\_key: SKA, role\_i: RI U (NI,P,mtfield), e\_com: C >
  from(B)to(A)send(ped(PKA,{F}))
  => < A : Agent | role\_i: RI, ECom: C U (i,NI,B,rest(F)) >
     from(A)to(B)send(ped(pk(B),{rest(F)}))
  if keypair(SKA,PKA) and (B == P) and (NI == first(F)) .

crl [Message2RecIncorrect] :
  < A : Agent | sec\_key: SKA, role\_i: RI U (NI,P,mtfield) >
  from(B)to(A)send(ped(PKA,{F}))
  => < A : Agent | role\_i: RI >
  if keypair(SKA, PKA) and (NI == first(F)) and (B =/= P) .
```

The following fragment gives a flavor for the specification of intruders and their actions,

```
class Intruder | e_com: EstabCom sec_key: Key ncs: FieldSet,
                 msgs: FieldSet agents: FieldSet role_i: Run,
                 role_r: Run d_com: Field cnt: Nat.
crl [IntruderFakeMessage] :
```

```
  < I : Intruder | ncs: N U F, agents: S U A U B >
  => < I : Intruder | ncs: N U F > from(A)to(B)send(ped(pk(B),{F}))
  if B =/= I .

*** similar: IntruderInterceptMessage, IntruderOverhearMessage,
*** IntruderReplayMessage}
```

In a topmost version of the specification in which the configuration is enclosed in a `boundary` operator, the situation where *authentication information has been compromised* is specified by the following state predicate:

```
op attack : -> Prop .
op _|=_ : Configuration Prop -> [Bool] .

ceq boundary(
     < INTR : Intruder | ecom : EC, rolei : RI, roler : RR,
                         ncs : fset+(fset+(FSET1, N1), N2) >
     < A : NSPKAgent | ecom : ecom+(EC2, ecom(ROLE,N1,B,N2)) >
     Conf) |= attack = true
     if (not(inEstabCom(ecom(r,N2,A,N1),EC))
         and not(inEstabCom(ecom(i,N2,A,N1), EC))
         and not(in(N1,RI)) and not(in(N1,RR))
         and not(in(N2,RI)) and not(in(N2,RR))
         and B =/= INTR) == true .
```

The relevant *safety property* is that no such attack is possible under reasonable initial conditions. For example, we can consider a simple scenario with two agents and an attacker in an initial state `cf2Agents1Intruder` equationally defined in the obvious manner. Then, the desired safety property is `init` $\rightarrow \Box(\neg$ `attack`) where `init` is an atomic predicate that only holds true of the state `cf2Agents1Intruder`, and `attack` has the semantics defined above.

Maude's `search` command finds a counterexample to such a property:

```
Maude> search [1] cf2Agents1Intruder =>+ C:Configuration s.t.
                                   C:Configuration |= attack = true .
search [1] in NSPK : cf2Agents1Intruder =>+ C:Configuration such that
                                   C:Configuration |= attack = true .
Solution 1 (state 37826)
states: 37827  rewrites: 1274741 in 21790ms cpu (21790ms real) (58501
    rewrites/second)
C:Configuration --> boundary(< alice : NSPKAgent | cnt : 2,dcom : mtfset,roler
    : mtrun,rolei : mtrun,seckey : skalice,ecom : ecom(i, n(alice, 1), mrx, n(
    bob, 1)) > < bob : NSPKAgent | cnt : 2,dcom : mtfield,roler : mtrun,rolei :
    mtrun,seckey : skbob,ecom : ecom(r, n(bob, 1), alice, n(alice, 1)) > < mrx
    : Intruder | cnt : 1,dcom : mtfield,roler : mtrun,rolei : mtrun,seckey :
    skmrx,ecom : mtecom,agents : fset+(alice, bob, mrx),ncs : fset+(mtfset, n(
    alice, 1), n(bob, 1)),msgs : fset+(mtfset, ped(pkalice, cat(n(alice, 1), n(
    bob, 1)))) >)
```

We can find the actual sequence of rewrites leading to the attack by giving to Maude the command `show path 37826 .` A detailed trace is then shown. From this trace we can extract the following sequence of rewrite rule applications:

```
[BeginRun];[IntruderAcceptEveryMessage1];[IntruderFakeMessage1];

[Message1Rec];[IntruderInterceptMessage2];[IntruderReplayMessage];

[Message2Rec];[IntruderAcceptEveryMessage3];

[IntruderFakeMessage3];[Message3Rec]
```

which describes in our rewriting logic axiomatization the same "man in the middle" attack originally found by Lowe [27].

4.4 Verifying LTL Properties of Imperative Concurrent Programs

How can the rewriting logic approach that we have been developing in these lectures be applied to the verification of *imperative concurrent programs*, say in an imperative language $\mathcal{L}$? Essentially we have to do three things:

1. specify precisely the *semantics* of the imperative concurrent language $\mathcal{L}$ as a *rewrite theory* $\mathcal{R}_{\mathcal{L}}$,
2. choose an appropriate kind k of states in $\mathcal{R}_{\mathcal{L}}$ and define appropriate state predicates Π in an equational theory extending that of $\mathcal{R}_{\mathcal{L}}$ in protecting mode,
3. express the desired verification of properties of a program P in $\mathcal{L}$ as the satisfaction of, for example, an LTL formula (or formulas) φ by the Kripke structure $\mathcal{K}(\mathcal{R}_{\mathcal{L}}, k)_{\Pi}$, and apply some verification or proof method (model checking, search, abstraction, deductive proof, etc.) to either verify or disprove φ.

4.4.1 The Semantics of a Simple Parallel Language

We illustrate step (1) in the above process by defining in Maude the semantics of a simple parallel language as a rewrite theory. The language and its specification are exactly those in [17]. 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`.

```
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

---(Equality test comparing the contents of a named memory
location to a given machine integer.)

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

---(Syntax for a trivial sequential programming language.
We can abstract certain terminating, or potentially
nonterminating, program fragments as constants of sorts
UserStatement and LoopingUserStatement.)

fmod SEQUENTIAL is
  inc TESTS .

  sorts UserStatement LoopingUserStatement Program .
  subsort LoopingUserStatement < 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
```

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; this is used to express fairness properties.

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

```
---(The operational semantics of the programming language running
on this machine is given by just 6 rules.  The first two rules deal
with terminating and potentially nonterminating user statements.
Note that there is no need to give a rule for skip, because it is
the identity element for the sequential composition operator _;_.)

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 .
  var L : LoopingUserStatement .
  vars I J : Pid .
  var M : Memory .
  var Q : Qid .
  vars N X : Int .
  var T : Test .

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

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

4.4.2 Model Checking Dekker's Algorithm

We illustrate steps (2) and (3) in the verification of imperative concurrent programs, as well as the use of the Maude model checker, with the example of Dekker's algorithm, one of the earliest correct solutions to the mutual exclusion problem. The Maude specification of this algorithm is exactly as in [17]. 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 `p1` 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 right away. In case of a tie (both variables set to 1) the tie is broken using a variable `turn` that takes values in $\{1, 2\}$.

We can then define the two processes for Dekker's algorithm as programs in our simple language, as well as the desired initial state, in the following module extending `PARALLEL`:

```
---(The classical Dekker's algorithm for mutual exclusion between two
processes using 3 variables, 'c1, 'c2 and 'turn, in shared memory.
crit is used to represent the critical section (which must be
terminating) and rem is used to represent the remainder (non-critical)
part of each program, which may be nonterminating.)

mod DEKKER is
  inc PARALLEL .
  subsort Int < Pid .
  op crit : -> UserStatement .
  op  rem : -> LoopingUserStatement .
  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
```

where the fragments of code for the critical section and for the remaining part of the program are respectively abstracted as constants `crit` and `rem`. We assume that `crit` is *terminating*, but no such assumption is made about `rem`. In the Maude specification of our simple parallel language in Section 4.4.1 this is achieved by declaring subsorts and constants,

```
  subsorts LoopingUserStatement < UserStatement < Program .
  op crit : -> UserStatement .
  op  rem : -> LoopingUserStatement .
```

and by semantic rules where a `UserStatement` not in `LoopingUserStatement` always terminates, but a `LoopingUserStatement` may not terminate.

To specify relevant properties of Dekker's algorithm we define three state predicates parameterized by the process id: `enterCrit`, when the process enters its critical section; `in-rem`, when it is in its `rem` part; 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 in-rem exec : Pid -> Prop .
  var M : Memory .
  var R : Program .
  var S : Soup .
  vars I J : Pid .
  eq {[I, crit ; R] | S, M, J} |= enterCrit(I) = true .
  eq {[I, rem ; R] | S, M, J} |= in-rem(I) = true .
  eq {S, M, J} |= exec(J) = true .
endm
```

We can then verify that the *mutual exclusion property* is satisfied:

```
reduce in CHECK : initial |= []~ (enterCrit(1) /\ enterCrit(2)) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1156 in 40ms cpu (40ms real) (28900 rewrites/second)
result Bool: true
```

The *strong liveness property* that executing infinitely often implies entering one's critical section infinitely often fails. The Maude LTL model checker returns a counterexample.

```
reduce in CHECK : initial |= []<> exec(1) -> []<> enterCrit(1) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 16 system states.
rewrites: 148 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: 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 ;  ...
```

Since `rem` may not terminate, the weaker liveness property that if *both* `p1` and `p2` execute infinitely often, then both enter their critical sections infinitely often also fails.

```
reduce in CHECK : initial |= []<> exec(1) /\ []<> exec(2) ->
                           []<> enterCrit(1) /\ []<> enterCrit(2) .
ModelChecker: Property automaton has 7 states.
ModelCheckerSymbol: Examined 236 system states.
rewrites: 1463 in 60ms cpu (60ms real) (24383 rewrites/second)
result ModelCheckResult: 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 ;  ...
```

What *does* hold is the more subtle weak liveness property that if p1 and p2 both get to execute infinitely often, then if p1 is infinitely often out of its rem section, then p1 enters its critical section infinitely often. Of course, the symmetric statement holds true for p2.

```
reduce in CHECK : initial |= []<> exec(1) /\ []<> exec(2) ->
                      []<> ~ in-rem(1) -> []<> enterCrit(1) .
ModelChecker: Property automaton has 5 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1661 in 70ms cpu (70ms real) (23728 rewrites/second)
result Bool: true
```

The above Dekker algorithm example illustrates a general capability to model check in Maude *any* program (or abstraction of a program) having finitely many reachable states in *any* programing language: we just have to define in Maude the language's rewriting semantics and the state predicates.

5 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 *deductive and algorithmic techniques* and associated Maude tools has been shown applicable to the verification of:

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

For the applicability to *imperative sequential* programs see [37], which also explains how a "logic of programs" such as Hoare's logic—involving triples of the form $\{A\}P\{B\}$, with P the program and A, B formulas—can be seamlessly integrated with the approach proposed in these lectures. See also [22], on which our algebraic treatment of imperative sequential program semantics was inspired.

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 *declarative programs* and also *executable specifications* (for example, of the semantics of an imperative language) that can be *executed* and can be *symbolically simulated and analyzed* to discover many bugs; and
- 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.

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 are theories in equational or rewriting 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, in the near future it will be possible to execute Maude 2.0 in a distributed way.

However, this approach is very much *work in progress*; much more needs to be done to:

- develop new *proof techniques*,
- *improve* the existing tools,
- develop *new tools*, and
- develop many more *applications*, in the four areas of $\{$*declarative*, *imperative*$\}\times\{$*sequential*, *concurrent*$\}$ programs.

For example, tools supporting *deductive reasoning* and *abstraction* for proving *temporal logic* properties of rewrite theories should be added to the toolset. A tool for reasoning about *real time systems* expressed as *real-time rewrite theories* [40] already exists [39], and an improved version of it is currently under development. Similarly, theorem proving for *full first-order logic* would likewise be highly desirable to prove general properties of equational theories. Yet another useful technique worth investigating is *narrowing-based analysis* for both equational and rewrite theories. This could yield new notions of "symbolic model checking" quite different from the usual BDD-based notion; notions directly applicable to infinite state systems.

To *scale up* in the verification of large systems, it is crucial to develop *compositional proof techniques* that exploit the *modularity* of specifications to make proof efforts highly modular and reusable. In particular, the structure of Maude's *parameterized modules* and of its Full Maude *module calculus* should be systematically exploited by the proof tools, to achieve *generic*, or if you wish "polymorphic" proofs, reusable in many contexts. In general, a "structure and conquer" approach to taming complexity should be developed.

Much work remains ahead also in the area of Maude-based *executable specification, analysis, and verification of imperative languages*, both sequential and concurrent, including:

- Maude specification of entire conventional languages such as Java (or JVM), C++, and so on;
- development of "lightweight" Maude-based program analysis and synthesis tools, in the style of those already developed at NASA Ames and at UIUC by Grigore Roşu and his colleagues for runtime verification and monitoring and for domain-based certification [23, 24, 28];
- development of special-purpose reasoning tools for specific imperative languages (for example, the mechanization of a suitable Hoare logic for a given sequential language) based on each language's semantic definition in Maude.

In summary, much more research on proof methods, tools, and case studies remains ahead, but the prospects look promising.

Where to Go from Here

Lecture notes of a more comprehensive course on these ideas can be found in [37]. 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.cs.uiuc.edu/`

Acknowledgments

The development of the ideas and tools presented in these lectures is the fruit of a *distributed team effort* involving:

- G. Denker, S. Eker, P. Lincoln, and C. Talcott at SRI International in Menlo Park, California;
- R. Bruni, J. Meseguer, P. Ölveczky, M. Palomino, G. Roşu, M.-O. Stehr, and A. Sridharanarayanan at the Univ. of Illinois at Urbana-Champaign;
- M. Clavel, N. Martí-Oliet, and their students at the Universidad Complutense, Madrid;
- F. Durán and his students at the University of Málaga, Spain.

I would like to thank the organizers of the 2002 Marktoberdorf Summer School on "Models, Algebras, and Logic of Engineering Software" for giving me the opportunity of presenting these ideas in such a pleasant and stimulating environment, and the School participants, both faculty and students, for many discussions. In particular, I benefited from very helpful discussions with Amir Pnueli who suggested, among other things, an improvement to the treatment of Dekker's algorithm. Finally, I would also like to thank Manuel Clavel for his untiring and generous help with ITP matters, Miguel Palomino for his very careful reading of the manuscript and his many suggested improvements, and Dan Hu for her assistance with some of the pictures in the text.

References

[1] D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *FST TCS 2000*, pages 55–80. Springer LNCS, 2000.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] R. Bruni and J. Meseguer. Generalized rewrite theories. Manuscript, January 2003, `http://maude.cs.uiuc.edu`.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2001.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[7] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method.* Elsevier, 2000. `http://maude.cs.uiuc.edu`.

[8] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`.

[9] F. Durán. A reflective module algebra with applications to the Maude language. Ph.D. Thesis, University of Málaga, 1999.

[10] F. Durán. Coherence checker and completion tools for Maude specifications. Manuscript, Computer Science Laboratory, SRI International, `http://maude.cs.uiuc.edu/papers`, 2000.

[11] F. Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, `http://maude.cs.uiuc.edu/papers`, 2000.

[12] F. Durán and J. Meseguer. Structured theories and institutions. *Proc. Category Theory and Computer Science 1999*, (Edinburgh, Scotland, September 1999) ENTCS, Vol. 29, Elsevier, 1999, `http://www.elsevier.nl/locate/entcs/volume29.html`.

[13] F. Durán and J. Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, `http://maude.cs.uiuc.edu/papers`, 2000.

[14] F. Durán and J. Meseguer. On parameterized theories and views in Full Maude 2.0. In K. Futatsugi, editor, *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.* ENTCS, Elsevier, 2000.

[15] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway logic: Symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, January 2002.

[16] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott. Pathway Logic: executable models of biological networks. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications.* ENTCS, Elsevier, 2002.

[17] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications.* ENTCS, Elsevier, 2002.

[18] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[19] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[20] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[21] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.

[22] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996.

[23] K. Havelund and G. Roşu. Java PathExplorer — A runtime verification tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 18–22, 2001.

[24] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Proceedings First Workshop on Runtime Verification, RV'01, Paris, France, July 23, 2001*, volume 55 (2). ENTCS, Elsevier, 2001. `http://www.elsevier.nl/locate/entcs/volume55.html`.

[25] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *Proc. Second SPIN Workshop, August 1996, Vol. 32 of Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 81–89, Rutgers, Piscataway, NJ, 1997. American Mathematical Society.

[26] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *Design: An International Journal*, 13(3):289–307, nov 1998.

[27] G. Lowe. Breaking and fixing the Needham-Schroeder public key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[28] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 81–90. IEEE, 2001. Coronado Island, California.

[29] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer-Verlag, 1992.

[30] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems – Safety.* Springer-Verlag, 1995.

[31] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[32] J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[33] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[34] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[35] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.

[36] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997.* Springer-Verlag, 1999.

[37] J. Meseguer. Lecture notes on program verification. CS 376, University of Illinois, `http://www-courses.cs.uiuc.edu/~cs376/`, Fall 2002.

[38] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. Manuscript, January 2003, `http://maude.cs.uiuc.edu`.

[39] P. C. Ölveczky and J. Meseguer. Real-Time Maude: a tool for simulating and analyzing real-time and hybrid systems. ENTCS, Elsevier, 2000. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.

[40] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[41] C. A. Petri. Concepts of net theory. In *Mathematical Foundations of Computer Science*, pages 137–146. Mathematical Institute of the Slovak Academy of Sciences, 1973.

[42] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[43] J. van Benthem. *Exploring Logical Dynamics*. CSLI Publications, 1996.

[44] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[45] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.