Model Checking Verification in Maude

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

LTL Verification of Declarative Concurrent Programs

Proving that a Maude system module satisfies a property φ means proving that the corresponding initial model does:

$$\mathcal{T}_{\mathcal{R}} \models \varphi.$$

Invariants can be verified with the `search` command. But properties that talk about infinite behavior (e.g., fairness) require a richer logic, such as Linear Temporal Logic (LTL). Because in LTL we have a "next" operator $\bigcirc$ which talks not just about what is reachable in general, but what is reachable in one step, we will need a tighter notion of model than $\mathcal{T}_{\mathcal{R}}$. This is provided by the Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\Pi}$ associated to the rewrite theory $\mathcal{R}$ with state predicates Π. So our satisfaction problem will be recast as:

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

The Syntax of $LTL(AP)$

Given a set AP of *atomic propositions*, we define the formulae of 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 formulae $\neg\varphi$, and $\varphi \vee \psi$ are in $LTL(AP)$.

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

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

- Other Boolean connectives:
  - **False**: $\bot = \neg\top$
  - **Conjunction**: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - **Implication**: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.

- 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**: $\varphi \leadsto \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
  - **Strong implication**: $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
  - **Strong equivalence**: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$.

Kripke Structures

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 is not total, it can be made total by defining $R^{\bullet} = R \cup \{(a, a) \in A^2 \mid \nexists a' \in A\ (a, a') \in R\}$.

Kripke Structures (II)

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.

How can we associate a Kripke structure 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*, that is, the relevant set AP of atomic propositions. Having fixed k, our associated Kripke structure has as set of states those of kind k in the initial model, that is, $T_{\Sigma/E}$.

Kripke Structures (III)

The corresponding transition relation will the totalization $(\to^1_{\mathcal{R}})^\bullet$ of the one-step rewrite relation $\to^1_{\mathcal{R}}$ on $T_{\Sigma/E,k}$, where, by definition, $[t] \to^1_{\mathcal{R}} [t']$ iff there are terms $u \in [t]$ and $u' \in [t']$ and a proof $\mathcal{R} \vdash' u \to^1_{\mathcal{R}} u'$.

If $\mathcal{R}$ satisfies the usual executability requirements we have an isomorphism $\mathcal{T}_{\mathcal{R}} \cong \mathcal{C}_{\mathcal{R}}$ and our desired Kripke structure has a much more intiuitive equivalent representation: its set of states is the set of canonical terms $C_{\Sigma/E,k}$, and its transition relation is the totalization $(\to^1_{\mathcal{C}_{\mathcal{R}}})^\bullet$ of the one-step transition relation $\to^1_{\mathcal{C}_{\mathcal{R}}}$.

We will explain later in this lecture how the remaining part of the Kripke structure, namely the labeling function specifying the state predicates, can also be defined.

The Semantics of $LTL(AP)$

The semantics of the temporal logic LTL is defined by means of a satisfaction relation

$$\mathcal{A}, a \models \varphi$$

between a Kripke structure $\mathcal{A}$ having AP as its atomic propositions, a state $a \in A$, and an LTL formula $\varphi \in LTL(AP)$. Specifically, $\mathcal{A}, a \models \varphi$ holds iff for each path $\pi \in Path(\mathcal{A})_a$ the path satisfaction relation

$$\mathcal{A}, \pi \models \varphi$$

holds, where we define the set $Path(\mathcal{A})_a$ of computation paths starting at state a as the set of functions of the form $\pi : \mathbb{N} \longrightarrow A$ such that $\pi(0) = a$ and, for each $n \in \mathbb{N}$, we have $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

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

We can define the path satisfaction relation (for any path, beginning at any state) inductively as follows:

- We always have $\mathcal{A}, \pi \models_{LTL} \top$.
- For $p \in AP$,
$$\mathcal{A}, \pi \models_{LTL} p \quad \Leftrightarrow \quad p \in L(\pi(0)).$$
- For $\bigcirc\varphi \in LTL(A)$,
$$\mathcal{A}, \pi \models_{LTL} \bigcirc\varphi \quad \Leftrightarrow \quad \mathcal{A}, s; \pi \models_{LTL} \varphi,$$
where $s : \mathbb{N} \longrightarrow \mathbb{N}$ is the successor function.

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

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

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

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

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

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

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

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

The LTL Module

The LTL syntax, in a typewriter approximation of the mathematical syntax, is supported in Maude by the following LTL functional module (in the file `model-checker.maude`).

```
mod LTL is
  protecting BOOL .
  sort Formula .

  *** primitive LTL operators
  ops True False : -> Formula [ctor format (g o)] .
  op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _/\_ : Formula Formula -> Formula [comm ctor gather (E e)
                                       prec 55 format (d r o d)] .
  op _\/_ : Formula Formula -> Formula [comm ctor gather (E e)
                                       prec 59 format (d r o d)] .
```

```
op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _U_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .
op _R_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .

*** defined LTL operators
op _->_ : Formula Formula -> Formula [gather (e E) prec 65
                                                format (d r o d)] .
op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
op <>_ : Formula -> Formula [prec 53 format (r o d)] .
op []_ : Formula -> Formula [prec 53 format (r d o d)] .
op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
                                                     *** leads-to
op _=>_ : Formula Formula -> Formula [gather (e E) prec 65
                                               format (d r o d)] .
op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d)] .

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)) .
  eq f => g = [] (f -> g) .
  eq f <=> g = [] (f <-> g) .

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

The LTL Module (II)

Note that, for the moment, no set AP of atomic propositions has been specified in the LTL module. We will explain in what follows how such atomic propositions are defined for a given system module `M`, and how they are added to the LTL module as a subsort `Prop` of `Formula`.

Note that the nonconstructor connectives have been defined in terms of more basic constructor connectives in the first set of equations. But since there are good reasons to put an LTL formula in negative normal form by pushing the negations next to the atomic propositions (this is specified by the second set of equations) we need to consider also the duals of the basic connectives $\top$, $\bigcirc$, $\mathcal{U}$, and $\vee$ as constructors. That is, we need to also have as constructors the dual connectives: $\bot$, $\mathcal{R}$, and $\wedge$ (note that $\bigcirc$ is self-dual).

Associating Kripke structures to Rewrite Theories

Since the models of temporal logic are Kripke structures, we need to explain how we can associate a Kripke structure to the rewrite theory specified by a Maude system module `M`.

Indeed, we associate a Kripke structure to the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by a Maude system module `M` by making explicit two things: (1) the intended kind k of states in the signature Σ; and (2) the relevant state predicates, that is, the relevant set AP of atomic propositions.

In general, the state predicates need not be part of the system specification and therefore they need not be specified in our system module `M`. They are typically part of the property specification.

Associating Kripke structures to Rewrite Theories (II)

This is because the state predicates need not be related to the operational semantics of `M`: they are just certain predicates about the states of the system specified by `M` that are needed to specify some properties.

Therefore, after choosing a given kind, say `[Foo]`, in `M` as our kind for states we can specify the relevant state predicates in a module `M-PREDS` which is a protecting extension of `M` according to the following general pattern:

```
mod M-PREDS is protecting M .
  including SATISFACTION .
  subsort Foo < State .
  ...
endm
```

Associating Kripke structures to Rewrite Theories (III)

Where the dots '. . .' indicate the part in which the syntax and semantics of the relevant state predicates is specified, as further explained in what follows. The module SATISFACTION (which is contained in the file model-checker.maude) is very simple, and has the following specification:

```
fmod SATISFACTION is
  protecting BOOL
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

where the sort State is unspecified. However, by importing SATISFACTION into M-PREDS and giving the subsort declaration

Associating Kripke structures to Rewrite Theories (IV)

```
  subsort Foo < State .
```

all terms of sort `Foo` in `M` are also made terms of sort `State`. Note that we then have the kind identity, `[Foo]`$=$`[State]`.

The operator

```
  op _|=_ : State Prop -> Bool [frozen] .
```

is crucial to define the semantics of the relevant state predicates in `M-PREDS`. Each such state predicate is declared as an operator of sort `Prop`.

In standard LTL propositional logic the set AP of atomic propositions is assumed to be a set of constants.

Associating Kripke structures to Rewrite Theories (V)

In Maude we can define parametric state predicates, that is, operators of sort `Prop` which need not be constants, but may have one or more sorts as parameter arguments. We then define the semantics of such state predicates (when the predicate holds) by appropriate equations.

We can illustrate all this by means of a simple mutual exclusion example. Suppose that our original system module `M` is the following module `MUTEX`, in which two processes, one named `a` and another named `b`, can be either waiting or in their critical section, and take turns accessing their critical section by passing each other a different token (either `$` or `*`).

Associating Kripke structures to Rewrite Theories (VI)

```
mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf .
  op __ : Conf Conf -> Conf [assoc comm id: none] .
  ops a b : -> Name .
  ops wait critical : -> Mode .
  op [_,_] : Name Mode -> Proc .
  ops * $ : -> Token .
  rl [a-enter] : $ [a,wait] => [a,critical] .
  rl [b-enter] : * [b,wait] => [b,critical] .
  rl [a-exit] : [a,critical] => [a,wait] * .
  rl [b-exit] : [b,critical] => [b,wait] $ .
endm
```

Associating Kripke structures to Rewrite Theories (VII)

Our obvious kind for states is the kind `[Conf]` of configurations. In order to state the desired safety and liveness properties we need state predicates telling us whether a process is waiting or is in its critical section. We can make these predicates parametric on the name of the process and define their semantics as follows:

```
mod MUTEX-PREDS is protecting MUTEX .  including SATISFACTION .
  subsort Conf < State .
  ops crit wait : Name -> Prop .
  var N : Name .
  var C : Conf .
  eq [N,critical] C |= crit(N) = true .
  eq C |= crit(N) = false [owise] .
  eq [N,wait] C |= wait(N) = true .
  eq C |= wait(N) = false [owise] .
endm
```

Associating Kripke structures to Rewrite Theories (VIII)

The above example illustrates a general method by which desired state predicates for a module `M` are defined in a protecting extension, say `M-PREDS`, of `M` which imports `SATISFACTION`.

One specifies the desired states by choosing a sort in `M` and declaring it as a subsort of `State`. One then defines the syntax of the desired state predicates as operators of sort `Prop`, and defines their semantics by means of a set of equations that specify for what states a given state predicate evaluates to `true`.

We assume that those equations, when added to those of `M`, are (ground) Church-Rosser and terminating.

Associating Kripke structures to Rewrite Theories (IX)

Since we should protect `BOOL`, it is important to make sure that satisfaction of state predicates is fully defined. This can be checked with Maude's `SCC` tool.

This means that we should give equations for when the predicates are `true` and when they are `false`. In practice, however, this often reduces to specifying when a predicate is true by means of (possibly conditional) equations of the general form,

$$t \models p(v_1, \ldots, v_n) = \mathit{true} \ \ \textsf{if} \ C$$

because we can cover all the remaining cases, when it is false, with an equation

$$x : \mathit{State} \models p(y_1, \ldots, y_n) = \mathit{false} \quad [\texttt{owise}] \ .$$

Associating Kripke structures to Rewrite Theories (X)

In other cases, however —for example because we want to perform further reasoning using formal tools— we may fully define the true and false cases of a predicate not by using the `[owise]` attribute, but by explicit (possibly conditional) equations of the more general form,

$$t \models p(v_1, \ldots, v_n) = bexp \;\; \textit{if} \;\; C,$$

where $bexp$ is an arbitrary Boolean expression.

We can now associate to a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ (with a selected kind k of states and with state predicates Π defined by means of equations D in a protecting extension `M-PREDS` of `M`) a Kripke structure whose atomic predicates are specified by the set

Associating Kripke structures to Rewrite Theories (XI)

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

where, by convention, we use the simplified notation $\theta(p)$ to denote the ground term $\theta(p(x_1, \ldots, x_n))$.

This defines a labeling function L_Π on the set of states $T_{\Sigma/E,k}$ assigning to each $[t] \in T_{\Sigma/E,k}$ the set of atomic propositions,

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

The Kripke structure we are interested in is then

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

Associating Kripke structures to Rewrite Theories (XII)

If $\mathcal{R}$ satisfies the usual executability requirements we have the isomorphism $\mathcal{T}_{\mathcal{R}} \cong \mathcal{C}_{\mathcal{R}}$, and $\mathcal{K}(\mathcal{R}, k)_{\Pi}$ has an isomorphic representation as the Kripke structure $(C_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{C}_{\mathcal{R}}})^{\bullet}, L^{\mathcal{C}}_{\Pi})$, where, by definition, for each $t \in C_{\Sigma/E,k}$ we have,

$$L^{\mathcal{C}}_{\Pi}(t) = \{\theta(p) \in AP_{\Pi} \mid can_{E \cup D}(t \models \theta(p)) = true\}.$$

This is the most intituitive and computable representation for our desired Kripke structure, and indeed the one used by Maude for LTL model checking purposes. Therefore, to ensure correctness of LTL model checking in Maude it is essential to check that $\mathcal{R}$ satisfies the usual executability requirements.

Decidability of Propositional LTL

It is well-known that, for any 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 $\varphi \in LTL(AP)$, where $L : A \longrightarrow \mathcal{P}(AP)$, 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.

Decidability of Propositional LTL (II)

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, and CTL^* is beyond the scope of this lecture; see the excellent text "Model Checking" by Clark, Grumberg, and Peled. There are two rough classes of model checking algorithms:

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

The Maude Model Checker

Suppose that, given a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, we have:

- chosen a kind k in `M` as our kind of states;
- defined some state predicates Π and their semantics in a module, say `M-PREDS`, protecting `M` by the method already explained in this lecture.

Then, as explained earlier, this defines a Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$ on the set of atomic propositions AP_Π. Given an initial state $[t] \in T_{\Sigma/E,k}$ and an LTL formula $\varphi \in LTL(AP_\Pi)$ we would like to have a procedure to decide the satisfaction relation,

The Maude Model Checker (II)

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

By applying the general LTL decidability results to our Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$, this satisfaction relation becomes decidable if two conditions hold:

1. The set of states in $T_{\Sigma/E,k}$ that are reachable from $[t]$ by rewriting is finite.

2. The rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by `M` plus the equations D defining the predicates Π are such that:

The Maude Model Checker (III)

- both E and $E \cup D$ are (ground) Church-Rosser and terminating, perhaps modulo some axioms A, and
- R is (ground) coherent relative to E (again, perhaps modulo some axioms A).

Under these assumptions, both the state predicates Π and the transition relation $\rightarrow^1_{\mathcal{R}}$ are computable and, given the finite reachability assumption, we can then settle the above satisfaction problem using a model checking procedure. Specifically, Maude uses an on-the-fly LTL model checking procedure of the style described by Clark, Grumberg, and Peled.

The Maude Model Checker (III)

The basis of this procedure is the following. Each LTL formula φ has an associated Büchi automaton B_φ whose acceptance ω-language is exactly that of the behaviors satisfying φ. We can then reduce the satisfaction problem

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

to the emptiness problem of the language accepted by the *synchronous product* of $B_{\neg\varphi}$ and (the Büchi automaton associated to) $(\mathcal{K}(\mathcal{R}, k)_\Pi, [t])$. The formula φ is satisfied iff such a language is empty. The model checking procedure checks emptiness by looking for a counterexample, that is, an infinite computation belonging to the language recognized by the synchronous product.

The Maude Model Checker (IV)

This makes clear our interest in obtaining the negative normal form of a formula $\neg\varphi$, since we need it to build the Büchi automaton $B_{\neg\varphi}$.

For efficiency purposes we need to make $B_{\neg\varphi}$ as small as possible. The following module `LTL-SIMPLIFIER` (also in the `model-checker.maude` file) tries to further simplify the negative normal form of the formula $\neg\varphi$ in the hope of generating a smaller Büchi automaton $B_{\neg\varphi}$. This module is optional (the user may choose to include it or not when doing model checking) but tends to help building a smaller $B_{\neg\varphi}$.

The Maude Model Checker (V)

```
fmod LTL-SIMPLIFIER is
  including LTL .

  *** The simplifier is based on:
  ***   Kousha Etessami and Gerard J. Holzman,
  ***   "Optimizing Buchi Automata", p153-167, CONCUR 2000, LNCS 1877.
  *** We use the Maude sort system to do much of the work.

  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
  PE-Formula PU-Formula < Formula .

  op True : -> TrueFormula [ctor ditto] .
  op False : -> FalseFormula [ctor ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op O_ : PE-Formula -> PE-Formula [ctor ditto] .
op O_ : PU-Formula -> PU-Formula [ctor ditto] .
op O_ : PureFormula -> PureFormula [ctor ditto] .
op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .

vars p q r s : Formula .
var pe : PE-Formula .
var pu : PU-Formula .
var pr : PureFormula .
```

```
*** Rules 1, 2 and 3; each with its dual.
eq (p U r) /\ (q U r) = (p /\ q) U r .
eq (p R r) \/ (q R r) = (p \/ q) R r .
eq (p U q) \/ (p U r) = p U (q \/ r) .
eq (p R q) /\ (p R r) = p R (q /\ r) .
eq True U (p U q) = True U q .
eq False R (p R q) = False R q .

*** Rules 4 and 5 do most of the work.
eq p U pe = pe .
eq p R pu = pu .

*** An extra rule in the same style.
eq O pr = pr .

*** We also use the rules from:
***   Fabio Somenzi and Roderick Bloem,
*** "Efficient Buchi Automata from LTL Formulae",
***   p247-263, CAV 2000, LNCS 1633.
*** that are not subsumed by the previous system.
*** Four pairs of duals.
eq O p /\ O q = O (p /\ q) .
eq O p \/ O q = O (p \/ q) .
eq O p U O q = O (p U q) .
eq O p R O q = O (p R q) .
eq True U O p = O (True U p) .
eq False R O p = O (False R p) .
eq (False R (True U p)) \/ (False R (True U q)) =
                             False R (True U (p \/ q)) .
eq (True U (False R p)) /\ (True U (False R q)) =
                             True U (False R (p /\ q)) .

*** <= relation on formula
op _<=_ : Formula Formula -> Bool [prec 75] .

eq p <= p = true .
eq False <= p  = true .
eq p <= True = true .
ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
ceq p <= (q \/ r) = true if p <= q .
```

```
  ceq (p /\ q) <= r = true if p <= r .
  ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .
  ceq p <= (q U r) = true if p <= r .
  ceq (p R q) <= r = true if q <= r .
  ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
  ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
  ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
  ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

  *** condition rules depending on <= relation
  ceq p /\ q = p if p <= q .
  ceq p \/ q = q if p <= q .
  ceq p /\ q = False if p <= ~ q .
  ceq p \/ q = True if ~ p <= q .
  ceq p U q = q if p <= q .
  ceq p R q = q if q <= p .
  ceq p U q = True U q if p =/= True /\ ~ q <= p .
  ceq p R q = False R q if p =/= False /\ q <= ~ p .
  ceq p U (q U r) = q U r if p <= q .
  ceq p R (q R r) = q R r if q <= p .
endfm
```

The Maude Model Checker (VI)

Suppose that all the requirements listed above to perform model checking are satisfied. How do we then model check a given LTL formula in Maude for a given initial state $[t]$ in a module `M`? We define a new module, say `M-CHECK`, according to the following pattern:

```
mod M-CHECK is
  protecting M-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER . *** optional
  op init : -> k .           *** optional
  eq init = t .              *** optional
endm
```

The declaration of a constant `init` of the kind of states is not necessary: it is a matter of convenience, since the initial state `t` may be a large term.

The Maude Model Checker (VII)

The module MODEL-CHECKER is as follows.

```
fmod MODEL-CHECKER is  protecting QID .  including SATISFACTION .
including LTL .
subsort Prop < Formula .

*** transitions and results
sorts RuleName Transition TransitionList ModelCheckResult .
subsort Qid < RuleName .
subsort Transition < TransitionList .
subsort Bool < ModelCheckResult .
ops unlabeled deadlock : -> RuleName .
op {_,_} : State RuleName -> Transition [ctor] .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList [ctor assoc id: nil] .
op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult [special ( ... )] .
endfm
```

The Maude Model Checker (VIII)

Its key operator is `modelCheck` (whose `special` attribute has been omitted here), which takes a state and an LTL formula and returns either the Boolean `true` if the formula is satisfied, or a counterexample when it is not satisfied.

Let us illustrate the use of this operator with our `MUTEX` example. Following the pattern described above, we can define the module

```
mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a,wait] [b,wait] .
  eq initial2 = * [a,wait] [b,wait] .
endm
```

The Maude Model Checker (X)

We are then ready to model check different LTL properties of `MUTEX`. The first obvious property to check is mutual exclusion:

```
Maude> red modelCheck(initial1,[] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []~ (crit(a) /\ crit(b))) .
rewrites: 18 in 10ms cpu (10ms real) (1800 rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,[] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []~ (crit(a) /\ crit(b))) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The Maude Model Checker (XII)

We can also model check the strong liveness property that if a process waits infinitely often, then it is in its critical section infinitely often:

```
Maude> red modelCheck(initial1,([] <> wait(a)) -> ([] <> crit(a))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(a) -> []<> crit(a)) .
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial1,([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(b) -> []<> crit(b)) .
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial2,([] <> wait(a)) -> ([] <> crit(a))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(a) -> []<> crit(a)) .
rewrites: 68 in 10ms cpu (10ms real) (6800 rewrites/second)
```

```
result Bool: true

Maude> red modelCheck(initial2,([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(b) -> []<> crit(b)) .
rewrites: 68 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The Maude Model Checker (XIII)

Of course, not all properties are true. Therefore, instead of a success we can get a counterexample showing why a property fails. Suppose that we want to check whether, beginning in the state `initial1`, process `b` will always be waiting. We then get the counterexample:

```
Maude> red modelCheck(initial1,[] wait(b)) .
reduce in MUTEX-CHECK : modelCheck(initial1, []wait(b)) .
rewrites: 14 in 10ms cpu (10ms real) (1400 rewrites/second)
result ModelCheckResult:
  counterexample({$ [a,wait] [b,wait],'a-enter}
                 {[a,critical] [b,wait],'a-exit}
                 {* [a,wait] [b,wait],'b-enter},
                 {[a,wait] [b,critical],'b-exit}
                 {$ [a,wait] [b,wait],'a-enter}
                 {[a,critical] [b,wait],'a-exit}
                 {* [a,wait] [b,wait],'b-enter})
```

The Maude Model Checker (XIV)

The main counterexample term constructors are:

```
    op {_,_} : State RuleName -> Transition .
    op nil : -> TransitionList [ctor] .
    op __ : TransitionList TransitionList -> TransitionList [ctor assoc id: nil]
    op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor]
```

A counterexample is a pair consisting of two lists of transitions: the first is a finite path beginning in the initial state, and the second describes a loop. 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. Note that each transition is represented as a pair, consisting of a state and the label of the rule applied to reach the next state.

Model Checking TOK-RING

Consider the following TOK-RING module,

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

(fmod NAT/{N :: NZNAT*} is
   sort Nat/{N} .
   op `[_`] : Nat -> Nat/{N} .
   op _+_ : Nat/{N} Nat/{N} -> Nat/{N} .
   op _*_ : Nat/{N} Nat/{N} -> Nat/{N} .
   vars I J : Nat .
   ceq [I] = [I rem *] if I >= * .
   eq [I] + [J] = [I + J] .
   eq [I] * [J] = [I * J] .
 endfm)
```

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

Model Checking `TOK-RING` (II)

The `TOK-RING` module satisfies the following two properties:

- mutual exclusion, and
- guaranteed reentrance, that is:
  - each process eventually reaches its critical section, and
  - it does so again after $2 \times n$ steps.

There isn't a single LTL formula stating each of these properties: they are parametric on n. However, in Full Maude we can specify these properties by parametic formula definitions as follows:

Model Checking TOK-RING (III)

```
(omod CHECK-TOK-RING{N :: NZNAT*} is
  inc TOK-RING{N} .
  inc MODEL-CHECKER .
  subsort Configuration < State .

  op inCrit : Nat/{N} -> Prop .
  op twoInCrit : -> Prop .

  var  I : Nat .
  vars X Y : Nat/{N} .
  var  C : Configuration .
  var  F : Formula .

  eq < X : Proc | mode : critical > C |= inCrit(X) = true .
  eq < X : Proc | mode : critical > < Y : Proc | mode : critical > C
       |= twoInCrit = true .
```

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

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

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

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

endom)
```

Model Checking `TOK-RING` (IV)

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

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

Then we can model check the mutual exclusion property for 5 processes as follows:

```
(red in CHECK-TOK-RING{5} : modelCheck(init,[] ~ twoInCrit) .)
result Bool :
  true
```

Model Checking `TOK-RING` (V)

In the same way, we can model check the guaranteed reentrance property for $n = 5$ by giving to Full Maude the command,

```
(red in CHECK-TOK-RING(5) : modelCheck(init,[] guaranteedReentrance) .)
result Bool :
  true
```

Simulations

Given two Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$, both having the same set AP of atomic propositions, an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ of $\mathcal{A}$ by $\mathcal{B}$ is a binary relation $H \subseteq A \times B$ such that, denoting pairs $(a, b) \in H$ by aHb, we have:

- if $a \rightarrow_{\mathcal{A}} a'$ and aHb, then there is a $b' \in B$ such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$, and

- $(\forall\, a \in A)(\forall\, b \in B)\ \ aHb \ \ \Rightarrow \ \ L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$.

If the relation H is a function, then we call H an AP-simulation map. If both H and H^{-1} are AP-simulations, then we call H a bisimulation.

Simulations (II)

Note that (exercise) AP-simulations (resp. AP-simulation maps, resp. AP-bisimulations) compose. That is, if we have AP-simulations (resp. AP-simulation maps, resp. AP-bisimulations)

$$H : \mathcal{A} \longrightarrow \mathcal{B} \qquad G : \mathcal{B} \longrightarrow \mathcal{C}$$

then $H;G : \mathcal{A} \longrightarrow \mathcal{C}$ is also an AP-simulation (resp. AP-simulation map, resp. AP-bisimulation).

Note also that the identity function 1_A is trivially an AP-simulation $1_A : \mathcal{A} \longrightarrow \mathcal{A}$, and also an AP-simulation map and an AP-bisimulation.

Simulations Reflect Satisfaction of LTL Formulae

We say that an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ reflects the satisfaction of an LTL formula φ iff $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$.

A fundamental result, allowing us to prove the satisfaction of an LTL formula φ in an infinite-state system $\mathcal{A}$ by proving the same satisfaction in a finite-state system $\mathcal{B}$ that simulates it is the following,

Theorem: AP-simulations always reflect satisfaction of $LTL(AP)$ formulae.

Simulations Reflect Satisfaction of LTL Formulae (II)

Proof: First of all, note that we can extend H to a binary relation $\hat{H} : Path(\mathcal{A}) \longrightarrow Path(\mathcal{B})$, where,

$$\pi \hat{H} \rho \quad \Leftrightarrow \quad \forall\, n \in \mathbb{N} \;\; \pi(n) H \rho(n).$$

Lemma1: If aHb, than for each $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$ there is a $\rho \in Path(\mathcal{B})$ such that $\rho(0) = b$ and $\pi \hat{H} \rho$.

Proof of Lemma1: The proof amounts to an inductive argument that, if we have built the first n stages of ρ, then we can always build the $n+1$ stage. So, assume that we have ρ defined for $0 \leq i \leq n$ with $\rho(0) = b$, and with:

Simulations Reflect Satisfaction of LTL Formulae (III)

- $\pi(i) H \rho(i)$, $0 \leq i \leq n$, and
- $\rho(i) \longrightarrow_{\mathcal{B}} \rho(i+1)$, $0 \leq i < n$.

Then, since H is a simulation, and since $\pi(n) \longrightarrow_{\mathcal{A}} \pi(n+1)$, we can find a $b' \in B$ such that $\rho(n) \longrightarrow_{\mathcal{B}} b'$, and $\pi(n+1) H b'$. Therefore, we can define $\rho(n+1) = b'$ and extend ρ to $n+1$ steps. q.e.d.

We will be essentially done if we prove the following,

Lemma2: For each aHb, $\pi \in Path(\mathcal{A})$ such that $\pi(0) = a$, $\rho \in Path(\mathcal{B})$ such that $\rho(0) = b$ and $\pi \hat{H} \rho$; and for each $\varphi \in LTL(AP)$ we have,

$$\mathcal{B}, b, \rho \models \varphi \quad \Rightarrow \quad \mathcal{A}, a, \pi \models \varphi.$$

Simulations Reflect Satisfaction of LTL Formulae (IV)

Proof of Lemma2: The proof is by structural induction on the structure of LTL formulae. We prove the base case and the case $\varphi = \bigcirc\psi$, and leave the rest as an exercise.

For $p \in AP$ we have, $\mathcal{B}, b, \rho \models p$ iff $p \in L_{\mathcal{B}}(b)$, which by H simulation and aHb is equivalent to $p \in L_{\mathcal{A}}(a)$, which is itself equivalent to, $\mathcal{A}, a, \pi \models \varphi$.

Assume that the result holds for ψ. Let us then show that it holds for $\varphi = \bigcirc\psi$. We have, $\mathcal{B}, b, \rho \models \bigcirc\psi$ iff $\mathcal{B}, \rho(1), s; \rho \models \psi$. But note that if $\pi \hat{H} \rho$, then for any $n \in \mathbb{N}$ we have, $s^n; \pi \hat{H} s^n; \rho$. Therefore, by the induction hypothesis we can conclude that, $\mathcal{A}, \pi(1), s; \pi \models \psi$, which is equivalent to, $\mathcal{A}, a, \pi \models \bigcirc\psi$. q.e.d.

Simulations Reflect Satisfaction of LTL Formulae (V)

We are now essentially done. Suppose $\mathcal{B}, b \models \varphi$ and aHb. Then we have, $\mathcal{B}, b, \rho \models \varphi$ for any $\rho \in Path(\mathcal{B})$ such that $\rho(0) = b$. To prove $\mathcal{A}, a \models \varphi$, we have to show that for each $\pi \in Path(\mathcal{A})$ with $\pi(0) = a$ we have, $\mathcal{A}, a, \pi \models \varphi$. But, by Lemma1, for each such π we have a ρ such that, $\rho(0) = b$, and $\pi \hat{H} \rho$. Then, by Lemma2, we have, $\mathcal{A}, a, \pi \models \varphi$. q.e.d.

We say that an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ preserves the satisfaction of an LTL formula φ iff $\mathcal{A}, a \models \varphi$ and aHb imply $\mathcal{B}, b \models \varphi$.

Corollary: AP-bisimulations always reflect and preserve satisfaction of $LTL(AP)$ formulae.

Abstraction Methods

To prove that a, possibly infinite-state, Kripke structure $\mathcal{A}$ and initial state a satisfy an LTL formula φ, with, say, AP the set of atomic propositions actually appearing in φ, it is enough to find an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ and an initial state $b \in B$ such that aHb and:

- the set of states reachable from b in $\mathcal{B}$ is finite, and
- $\mathcal{B}, b \models \varphi$.

Then, we can model check the property $\mathcal{B}, b \models \varphi$ to prove $\mathcal{A}, a \models \varphi$. Methods to find such an H are called abstraction methods. What follows describes abstraction methods developed in joint work with Narciso Martí-Oliet and Miguel Palomino for systems specified by rewrite theories.

Quotient Abstractions

Let $\mathcal{A} = (A, \to_{\mathcal{A}}, L_{\mathcal{A}})$ be a Kripke structure on AP. We call an equivalence relation $\equiv$ on A label-preserving if $a \equiv a' \Rightarrow L_{\mathcal{A}}(a) = L_{\mathcal{A}}(a')$. We can use a label-preserving equivalence relation $\equiv$ to define a new Kripke structure, $(\mathcal{A}/\equiv) = (A/\equiv, \to_{\mathcal{A}/\equiv}, L_{\mathcal{A}/\equiv})$, where:

- $[a_1] \to_{\mathcal{A}/\equiv} [a_2]$ iff $\exists\, a_1' \in [a_1]\ \exists\, a_2' \in [a_2]$ s.t. $a_1' \to_{\mathcal{A}} a_2'$.
- $L_{\mathcal{A}/\equiv}([a]) = L_{\mathcal{A}}(a)$

It is then trivial to check that the projection map to equivalence classes $q_{\equiv} : a \mapsto [a]$ is an AP-simulation map $q_{\equiv} : \mathcal{A} \longrightarrow \mathcal{A}/\equiv$, which we call the quotient abstraction defined by $\equiv$.

Equational Quotient Abstractions

We are of course particularly interested in abstraction methods for systems specified by rewrite theories. Recall that, given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ plus equations D defining state predicates Π in a kind k of states, we have associated to it the Kripke structure,

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

This Kripke structure may be infinite-state, so that we cannot use an LTL model checker to verify its properties; or it can have a finite state space too big to make model checking feasible. We are therefore interested in defining quotient abstractions of Kripke structures of this kind.

Equational Quotient Abstractions (II)

It is enough to focus on those state predicates actually occurring in a particular formula φ, which we may assume have been defined by the general method described when explaining the Maude LTL model checker.

Given $\mathcal{R} = (\Sigma, E, \phi, R)$, we assume that $(\Sigma', E \cup D)$ protects both (Σ, E) and BOOL. For the atomic predicates $AP_\Pi = \{\theta(p) \mid p \in \Pi,\ \theta \text{ canonical ground substitution}\}$ we then have a labeling function,

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid E \cup D \vdash t \texttt{ |= } \theta(p) = \texttt{true}\}.$$

We are then interested in the Kripke structure, $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi)$.

Equational Quotient Abstractions (III)

Then, a quite general method for defining quotient abstractions of the Kripke structure $\mathcal{K}(\mathcal{R},k)_\Pi = (T_{\Sigma/E,k}, (\to^1_{\mathcal{R}})^\bullet, L_\Pi)$ is to add to E a set G of equations such that $(\Sigma', E \cup G \cup D)$ also protects `BOOL`, and consider the rewrite theory $\mathcal{R}/G = (\Sigma, E \cup G, \phi, R)$, which has an associated Kripke structure $\mathcal{K}(\mathcal{R}/G,k)_\Pi = (T_{\Sigma/E \cup G,k}, (\to^1_{\mathcal{R}/G})^\bullet, L_{\Pi/G})$, with

$$L_{\Pi/G}([t]_{E\cup G}) = \{\theta(p) \in AP_\Pi \mid E \cup G \cup D \vdash t \texttt{|=} \theta(p) = \texttt{true}\}.$$

Note that we have an equivalence relation $\equiv_G$ on $T_{\Sigma/E,k}$, namely,

$$[t]_E \equiv_G [t']_E \quad \Leftrightarrow \quad E \cup G \vdash (\forall \emptyset)\ t = t' \quad \Leftrightarrow \quad [t]_{E\cup G} = [t']_{E\cup G}.$$

Equational Quotient Abstractions (IV)

And of course we have a bijection $T_{\Sigma/E,k}/\equiv_G \cong T_{\Sigma/E\cup G,k}$. The key question now is: under what conditions is the map $q_{\equiv_G} : [t]_E \mapsto [t]_{E\cup G}$ a quotient abstraction of Kripke structures

$$q_{\equiv_G} : \mathcal{K}(\mathcal{R},k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}/G,k)_\Pi$$

If it is so, we will call $\mathcal{K}(\mathcal{R}/G,k)_\Pi = (T_{\Sigma/E\cup G,k}, (\rightarrow^1_{\mathcal{R}/G})^\bullet, L_{\Pi/G})$ the equational quotient abstraction of $\mathcal{K}(\mathcal{R},k)_\Pi$ defined by G.

Of course, to use $\mathcal{R}/G$ as a system module in Maude to model check LTL properties of $\mathcal{R}$, we will also need to require that $\mathcal{R}/G$ satisfies the usual executability conditions.

Equational Quotient Abstractions (IV)

Let k be a kind in a rewrite theory, $\mathcal{R}$ is k-deadlock-free if we have the following identity of binary relations on $T_{\Sigma/E,k}$:

$$(\rightarrow^1_{\mathcal{R}})^{\bullet} = (\rightarrow^1_{\mathcal{R}})$$

Intutively this means that no states in $T_{\Sigma/E,k}$ are deadlock states, so the relation $(\rightarrow^1_{\mathcal{R}})$ is already total.

How restrictive is the requirement that $\mathcal{R}$ is k-deadlock-free? There is no real loss of generality. To a theory $\mathcal{R}$ satisfying the usual executability assumptions and having equational conditions we can always associate a semantically equivalent (from the LTL point of view) theory $\mathcal{R}^k_{d.f.}$ which is k-deadlock-free (see book "All About Maude").

Our main theorem is then the following:

Equational Quotient Abstractions (V)

Theorem 2. Let $\mathcal{R} = (\Sigma, E, \phi, R)$ be a k-deadlock free rewrite theory, and let D be equations defining (possibly parametric) state predicates Π fully defined for all states of kind k as either `true` or `false`, and assume that $(\Sigma', E \cup D)$ protects `BOOL`. Let then G be a set of Σ-equations such that $(\Sigma', E \cup G \cup D)$ also protects `BOOL`. Then the map $q_{\equiv_G} : [t]_E \mapsto [t]_{E \cup G}$ is a quotient abstraction of Kripke structures

$$q_{\equiv_G} : \mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}/G, k)_\Pi.$$

Proof: Since $\mathcal{R}$ is k-deadlock free, it is trivial to check that $\mathcal{R}/G$ is also k-deadlock free and that therefore we have

$$(\rightarrow^1_{\mathcal{R}/G})^\bullet = (\rightarrow^1_{\mathcal{R}/G}) = (\rightarrow^1_{\mathcal{R}})^\bullet / \equiv_G .$$

Equational Quotient Abstractions (VI)

The only remaining thing to check is that $\equiv_G$ is label-preserving. This is equivalent to proving the following equivalences for each $p \in \Pi$ and ground substitution θ:

$$E \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{true} \quad \Leftrightarrow \quad E \cup G \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{true}$$

$$E \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{false} \quad \Leftrightarrow \quad E \cup G \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{false}$$

The $(\Rightarrow)$ follow by monotonicity of equational reasoning. The $(\Leftarrow)$ follow from the protecting `BOOL` assumption, since we can reason by contradiction. Suppose, for example, that $E \cup G \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{true}$ but $E \cup D \vdash t\,\texttt{|=}\,\theta(p) \neq \texttt{true}$. By the protecting `BOOL` assumption this forces $E \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{false}$, which implies $E \cup G \cup D \vdash t\,\texttt{|=}\,\theta(p) = \texttt{false}$, contradicting the protection of `BOOL`. q.e.d.

Executability of Equational Quotient Abstractions

For Theorem 2 to be useful in practice, for example to use Maude to prove LTL such properties of $\mathcal{R}$ by model checking $\mathcal{R}/G$, we need to ensure the following executability requirements:

- $(\Sigma, E \cup G)$ and $(\Sigma', E \cup G \cup D)$ should be ground confluent, sort-decreasing and terminating.
- The rules R should be ground coherent relative to $E \cup G$.

These requirements can be checked using Maude's CRC, MTT, and ChC tools. Similarly, the protecting `BOOL` requirements can be checked using Maude's CRC, MTT, and SCC tools.

Executability of Equational Quotient Abstractions (II)

The checks for executability may be positive or negative. But if, say, the equations $E \cup G$ are not ground confluent, we may be able to complete them to get a semantically equivalent set of equations, say E' which is ground confluent, sort-decreasing, and terminating. Similarly, if the rules R are not ground coherent, we may be able to complete them to get an equivalent set R' of rules that is ground coherent. In this way we would obtain a rewrite theory $\mathcal{R}' = (\Sigma, E', \phi, R')$ semantically equivalent to $\mathcal{R}/G$.

Therefore we would have an isomorphism of a Kripke structures $\mathcal{K}(\mathcal{R}', k)_\Pi \cong \mathcal{K}(\mathcal{R}/G, k)_\Pi$, but now with the crucial propety that $\mathcal{R}'$ is executable, so we can use $\mathcal{R}'$ to verify our desired LTL properties about $\mathcal{R}$.

An Unordered Communication Protocol

Consider a communication channel in which messages can get out of order. There is a sender and a receiver. The sender is sending a sequence of data items, for example numbers. The receiver is supposed to get the sequence in the exact same order in which they were in the sender's sequence.

To achieve this in-order communication in spite of the unordered nature of the channel, the sender sends each data item in a message together with a sequence number; and the receiver sends back an ack indicating that has received the item. The specification in Maude of the protocol is as follows.

An Unordered Communication Protocol (II)

```
mod UNORDERED-CHANNEL is
sorts Natural List Msg Conf State .
subsort Msg < Conf .
op 0 : -> Natural [ctor] .   op s : Natural -> Natural [ctor] .
op nil : -> List [ctor] .
op _;_ : Natural List -> List [ctor] .   *** list constructor
op _@_ : List List -> List .       *** list append
op [_,_] : Natural Natural -> Msg [ctor] .
op ack : Natural -> Msg [ctor] .
op null : -> Conf [ctor] .
op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
op {_,_|_|_,_} : List Natural Conf List Natural -> State [ctor] .
vars N M J K : Natural . vars L P Q : List .  var  C : Conf .
eq nil @ L = L .
eq (N ; L) @ P = N ; (L @ P) .
rl [snd]: {N ; L, M | C | P, K} => {N ; L, M | [N, M] C | P, K} .
rl [rec]: {L, M | [N, J] C | P, J} => {L, M | ack(J) C | P @ (N ; nil), s(J)} .
rl [rec-ack]: {N ; L, J | ack(J) C | P, M}  => {L, s(J) | C | P, M} .
endm
```

An Unordered Communication Protocol (III)

The contents of the unordered channel is modeled as a multiset of messages of sort `Conf`. The entire system state is a 5-tuple of sort `State`, where the components are:

- a buffer with the items to be sent
- a counter for the acknowledged items
- the contents of the unordered channel
- a buffer with the items received, and
- a counter for the items received.

An Unordered Communication Protocol (IV)

We will always assume that all initial states are of the form:

```
{n1 ; ... ; nk ; nil , 0 | null | nil , 0}
```

That is, the sender's buffer contains a list of numbers `n1 ; ... ; nk ; nil` and has the counter set to `0`, the channel is empty, and the receiver's buffer is also empty. Also, the receiver's counter is initially set to `0`.

One essential property that we would like to verify of this protocol is of course that it achieves in-order communication in spite of the unordered channel.

Another property is that the list is eventually received under some fairness assumptions.

An Unordered Communication Protocol (V)

Since, due to the `snd` rule, the set of states reachable from an initial state is infinite, we should model check these properties using an abstraction. We can define the abstraction by adding to the equations of `UNORDERED-CHANNEL` a set G of additional equations defining a quotient of the set of states. We can do so in the following module extending `UNORDERED-CHANNEL` by equations and leaving the rules unchanged:

An Unordered Communication Protocol (VI)

```
mod UNORDERED-CHANNEL-ABSTRACTION is
  including UNORDERED-CHANNEL .
  vars M N P K : Natural .
  vars L L' L'' : List .
  var C : Conf .
  eq [A1]: {L, M | [N, P] [N, P] C | L', K} = {L, M | [N, P] C | L', K} .
endm
```

Three key questions are: (1) is the set of states rechable from an initial state now finite? (2) does this abstraction correspond to a rewrite theory whose equations are ground confluent, sort-decreasing and terminating? (3) are the rules still ground coherent?

Question (1) is clear, since a reachable channel can only contain at most one ack message at any time, and now it cannot contain repeated copies of sent messages.

An Unordered Communication Protocol (VI)

The lecture by Durán has already shown that `UNORDERED-CHANNEL-ABSTRACTION` is not ground coherent, but that it can be made so by adding one extra rule in the module:

```
mod UNORDERED-CHANNEL-ABSTRACTION-2 is
extending UNORDERED-CHANNEL-ABSTRACTION .
vars M N P K : Natural . vars L L L : List . var C : Conf .
rl [snd2]: {L, M | [N, K] C | L, K}
    => {L, M | [N, K] ack(K) C | L @ N ; nil, s(K)}.
endm
```

Furthermore, Duran's lecture showed that `UNORDERED-CHANNEL-ABSTRACTION-2` is confluent, sort-decreasing, terminating, and ground coherent.

An Unordered Communication Protocol (VII)

What about state predicates? Are they preserved by the abstraction? In order to specify the desired safety and liveness properties, it is enough to specify in Maude the following state predicates:

An Unordered Communication Protocol (VIII)

```
mod UNORDERED-CHANNEL-PREDS is pr UNORDERED-CHANNEL . inc TRUTH-VALUE .
 sort Proposition .
 op _~_ : Natural Natural -> Bool .  op _~_ : List List -> Bool .
 op _/\_ : Bool Bool -> Bool [assoc comm id: true] . *** conjunction
 op _|=_ : State Proposition -> Bool [frozen] .
 vars M N K P : Natural . vars L L' L'' : List . var  C : Conf . var  B : Bool .
 eq 0 ~ 0 = true .  eq 0 ~ s(N) = false .  eq s(N) ~ 0 = false .
 eq s(N) ~ s(M) = N ~ M .
 eq nil ~ nil = true .  eq nil ~ N ; L = false .  eq N ; L ~ nil = false .
 eq N ; L ~ M ; L' = (N ~ M) /\ (L ~ L') .
 eq false /\ false = false .
 ops prefix rec-q : List -> Proposition [ctor] .
 eq [I1]:  {L', N | C | K ; L'', P} |= prefix(M ; L)
    = (M ~ K) /\  {L', N | C | L'', P} |=  prefix(L) .
 eq [I3]: {L', N | C | nil, K} |=  prefix(L) = true .
 eq [I4]: {L', N | C | M ; L'', K} |=  prefix(nil) = false .
 eq [I5]: {L', N | C | L'', K} |=  rec-q(L) = L ~ L'' .
endm
```

These predicates are then imported without change, together with UNORDERED-CHANNEL-ABSTRACTION-2, in the module:

```
mod UNORDERED-CHANNEL-ABSTRACTION-2-PREDS is
   including UNORDERED-CHANNEL-PREDS .
   including UNORDERED-CHANNEL-ABSTRACTION-2 .
endm
```

By the second part of the proof of Theorem 2 (which does not use the deadlock-freedom assumption), the preservation of these state predicates can be guaranteed if we show that both UNORDERED-CHANNEL-PREDS and UNORDERED-CHANNEL-ABSTRACTION-2-PREDS protect TRUTH-VALUE. This follows from the absence of any equations having `true` or `false` in their lefthand sides plus the following facts, all of which are checked by Maude tools:

1. both `UNORDERED-CHANNEL-PREDS` and `UNORDERED-CHANNEL-ABSTRACTION-2-PREDS` are sufficiently complete;

2. both `UNORDERED-CHANNEL-PREDS` and `UNORDERED-CHANNEL-ABSTRACTION-2-PREDS` are locally confluent and sort-decreasing;

3. both `UNORDERED-CHANNEL-PREDS` and `UNORDERED-CHANNEL-ABSTRACTION-2-PREDS` are terminating.

The SCC tool checks fact (1):

```
Checking sufficient completeness of UNORDERED-CHANNEL-PREDS ...
Success: UNORDERED-CHANNEL-PREDS is sufficiently complete under the assumption
    that it is ground weakly-normalizing, confluent, and ground
    sort-decreasing.

Checking sufficient completeness of UNORDERED-CHANNEL-ABSTRACTION-2-PREDS ...
Warning: This module has equations that are not left-linear which will be
    ignored when checking.
Success: UNORDERED-CHANNEL-ABSTRACTION-2-PREDS is sufficiently complete under th
    assumption that it is ground weakly-normalizing, confluent, and ground
    sort-decreasing.
```

The CRC tool checks fact (2):

```
Church-Rosser checking of UNORDERED-CHANNEL-PREDS
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.

Church-Rosser checking of UNORDERED-CHANNEL-ABSTRACTION-2-PREDS
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

All we have left is checking termination of the equations in `UNORDERED-CHANNEL-PREDS` and `UNORDERED-CHANNEL-ABSTRACTION-PREDS` that is, fact (3). But since the equations in `UNORDERED-CHANNEL-ABSTRACTION-PREDS` are precisely the union of those in `UNORDERED-CHANNEL-ABSTRACTION` and `UNORDERED-CHANNEL-PREDS`, it is enough to check that `UNORDERED-CHANNEL-ABSTRACTION-PREDS` is terminating. This check succeeds with the Maude Termination Tool (MTT).

This finishes all the checks of correctness and executability. The only remaining issue is deadlock freedom, which is required for the correctness of the abstraction. To ensure deadlock freedom we can perform the automatic module transformation described in Section 15.3 of the Maude Book, that preserves all the desired executability properties to obtain a semantically equivalent, deadlock-free version.

However, since in this example a state in `UNORDERED-CHANNEL` is a deadock state iff its canonical form in `UNORDERED-CHANNEL-ABSTRACTION` is a deadlock state, in this case we do not need to perform this extra transformation.

We can now verify our safety property of in-order-communication as follows:

```
mod UNORDERED-CHANNEL-ABSTRACTION-CHECK is
    extending UNORDERED-CHANNEL-ABSTRACTION-2 .
    including UNORDERED-CHANNEL-PREDS .
    extending MODEL-CHECKER .
    subsort Proposition < Prop .
    op init : -> State .
    eq init = {0 ; s(0) ; s(s(0)) ; nil , 0 | null | nil , 0} .
endm

reduce in UNORDERED-CHANNEL-ABSTRACTION-CHECK :
     modelCheck(init, []prefix(0 ; s(0) ; s(s(0)) ; nil)) .
rewrites: 361 in 41ms cpu (42ms real) (8780 rewrites/second)
result Bool: true
```

The Need for the Temporal Logic of Rewriting

The other important property we would like to verify about the unordered channel is the liveness property that the list initially in the sender buffer eventually gets delivered in its entirety to the receiver buffer.

It is obvious that this property does not hold without some extra assumptions, because one possible behavior is just for the `snd` rule to keep resending a given element forever. So, if L is the list in the sender buffer of the initial state, the property we would like to verify is the following:

$$(\Diamond\ rec-q(L)) \vee (\Diamond\Box\ snd)$$

The Need for the Temporal Logic of Rewriting (II)

That is, either the entire list eventually arrives to the receiver buffer, or after some time, all the protocol does is to resend the same data item forever. Since this second behavior is totally unreasonable, we then will have verified that, under very reasonable assumptions, the protocol terminates with the expected result.

The problem, however, is that `snd` is not a state predicate but a rule label. Furthermore, without modifying the protocol there is no way to specify the `snd` action as a state predicate.

The Need for the Temporal Logic of Rewriting (III)

This is where the Linear Temporal Logic of Rewriting (LTLR) comes in: it adds to the standard linear temporal logic (LTL) action patterns, including simple patterns such as rule labels, and more complex spatial action patterns. Its semantics is interpreted not just over sequences of states but about computations κ of the form

$$t_0 \xrightarrow{\gamma_0} t_1 \xrightarrow{\gamma_1} t_2 \xrightarrow{\gamma_2} t_3 \ldots \xrightarrow{\gamma_n} t_{n+1} \ldots$$

where the γ_i are proofs of one-step rewrites. In particular, for l a rule label, its semantics for a rewrite theory $\mathcal{R}$ is defined for the above computation κ by: $\mathcal{R}, \kappa \models l$ iff the rule applied in γ_0 has label l.

Verification in the Maude LTLR Model Checker

Our liveness property for the unordered channel cannot be verified in LTL without modifying the protocol specification. However, it can be verified in Maude's LTLR Model Checker, developed by Kyungmin Bae, as follows:

```
Maude> red
modelCheck(init, (<> rec-q(0 ; s(0) ; s(s(0)) ; nil)) \/ (<>[] {'snd}) ) .
result Bool: true
```