Program Verification: Lecture 23

José Meseguer

University of Illinois at Urbana-Champaign

Decidability of Propositional LTL

It is well-known that, for any computable Kripke structure $\mathcal{Q} = (Q, \rightarrow_{\mathcal{Q}}, {}_{\mathcal{Q}})$ on state predicates Π, any state $q \in Q$ such that the set of states reachable from q in $\mathcal{Q}$ is finite, and any LTL formula $\varphi \in LTL(\Pi)$ there is a decision procedure that can effectively decide the satisfaction relation

$$\mathcal{Q}, q \models_{LTL} \varphi.$$

Decidability of Propositional LTL

It is well-known that, for any computable Kripke structure $\mathcal{Q} = (Q, \rightarrow_{\mathcal{Q}}, _{\mathcal{Q}})$ on state predicates Π, any state $q \in Q$ such that the set of states reachable from q in $\mathcal{Q}$ is finite, and any LTL formula $\varphi \in LTL(\Pi)$ there is a decision procedure that can effectively decide the satisfaction relation

$$\mathcal{Q}, q \models_{LTL} \varphi.$$

Furthermore, if $\mathcal{Q}, q \not\models_{LTL} \varphi$, the decision procedure will exhibit a counterexample, that is, a path $\pi \in Path(\mathcal{Q}^{\bullet})_q$ violating φ.

Decidability of Propositional LTL

It is well-known that, for any computable Kripke structure $\mathcal{Q} = (Q, \rightarrow_{\mathcal{Q}}, _{\mathcal{Q}})$ on state predicates Π, any state $q \in Q$ such that the set of states reachable from q in $\mathcal{Q}$ is finite, and any LTL formula $\varphi \in LTL(\Pi)$ there is a decision procedure that can effectively decide the satisfaction relation

$$\mathcal{Q}, q \models_{LTL} \varphi.$$

Furthermore, if $\mathcal{Q}, q \not\models_{LTL} \varphi$, the decision procedure will exhibit a counterexample, that is, a path $\pi \in Path(\mathcal{Q}^{\bullet})_q$ violating φ.

Since in $LTL(\Pi)^+$ we have $\mathcal{Q}, q \not\models_{LTL} \varphi$ iff $\mathcal{Q}, q \models_{LTL^+} \mathbf{E}\neg\varphi$, the counterxample path is a constructive proof of $\mathcal{Q}, q \models_{LTL^+} \mathbf{E}\neg\varphi$.

Decidability of Propositional LTL

It is well-known that, for any computable Kripke structure $\mathcal{Q} = (Q, \to_{\mathcal{Q}}, _{\mathcal{Q}})$ on state predicates Π, any state $q \in Q$ such that the set of states reachable from q in $\mathcal{Q}$ is finite, and any LTL formula $\varphi \in LTL(\Pi)$ there is a decision procedure that can effectively decide the satisfaction relation

$$\mathcal{Q}, q \models_{LTL} \varphi.$$

Furthermore, if $\mathcal{Q}, q \not\models_{LTL} \varphi$, the decision procedure will exhibit a counterexample, that is, a path $\pi \in Path(\mathcal{Q}^{\bullet})_q$ violating φ.

Since in $LTL(\Pi)^+$ we have $\mathcal{Q}, q \not\models_{LTL} \varphi$ iff $\mathcal{Q}, q \models_{LTL^+} \mathbf{E}\neg\varphi$, the counterxample path is a constructive proof of $\mathcal{Q}, q \models_{LTL^+} \mathbf{E}\neg\varphi$.

Therefore, we can prove a desired **E**-property $\mathcal{Q}, q \models_{LTL^+} \mathbf{E}\,\psi$ precisely by getting a counterexample disproving $\mathcal{Q}, q \models_{LTL} \neg\psi$.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \to \mathcal{P}(\Pi)]$ denotes a set of infinite words.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \rightarrow \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \rightarrow \mathcal{P}(\Pi)]$ denotes a set of infinite words. The regular languages we need are subsets $L \subseteq \mathcal{P}(\Pi)^\omega$ called ω-regular languages.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \to \mathcal{P}(\Pi)]$ denotes a set of infinite words. The regular languages we need are subsets $L \subseteq \mathcal{P}(\Pi)^\omega$ called ω-regular languages. They are recognized by finite automata, called here Büchi automata, and are closed under all Boolean operations.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \to \mathcal{P}(\Pi)]$ denotes a set of infinite words. The regular languages we need are subsets $L \subseteq \mathcal{P}(\Pi)^\omega$ called ω-regular languages. They are recognized by finite automata, called here Büchi automata, and are closed under all Boolean operations. Furthermore, it is decidable wether any such $L \subseteq \mathcal{P}(\Pi)^\omega$ is empty.

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \to \mathcal{P}(\Pi)]$ denotes a set of infinite words. The regular languages we need are subsets $L \subseteq \mathcal{P}(\Pi)^\omega$ called ω-regular languages. They are recognized by finite automata, called here Büchi automata, and are closed under all Boolean operations. Furthermore, it is decidable wether any such $L \subseteq \mathcal{P}(\Pi)^\omega$ is empty.

The two key crucial facts are:

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \to \mathcal{P}(\Pi)]$ denotes a set of infinite words. The regular languages we need are subsets $L \subseteq \mathcal{P}(\Pi)^\omega$ called ω-regular languages. They are recognized by finite automata, called here Büchi automata, and are closed under all Boolean operations. Furthermore, it is decidable wether any such $L \subseteq \mathcal{P}(\Pi)^\omega$ is empty.

The two key crucial facts are: (1) $\forall \varphi \in LTL(\Pi)$, the language $L_\varphi =_{def} \{\tau \in \mathcal{P}(\Pi)^\omega \mid \tau \models_{LTL} \varphi\}$ is ω-regular, and

Decidability of Propositional LTL (II)

The procedure to decide whether $\mathcal{Q}, q \models_{LTL} \varphi$ holds is called a model checking algorithm. As explained in the Appendix, the problem can be reduced to a decidable emptiness check for regular languages, where a trace $\tau \in [\mathbb{N} \to \mathcal{P}(\Pi)]$ is viewed as an infinite word in the alphabet $\mathcal{P}(\Pi)$. Just as $\mathcal{P}(\Pi)^*$ denotes a set of finite words, $\mathcal{P}(\Pi)^\omega =_{def} [\mathbb{N} \to \mathcal{P}(\Pi)]$ denotes a set of infinite words. The regular languages we need are subsets $L \subseteq \mathcal{P}(\Pi)^\omega$ called ω-regular languages. They are recognized by finite automata, called here Büchi automata, and are closed under all Boolean operations. Furthermore, it is decidable wether any such $L \subseteq \mathcal{P}(\Pi)^\omega$ is empty.

The two key crucial facts are: (1) $\forall \varphi \in LTL(\Pi)$, the language $L_\varphi =_{def} \{\tau \in \mathcal{P}(\Pi)^\omega \mid \tau \models_{LTL} \varphi\}$ is ω-regular, and (2) $\mathcal{Q}, q \models_{LTL} \varphi$ iff $Tr(\mathcal{Q}^\bullet)_q =_{def} \{\pi; preds \mid \pi \in Path(\mathcal{Q}^\bullet)_q\} \subseteq L_\varphi$.

The Maude Model Checker

Lecture 22 explained how, given an admissible system module `M` with rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, we can equationally define (possibly parametric) state predicates Π in an extended module `M-PREDS`, thus defining the Kripke structure $\mathbb{C}^{\Pi}_{\mathcal{R}}$.

The Maude Model Checker

Lecture 22 explained how, given an admissible system module `M` with rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, we can equationally define (possibly parametric) state predicates Π in an extended module `M-PREDS`, thus defining the Kripke structure $\mathbb{C}^{\Pi}_{\mathcal{R}}$.

Given a ground LTL formula $\varphi \in LTL(\Pi)$ and an initial state $[u] \in C_{\Sigma/E \cup B, State}$ having a finite set of reachable states, we can decide the satisfaction relation $\mathbb{C}^{\Pi}_{\mathcal{R}}, [u] \models \varphi$ by applying the general LTL decidability result to the Kripke structure $\mathbb{C}^{\Pi}_{\mathcal{R}}$.

The Maude Model Checker

Lecture 22 explained how, given an admissible system module `M` with rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, we can equationally define (possibly parametric) state predicates Π in an extended module `M-PREDS`, thus defining the Kripke structure $\mathbb{C}^{\Pi}_{\mathcal{R}}$.

Given a ground LTL formula $\varphi \in LTL(\Pi)$ and an initial state $[u] \in C_{\Sigma/E \cup B, State}$ having a finite set of reachable states, we can decide the satisfaction relation $\mathbb{C}^{\Pi}_{\mathcal{R}}, [u] \models \varphi$ by applying the general LTL decidability result to the Kripke structure $\mathbb{C}^{\Pi}_{\mathcal{R}}$.

Maude uses an on-the-fly LTL model checking procedure that performs the ω-regular language operations (see page 3 above and further details in the Appendix).

The Maude Model Checker

Lecture 22 explained how, given an admissible system module M with rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, we can equationally define (possibly parametric) state predicates Π in an extended module M-PREDS, thus defining the Kripke structure $\mathbb{C}_{\mathcal{R}}^{\Pi}$.

Given a ground LTL formula $\varphi \in LTL(\Pi)$ and an initial state $[u] \in C_{\Sigma/E \cup B, State}$ having a finite set of reachable states, we can decide the satisfaction relation $\mathbb{C}_{\mathcal{R}}^{\Pi}, [u] \models \varphi$ by applying the general LTL decidability result to the Kripke structure $\mathbb{C}_{\mathcal{R}}^{\Pi}$.

Maude uses an on-the-fly LTL model checking procedure that performs the ω-regular language operations (see page 3 above and further details in the Appendix). Specifically, a procedure of the kind described in §9.5 of Clarke, Grumberg, and Peled's *Model Checking*, MIT Press, 2001, that I sketch in what follows.

The Maude Model Checker (II)

The basis of this procedure (further explained in the Appendix) is the following. Each *LTL* formula φ has an associated Büchi automaton $\mathbf{B}_\varphi$ whose acceptance ω-language is exactly that the set of traces satisfying φ.

The Maude Model Checker (II)

The basis of this procedure (further explained in the Appendix) is the following. Each *LTL* formula φ has an associated Büchi automaton $\mathbf{B}_\varphi$ whose acceptance ω-language is exactly that the set of traces satisfying φ. We can then reduce the satisfaction problem

$$\mathbb{C}^{\Pi}_{\mathcal{R}}, [u] \models \varphi$$

The Maude Model Checker (II)

The basis of this procedure (further explained in the Appendix) is the following. Each *LTL* formula φ has an associated Büchi automaton $\mathbf{B}_{\varphi}$ whose acceptance ω-language is exactly that the set of traces satisfying φ. We can then reduce the satisfaction problem

$$\mathbb{C}^{\Pi}_{\mathcal{R}}, [u] \models \varphi$$

to the emptiness problem of the language accepted by the synchronous product of $\mathbf{B}_{\neg\varphi}$ with (the Büchi automaton $\mathbf{B}(\mathbb{C}^{\Pi\bullet}_{\mathcal{R}}, [u])$ associated to) $\mathbb{C}^{\Pi}_{\mathcal{R}}, [u]$.

The Maude Model Checker (II)

The basis of this procedure (further explained in the Appendix) is the following. Each *LTL* formula φ has an associated Büchi automaton $\mathbf{B}_{\varphi}$ whose acceptance ω-language is exactly that the set of traces satisfying φ. We can then reduce the satisfaction problem

$$\mathbb{C}^{\Pi}_{\mathcal{R}}, [u] \models \varphi$$

to the emptiness problem of the language accepted by the synchronous product of $\mathbf{B}_{\neg\varphi}$ with (the Büchi automaton $\mathbf{B}(\mathbb{C}^{\Pi\bullet}_{\mathcal{R}}, [u])$ associated to) $\mathbb{C}^{\Pi}_{\mathcal{R}}, [u]$. The formula φ is satisfied iff such a language is empty.

The Maude Model Checker (II)

The basis of this procedure (further explained in the Appendix) is the following. Each *LTL* formula φ has an associated Büchi automaton $\mathbf{B}_{\varphi}$ whose acceptance ω-language is exactly that the set of traces satisfying φ. We can then reduce the satisfaction problem

$$\mathbb{C}^{\Pi}_{\mathcal{R}}, [u] \models \varphi$$

to the emptiness problem of the language accepted by the synchronous product of $\mathbf{B}_{\neg\varphi}$ with (the Büchi automaton $\mathbf{B}(\mathbb{C}^{\Pi\bullet}_{\mathcal{R}}, [u])$ associated to) $\mathbb{C}^{\Pi}_{\mathcal{R}}, [u]$. The formula φ is satisfied iff such a language is empty. The model checking procedure checks emptiness by searching for a counterexample, that is, for an infinite path π in $\mathbb{C}^{\Pi\bullet}_{\mathcal{R}}$ from $[u]$ generating a trace τ in the language recognized by the synchronous product $\mathbf{B}(\mathbb{C}^{\Pi\bullet}_{\mathcal{R}}, [u]) \otimes \mathbf{B}_{\neg\varphi}$, i.e., a trace of $\mathbb{C}^{\Pi\bullet}_{\mathcal{R}}$ from $[u]$ such that $\tau \in L_{\neg\varphi}$.

The Maude Model Checker (III)

How do we then model check a given LTL formula φ in Maude for a given initial state $[u]$ in an admissible system module `M` whose state predicates Π have been specified in `M-PREDS`?

The Maude Model Checker (III)

How do we then model check a given LTL formula φ in Maude for a given initial state $[u]$ in an admissible system module `M` whose state predicates Π have been specified in `M-PREDS`? We define a new module, say `M-CHECK`, according to the following pattern:

The Maude Model Checker (III)

How do we then model check a given LTL formula φ in Maude for a given initial state $[u]$ in an admissible system module `M` whose state predicates Π have been specified in `M-PREDS`? 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 : -> State .       *** optional
  eq init = u .              *** optional
endm
```

The Maude Model Checker (III)

How do we then model check a given LTL formula φ in Maude for a given initial state $[u]$ in an admissible system module `M` whose state predicates Π have been specified in `M-PREDS`? 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 : -> State .       *** optional
  eq init = u .              *** optional
endm
```

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

The Maude Model Checker (III)

How do we then model check a given LTL formula φ in Maude for a given initial state $[u]$ in an admissible system module `M` whose state predicates Π have been specified in `M-PREDS`? 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 : -> State .        *** optional
  eq init = u .               *** optional
endm
```

The declaration of `init` is not necessary: it is a matter of convenience, since the initial state `u` may be a large term. Including the module `LTL-SIMPLIFIER` is also optional. Its purpose is to simplify the formula $\neg\varphi$ to generate a smaller Büchi automaton $\mathbf{B}_{\neg\varphi}$, since $|\mathbf{B}_{\neg\varphi}|$ is exponential in $|\varphi|$.

The LTL Module

MODEL-CHECKER imports the following LTL functional module (in the file model-checker.maude) providing syntax for LTL formulas:

The LTL Module

MODEL-CHECKER imports the following LTL functional module (in the file model-checker.maude) providing syntax for LTL formulas:

```
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)] .
```

The LTL Module (II)

```
*** 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) .
```

The LTL Module (III)

```
 *** 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 (III)

```
 *** 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 set Π of state predicates is not specified. This happens in `MODEL-CHECKER` through declaration: `subsort Prop < Formula` and the importation of `M-PREDS`, where Π is equationally specified.

The LTL Module (III)

```
 *** 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 set Π of state predicates is not specified. This happens in `MODEL-CHECKER` through declaration: `subsort Prop < Formula` and the importation of `M-PREDS`, where Π is equationally specified.

Since, for model checking, LTL formulas are put in negative normal form, we also need as constructors the duals of the basic constructor connectives $\top$, $\bigcirc$, $\mathcal{U}$, and $\vee$, i.e., the dual connectives: $\bot$, $\mathcal{R}$, and $\wedge$ ($\bigcirc$ is self-dual).

The Maude Model Checker (IV)

The module `MODEL-CHECKER` is as follows.

The Maude Model Checker (IV)

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
```

A MUTEX Example

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

A MUTEX Example

Its key operator is `modelCheck` (whose `special` attribute has been omitted here), which takes an initial 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 MODEL-CHECKER with a very simple MUTEX mutual exclusion protocol with two processes `a` and `b`.

A MUTEX Example

Its key operator is modelCheck (whose special attribute has been omitted here), which takes an initial 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 MODEL-CHECKER with a very simple MUTEX mutual exclusion protocol with two processes a and b.

```
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
```

A MUTEX Example (II)

```
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

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
```

A MUTEX Example (III)

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

A MUTEX Example (III)

We are now 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))) .

result Bool: true

Maude> red modelCheck(initial2,[] ~(crit(a) /\ crit(b))) .

result Bool: true
```

A MUTEX Example (IV)

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

A MUTEX Example (IV)

We can also model check the strong fairness property (a kind of 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))) .

result Bool: true

Maude> red modelCheck(initial1,([] <> wait(b)) -> ([] <> crit(b))) .

result Bool: true

Maude> red modelCheck(initial2,([] <> wait(a)) -> ([] <> crit(a))) .

result Bool: true

Maude> red modelCheck(initial2,([] <> wait(b)) -> ([] <> crit(b))) .

result Bool: true
```

A `MUTEX` Example (V)

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:

A MUTEX Example (V)

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)) .

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 (V)

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] .
```

The Maude Model Checker (V)

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.

The Maude Model Checker (V)

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-state 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 Maude Model Checker (V)

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-state 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.

COMM Revisited

In Lecture 22 we defined equationally the state predicates used in formalizing the requirements for successful communication between a sender A and a receiver B as a parametric formula.

COMM Revisited

In Lecture 22 we defined equationally the state predicates used in formalizing the requirements for successful communication between a sender `A` and a receiver `B` as a parametric formula. Now we can verify it for any initial state with a list of data to be sent:

COMM Revisited

In Lecture 22 we defined equationally the state predicates used in formalizing the requirements for successful communication between a sender A and a receiver B as a parametric formula. Now we can verify it for any initial state with a list of data to be sent:

```
omod COMM-CHECK is
  protecting COMM-PREDS .
  inc MODEL-CHECKER .

  vars A B : Oid .  var L : List .

   op success-comm : Oid Oid List -> Formula .

   eq success-comm(A,B,L) =
   <> ((~ enabled) /\ no-msgs /\ holds(B,L) /\ holds(A,nil) /\
       (~ waits-ack(A)) /\ cnt(A,| L |) /\ cnt(B,| L |)) .
endom

red modelCheck(init('a,'b,1 ; 2 ; 3),success-comm('a,'b,1 ; 2 ; 3)) .

result Bool: true
```

COMM Revisited (II)

Usually, by a deadlock we mean an unwanted terminating state.

COMM Revisited (II)

Usually, by a deadlock we mean an unwanted terminating state. For example, the final state guaranteed by the `success-comm` formula is a wanted terminating state and therefore not a deadlock in this sense.

COMM Revisited (II)

Usually, by a deadlock we mean an unwanted terminating state. For example, the final state guaranteed by the `success-comm` formula is a wanted terminating state and therefore not a deadlock in this sense. So we can also ask: Are there any deadlocks in COMM? The LTL formula asserting that there are none is remarkably simple:

COMM Revisited (II)

Usually, by a deadlock we mean an unwanted terminating state. For example, the final state guaranteed by the success-comm formula is a wanted terminating state and therefore not a deadlock in this sense. So we can also ask: Are there any deadlocks in COMM? The LTL formula asserting that there are none is remarkably simple:

```
red modelCheck(init('a,'b,nil),((~ enabled) => success-comm('a,'b,nil))) .

result Bool: true

red modelCheck(init('a,'b,1 ; 2 ; 3),((~ enabled) =>
                                         success-comm('a,'b,1 ; 2 ; 3))) .
result Bool: true

red modelCheck(init('a,'b,1 ; 2 ; 3 ; 4 ; 5),((~ enabled) =>
                                 success-comm('a,'b,1 ; 2 ; 3 ; 4 ; 5))) .
result Bool: true
```

COMM Revisited (III)

We can try to ask an answer a stronger question about COMM.

COMM Revisited (III)

We can try to ask an answer a stronger question about `COMM`. Given a parametric initial state `init(A,B,L)` the `success-comm(A,B,L)` property ensures that `L` is received correctly. But is the order of `L` preserved in all intermediate states of the computation?

COMM Revisited (III)

We can try to ask an answer a stronger question about COMM. Given a parametric initial state `init(A,B,L)` the `success-comm(A,B,L)` property ensures that L is received correctly. But is the order of L preserved in all intermediate states of the computation?

This question is interesting because it requires us to:

COMM Revisited (III)

We can try to ask an answer a stronger question about `COMM`. Given a parametric initial state `init(A,B,L)` the `success-comm(A,B,L)` property ensures that `L` is received correctly. But is the order of `L` preserved in all intermediate states of the computation?

This question is interesting because it requires us to:

1. Think carefully about `COMM` to see how we can specify those intermediate states as a disjunction of constrained constructor patterns, and therefore as a (parametric) state predicate.

COMM Revisited (III)

We can try to ask an answer a stronger question about `COMM`. Given a parametric initial state `init(A,B,L)` the `success-comm(A,B,L)` property ensures that `L` is received correctly. But is the order of `L` preserved in all intermediate states of the computation?

This question is interesting because it requires us to:

1. Think carefully about `COMM` to see how we can specify those intermediate states as a disjunction of constrained constructor patterns, and therefore as a (parametric) state predicate.
2. Once we have done so, verify that this conjectured set of intermediate states is an invariant from `init(A,B,L)`.

COMM Revisited (III)

We can try to ask an answer a stronger question about COMM. Given a parametric initial state init(A,B,L) the success-comm(A,B,L) property ensures that L is received correctly. But is the order of L preserved in all intermediate states of the computation?

This question is interesting because it requires us to:

1. Think carefully about COMM to see how we can specify those intermediate states as a disjunction of constrained constructor patterns, and therefore as a (parametric) state predicate.
2. Once we have done so, verify that this conjectured set of intermediate states is an invariant from init(A,B,L).

Part (1) of the question can be answered by adding to COMM-PREDS the following parametric state predicate and its defining equations:

COMM Revisited (IV)

```
*** parametric predicate: in-order-comm

op in-order-comm : Oid Oid List -> Prop [ctor] .

ceq < A : Sender | buff : L2, rec : B, cnt : M, ack-w : false >
    < B : Receiver | buff : L1, snd : A, cnt : M >
  |= in-order-comm(A,B,L) = true if L = L1 ; L2 /\ M = | L1 | .

ceq < A : Sender | buff : L2, rec : B, cnt : M, ack-w : true >
    (to B from A val N cnt M)
    < B : Receiver | buff : L1, snd : A, cnt : M >
  |= in-order-comm(A,B,L) = true if L = L1 ; N ; L2 /\ | L1 | = M .

ceq < A : Sender | buff : L2, rec : B, cnt : M, ack-w : true >
    (to A from B ack M)
    < B : Receiver | buff : (L1 ; N), snd : A, cnt : s(M) >
  |= in-order-comm(A,B,L) = true if L = L1 ; N ; L2 /\ | L1 | = M .
```

COMM Revisited (IV)

```
*** parametric predicate: in-order-comm

op in-order-comm : Oid Oid List -> Prop [ctor] .

ceq < A : Sender | buff : L2, rec : B, cnt : M, ack-w : false >
    < B : Receiver | buff : L1, snd : A, cnt : M >
  |= in-order-comm(A,B,L) = true if L = L1 ; L2 /\ M = | L1 | .

ceq < A : Sender | buff : L2, rec : B, cnt : M, ack-w : true >
    (to B from A val N cnt M)
    < B : Receiver | buff : L1, snd : A, cnt : M >
  |= in-order-comm(A,B,L) = true if L = L1 ; N ; L2 /\ | L1 | = M .

ceq < A : Sender | buff : L2, rec : B, cnt : M, ack-w : true >
    (to A from B ack M)
    < B : Receiver | buff : (L1 ; N), snd : A, cnt : s(M) >
  |= in-order-comm(A,B,L) = true if L = L1 ; N ; L2 /\ | L1 | = M .
```

Note that, as explained in Lecture 22, each conditional equation uses each of the constrained constructor patterns in the disjunction to define the `in-order-comm` state predicate.

`COMM` Revisited (V)

We can now answer Part (2) of the question by giving, for various instances of the parametric initial state `init(A,B,L)`, the model checking commands:

COMM Revisited (V)

We can now answer Part (2) of the question by giving, for various instances of the parametric initial state `init(A,B,L)`, the model checking commands:

```
red modelCheck(init('a,'b,nil),[] in-order-comm('a,'b,nil)) .

result Bool: true

red modelCheck(init('a,'b,1 ; 2 ; 3),[] in-order-comm('a,'b,1 ; 2 ; 3)) .

result Bool: true

red modelCheck(init('a,'b,1 ; 2 ; 3 ; 4 ; 5),
                             [] in-order-comm('a,'b,1 ; 2 ; 3 ; 4 ; 5)) .

result Bool: true
```

COMM Revisited (VI)

As a last example, we can use COMM to illustrate how we can verify $LTL(\Pi)^{+}$ formulas $\mathbf{E}\,\varphi$ by model checking $\neg\varphi$ and getting a counterexample as a proof of $\mathbf{E}\,\varphi$.

COMM Revisited (VI)

As a last example, we can use `COMM` to illustrate how we can verify $LTL(\Pi)^+$ formulas $\mathbf{E}\,\varphi$ by model checking $\neg\varphi$ and getting a counterexample as a proof of $\mathbf{E}\,\varphi$.

The point is that $LTL(\Pi)^+$ allows us to ask useful questions regarding possible relations between reachable states not expressible in $LTL(\Pi)$. For example, we can ask:

COMM Revisited (VI)

As a last example, we can use COMM to illustrate how we can verify $LTL(\Pi)^+$ formulas $\mathbf{E}\,\varphi$ by model checking $\neg\varphi$ and getting a counterexample as a proof of $\mathbf{E}\,\varphi$.

The point is that $LTL(\Pi)^+$ allows us to ask useful questions regarding possible relations between reachable states not expressible in $LTL(\Pi)$. For example, we can ask:

Are there states reachable from init(A,B,L) such that the counters of A and B hold different values?

COMM Revisited (VI)

As a last example, we can use `COMM` to illustrate how we can verify $LTL(\Pi)^+$ formulas $\mathbf{E}\,\varphi$ by model checking $\neg\varphi$ and getting a counterexample as a proof of $\mathbf{E}\,\varphi$.

The point is that $LTL(\Pi)^+$ allows us to ask useful questions regarding possible relations between reachable states not expressible in $LTL(\Pi)$. For example, we can ask:

Are there states reachable from `init(A,B,L)` such that the counters of `A` and `B` hold different values?

We can express the negation $\neg\varphi$ of this property by adding to `CHECK-PREDS` the following parametric predicate definition:

COMM Revisited (VII)

```
op same-cnts : Oid Oid -> Prop .

eq < A : Sender | buff : L2, rec : B, cnt : N, ack-w : TV >
   < B : Receiver | buff : L1, snd : A, cnt : N > C |= same-cnts(A,B) = true .
```

COMM Revisited (VII)

```
op same-cnts : Oid Oid -> Prop .

eq < A : Sender | buff : L2, rec : B, cnt : N, ack-w : TV >
   < B : Receiver | buff : L1, snd : A, cnt : N > C |= same-cnts(A,B) = true .
```

Now we can ask and answer the original question $\mathbf{E}\,\varphi(A, B)$, i.e.,

COMM Revisited (VII)

```
op same-cnts : Oid Oid -> Prop .

eq < A : Sender | buff : L2, rec : B, cnt : N, ack-w : TV >
   < B : Receiver | buff : L1, snd : A, cnt : N > C |= same-cnts(A,B) = true .
```

Now we can ask and answer the original question **E** $\varphi(A, B)$, i.e.,

E `<> ~ same-cnts(A,B)`

COMM Revisited (VII)

```
op same-cnts : Oid Oid -> Prop .

eq < A : Sender | buff : L2, rec : B, cnt : N, ack-w : TV >
   < B : Receiver | buff : L1, snd : A, cnt : N > C |= same-cnts(A,B) = true .
```

Now we can ask and answer the original question **E** $\varphi(A, B)$, i.e.,

E `<> ~ same-cnts(A,B)`

by model checking $\neg\varphi(A, B)$, that is, by model checking

COMM Revisited (VII)

```
op same-cnts : Oid Oid -> Prop .

eq < A : Sender | buff : L2, rec : B, cnt : N, ack-w : TV >
   < B : Receiver | buff : L1, snd : A, cnt : N > C |= same-cnts(A,B) = true .
```

Now we can ask and answer the original question **E** $\varphi(A, B)$, i.e.,

E `<> ~ same-cnts(A,B)`

by model checking $\neg\varphi(A, B)$, that is, by model checking

`[] same-cnts(A,B)`

COMM Revisited (VII)

```
op same-cnts : Oid Oid -> Prop .

eq < A : Sender | buff : L2, rec : B, cnt : N, ack-w : TV >
   < B : Receiver | buff : L1, snd : A, cnt : N > C |= same-cnts(A,B) = true .
```

Now we can ask and answer the original question **E** $\varphi(A, B)$, i.e.,

E `<> ~ same-cnts(A,B)`

by model checking $\neg\varphi(A, B)$, that is, by model checking

`[] same-cnts(A,B)`

and getting as a proof the counterexample:

COMM Revisited (VIII)

```
red modelCheck(init('a,'b,1),[] same-cnts('a,'b)) .

result ModelCheckResult: counterexample(
{< 'a : Sender | buff : 1, rec : 'b, cnt : 0, ack-w : false >
 < 'b : Receiver | buff : nil, cnt : 0, snd : 'a >,'snd}
{< 'a : Sender | buff : nil, rec : 'b, cnt : 0, ack-w : true >
 < 'b : Receiver | buff : nil, cnt : 0, snd : 'a >
  to 'b from 'a val 1 cnt 0,'rec}
{< 'a : Sender | buff : nil, rec : 'b, cnt : 0, ack-w : true >
 < 'b : Receiver | buff : 1, cnt : 1, snd : 'a >
  to 'a from 'b ack 0,'ack-rec},
{< 'a : Sender | buff : nil, rec : 'b, cnt : 1, ack-w : false >
 < 'b : Receiver | buff : 1, cnt : 1, snd : 'a >,deadlock})
```