Program Verification: Lecture 24

José Meseguer

University of Illinois at Urbana-Champaign

Fairness Properties

Fairness properties are closely related to the phenomena of conflict and preemption between transitions.

Fairness Properties

Fairness properties are closely related to the phenomena of conflict and preemption between transitions. Since any computable Kripke structure can be specified as a Maude system module, W.L.O.G. we can consider these fenomena for Kripke structures $\mathbb{C}^{\Pi}_{\mathcal{R}}$.

Fairness Properties

Fairness properties are closely related to the phenomena of conflict and preemption between transitions. Since any computable Kripke structure can be specified as a Maude system module, W.L.O.G. we can consider these fenomena for Kripke structures $\mathbb{C}^{\Pi}_{\mathcal{R}}$. Given a state $[u]$ in $\mathbb{C}^{\Pi}_{\mathcal{R}}$, two different state transitions $[u] \rightarrow_{\mathcal{R}} [v]$ and $[u] \rightarrow_{\mathcal{R}} [w]$ are said to be in conflict, because the application of one will preempt that of the other.

Fairness Properties

Fairness properties are closely related to the phenomena of conflict and preemption between transitions. Since any computable Kripke structure can be specified as a Maude system module, W.L.O.G. we can consider these fenomena for Kripke structures $\mathbb{C}^{\Pi}_{\mathcal{R}}$. Given a state $[u]$ in $\mathbb{C}^{\Pi}_{\mathcal{R}}$, two different state transitions $[u] \rightarrow_{\mathcal{R}} [v]$ and $[u] \rightarrow_{\mathcal{R}} [w]$ are said to be in conflict, because the application of one will preempt that of the other. The notions of race condition and data race are phenomena arising from conflict and preemption.

Fairness Properties

Fairness properties are closely related to the phenomena of conflict and preemption between transitions. Since any computable Kripke structure can be specified as a Maude system module, W.L.O.G. we can consider these fenomena for Kripke structures $\mathbb{C}_{\mathcal{R}}^{\Pi}$. Given a state $[u]$ in $\mathbb{C}_{\mathcal{R}}^{\Pi}$, two different state transitions $[u] \rightarrow_{\mathcal{R}} [v]$ and $[u] \rightarrow_{\mathcal{R}} [w]$ are said to be in conflict, because the application of one will preempt that of the other. The notions of race condition and data race are phenomena arising from conflict and preemption.

Fairness is a property ensuring that in certain kinds of conflict situations a given transition will not be preempted almost forever.

Fairness Properties

Fairness properties are closely related to the phenomena of conflict and preemption between transitions. Since any computable Kripke structure can be specified as a Maude system module, W.L.O.G. we can consider these fenomena for Kripke structures $\mathbb{C}^{\Pi}_{\mathcal{R}}$. Given a state $[u]$ in $\mathbb{C}^{\Pi}_{\mathcal{R}}$, two different state transitions $[u] \rightarrow_{\mathcal{R}} [v]$ and $[u] \rightarrow_{\mathcal{R}} [w]$ are said to be in conflict, because the application of one will preempt that of the other. The notions of race condition and data race are phenomena arising from conflict and preemption.

Fairness is a property ensuring that in certain kinds of conflict situations a given transition will not be preempted almost forever. That is, if it is infinitely enabled to be applied, it will actually be applied, not a finite, but an infinite number of times.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $\mathit{enabled}_\tau$ is the state predicate describing that transition τ is enabled.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either:

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_\tau$, i.e., after a while τ is forever enabled; or

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond enabled_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $\mathit{enabled}_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box\mathit{enabled}_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond\mathit{enabled}_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_{\tau}$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_{\tau}$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond enabled_{\tau}$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

1. **Weak Fairness**: $\Diamond\Box enabled_{\tau} \rightarrow \Box\Diamond taken_{\tau}$

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $\mathit{enabled}_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box\mathit{enabled}_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond\mathit{enabled}_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

1. **Weak Fairness**: $\Diamond\Box\mathit{enabled}_\tau \rightarrow \Box\Diamond\mathit{taken}_\tau$
2. **Strong Fairness**: $\Box\Diamond\mathit{enabled}_\tau \rightarrow \Box\Diamond\mathit{taken}_\tau$.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond enabled_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

1. **Weak Fairness**: $\Diamond\Box enabled_\tau \to \Box\Diamond taken_\tau$
2. **Strong Fairness**: $\Box\Diamond enabled_\tau \to \Box\Diamond taken_\tau$.

The problem is that $taken_\tau$ is the property of an action (applying τ), which need not be reflected in a state.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond enabled_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

1. **Weak Fairness**: $\Diamond\Box enabled_\tau \rightarrow \Box\Diamond taken_\tau$
2. **Strong Fairness**: $\Box\Diamond enabled_\tau \rightarrow \Box\Diamond taken_\tau$.

The problem is that $taken_\tau$ is the property of an action (applying τ), which need not be reflected in a state. But $LTL(\Pi)$ is a state-based temporal logic (the $p \in \Pi$ are state predicates).

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond enabled_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

1. **Weak Fairness**: $\Diamond\Box enabled_\tau \rightarrow \Box\Diamond taken_\tau$
2. **Strong Fairness**: $\Box\Diamond enabled_\tau \rightarrow \Box\Diamond taken_\tau$.

The problem is that $taken_\tau$ is the property of an action (applying τ), which need not be reflected in a state. But $LTL(\Pi)$ is a state-based temporal logic (the $p \in \Pi$ are state predicates). So specifying fairness conditions in $LTL(\Pi)$ can be tricky.

Weak and Strong Fairness

Depending on how we interpret that a transition is infinitely enabled to be applied, we get two different notions of fairness. Suppose that $enabled_\tau$ is the state predicate describing that transition τ is enabled. Then, "infinitely enabled to be applied" can mean either: (1) $\Diamond\Box enabled_\tau$, i.e., after a while τ is forever enabled; or (2) $\Box\Diamond enabled_\tau$, i.e., τ is infinitely often enabled, but can be so in an intermitten way. This distinction yields two notions:

1. **Weak Fairness**: $\Diamond\Box enabled_\tau \rightarrow \Box\Diamond taken_\tau$
2. **Strong Fairness**: $\Box\Diamond enabled_\tau \rightarrow \Box\Diamond taken_\tau$.

The problem is that $taken_\tau$ is the property of an action (applying τ), which need not be reflected in a state. But $LTL(\Pi)$ is a state-based temporal logic (the $p \in \Pi$ are state predicates). So specifying fairness conditions in $LTL(\Pi)$ can be tricky. Can consider three, increasingly more expressive modes of specifying fairness:

Three Modes of Specifying Fairness

Weak or strong fairness can be specified in several modes:

Three Modes of Specifying Fairness

Weak or strong fairness can be specified in several modes:

1. **State-Based** mode, when the $taken_\tau$ property can be expressed as a state predicate holding in the resulting state.

Three Modes of Specifying Fairness

Weak or strong fairness can be specified in several modes:

1. **State-Based** mode, when the $taken_\tau$ property can be expressed as a state predicate holding in the resulting state.
2. **Action-Based** mode, by encoding in the system's state the label l of the transition used to reach it.

Three Modes of Specifying Fairness

Weak or strong fairness can be specified in several modes:

1. **State-Based** mode, when the $taken_\tau$ property can be expressed as a state predicate holding in the resulting state.
2. **Action-Based** mode, by encoding in the system's state the label l of the transition used to reach it. This increases the number of states, since a state $[u]$ now splits into $[u].l_1, \ldots, [u].l_n$ if it can be reached by n different transitions.

Three Modes of Specifying Fairness

Weak or strong fairness can be specified in several modes:

1. **State-Based** mode, when the $taken_\tau$ property can be expressed as a state predicate holding in the resulting state.
2. **Action-Based** mode, by encoding in the system's state the label l of the transition used to reach it. This increases the number of states, since a state $[u]$ now splits into $[u].l_1, \ldots, [u].l_n$ if it can be reached by n different transitions.
3. **Object/Process/Thread Fairness** is even more detailed: we need to specify to which object/process/thread has transition l been applied by encoding this in the resulting state $[v]$ as, say, $[v].l(o)$, where o is the object/process/thread identifier.

Three Modes of Specifying Fairness

Weak or strong fairness can be specified in several modes:

1. **State-Based** mode, when the $taken_\tau$ property can be expressed as a state predicate holding in the resulting state.
2. **Action-Based** mode, by encoding in the system's state the label l of the transition used to reach it. This increases the number of states, since a state $[u]$ now splits into $[u].l_1, \ldots, [u].l_n$ if it can be reached by n different transitions.
3. **Object/Process/Thread Fairness** is even more detailed: we need to specify to which object/process/thread has transition l been applied by encoding this in the resulting state $[v]$ as, say, $[v].l(o)$, where o is the object/process/thread identifier.

The difference between (2) and (3) is that between applying a rule l, and applying an instance of rule l to a given object o.

I will illustrate modes (1) and (3) by examples.

A Traffic Lights Example

In Urbana-Champaign most street intersections are $N - S$ intersecting $E - W$, which, by analogy with a map, I shall call vertical intersecting horizontal.

A Traffic Lights Example

In Urbana-Champaign most street intersections are $N - S$ intersecting $E - W$, which, by analogy with a map, I shall call vertical intersecting horizontal. Here is a simple specification of a traffic lights system of this kind:

A Traffic Lights Example

In Urbana-Champaign most street intersections are $N - S$ intersecting $E - W$, which, by analogy with a map, I shall call vertical intersecting horizontal. Here is a simple specification of a traffic lights system of this kind:

```
mod TRAFFIC-LIGHTS is
  sorts Conf LightState Intersection Direction Light Car .
  subsorts LightState Intersection Car < Conf .
  op mt : -> Conf [ctor] .
  op _ _ : Conf Conf -> Conf [ctor assoc comm id: mt] .
  op [_] : Conf -> Intersection [ctor] .
  ops h v : -> Direction [ctor] .
  op car : Direction -> Car [ctor] .
  ops green red yellow  : Direction -> Light [ctor] .
  op {_,_} : Light Light -> LightState [comm] .

   op init : -> Conf .

  vars d d1 d2 : Direction . var L : Light .  var C : Conf .

  eq init = {green(h),red(v)} [mt] .
```

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

Within the horizontal, resp. vertical, direction no distinction is made between a light facing (or a car moving) N or S (resp. E or W).

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

Within the horizontal, resp. vertical, direction no distinction is made between a light facing (or a car moving) N or S (resp. E or W). The light system has just two transitions.

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

Within the horizontal, resp. vertical, direction no distinction is made between a light facing (or a car moving) N or S (resp. E or W). The light system has just two transitions. While a ligh is green in direction d, cars moving along d can enter the intersection.

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

Within the horizontal, resp. vertical, direction no distinction is made between a light facing (or a car moving) N or S (resp. E or W). The light system has just two transitions. While a ligh is green in direction d, cars moving along d can enter the intersection. We assume that no more than two such cars (e.g., one going $N \rightarrow S$ and another $S \rightarrow N$) do so; and of course they leave.

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

Within the horizontal, resp. vertical, direction no distinction is made between a light facing (or a car moving) N or S (resp. E or W). The light system has just two transitions. While a ligh is green in direction d, cars moving along d can enter the intersection. We assume that no more than two such cars (e.g., one going $N \rightarrow S$ and another $S \rightarrow N$) do so; and of course they leave. This is modeled by the `[car.in]` and `[car.out]` rules.

A Traffic Lights Example (II)

```
  rl [g2y] : {green(d1),red(d2)} [C] => {yellow(d1),red(d2)} [C] .
  rl [y2r] : {yellow(d1),red(d2)} [mt]  => {red(d1),green(d2)} [mt] .

  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d)] .
  rl [car.in] : {green(d),L} [mt] => {green(d),L} [car(d) car(d)] .
  rl [car.out] : {green(d),L} [car(d) car(d)] => {green(d),L} [mt] .
  rl [car.out] : {green(d),L} [car(d)] => {green(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d) car(d)] => {yellow(d),L} [mt] .
  rl [car.out] : {yellow(d),L} [car(d)] => {yellow(d),L} [mt] .
endm
```

Within the horizontal, resp. vertical, direction no distinction is made between a light facing (or a car moving) N or S (resp. E or W). The light system has just two transitions. While a ligh is green in direction d, cars moving along d can enter the intersection. We assume that no more than two such cars (e.g., one going $N \rightarrow S$ and another $S \rightarrow N$) do so; and of course they leave. This is modeled by the `[car.in]` and `[car.out]` rules. Let us define some state predicates and formulas for `TRAFFIC-LIGHTS`.

A Traffic Lights Example (III)

```
in model-checker.maude

mod TRAFFIC-LIGHTS-PREDS is
  protecting TRAFFIC-LIGHTS .  protecting SATISFACTION .

  subsort Conf < State .

  vars L L' : Light .  vars C C' : Conf .  vars d d1 d2 : Direction .

  op enabled : -> Prop [ctor] .

  eq {green(d1),red(d2)} [C] C' |= enabled = true .
  eq {yellow(d1),red(d2)} [mt] C |= enabled = true .
  eq {green(d),L} [mt] C |= enabled = true .
  eq  {green(d),L} [car(d) car(d)]  C |= enabled = true .
  eq  {green(d),L} [car(d)]  C |= enabled = true .
  eq  {yellow(d),L} [car(d) car(d)]  C |= enabled = true .
  eq  {yellow(d),L} [car(d)]  C |= enabled = true .

  op on : Light -> Prop [ctor] .

   eq {L,L'} C |= on(L) = true .
```

A Traffic Lights Example (IV)

```
   op side-collision-dngr : -> Prop [ctor] .

   eq [car(h) car(v) C'] C |= side-collision-dngr = true .

   op yellow-enabled : Direction -> Prop .

   eq {green(d1),red(d2)} [C] C' |= yellow-enabled(d1) = true .
endm

mod TRAFFIC-LIGHTS-CHECK is
  protecting TRAFFIC-LIGHTS-PREDS .
  including MODEL-CHECKER .

  op yellow-fair : -> Formula .

  eq yellow-fair = (([] <> yellow-enabled(h)) ->  ([] <> on(yellow(h)))) /\
                   (([] <> yellow-enabled(v)) ->  ([] <> on(yellow(v)))) .
endm
```

A Traffic Lights Example (IV)

```
   op side-collision-dngr : -> Prop [ctor] .

   eq [car(h) car(v) C'] C |= side-collision-dngr = true .

   op yellow-enabled : Direction -> Prop .

   eq {green(d1),red(d2)} [C] C' |= yellow-enabled(d1) = true .
endm

mod TRAFFIC-LIGHTS-CHECK is
  protecting TRAFFIC-LIGHTS-PREDS .
  including MODEL-CHECKER .

  op yellow-fair : -> Formula .

  eq yellow-fair = (([] <> yellow-enabled(h)) ->  ([] <> on(yellow(h)))) /\
                   (([] <> yellow-enabled(v)) ->  ([] <> on(yellow(v)))) .
endm
```

Let's verify some properties.

A Traffic Lights Example (IV)

```
   op side-collision-dngr : -> Prop [ctor] .

   eq [car(h) car(v) C'] C |= side-collision-dngr = true .

   op yellow-enabled : Direction -> Prop .

   eq {green(d1),red(d2)} [C] C' |= yellow-enabled(d1) = true .
endm

mod TRAFFIC-LIGHTS-CHECK is
  protecting TRAFFIC-LIGHTS-PREDS .
  including MODEL-CHECKER .

  op yellow-fair : -> Formula .

  eq yellow-fair = (([] <> yellow-enabled(h)) ->  ([] <> on(yellow(h)))) /\
                   (([] <> yellow-enabled(v)) ->  ([] <> on(yellow(v)))) .
endm
```

Let's verify some properties. The main safety invariant is absence of side collisions:

A Traffic Lights Example (V)

```
red modelCheck(init,[] ~ side-collision-dngr) .

result Bool: true
```

A Traffic Lights Example (V)

```
red modelCheck(init,[] ~ side-collision-dngr) .

result Bool: true
```

Another important invariant is deadlock freedom:

A Traffic Lights Example (V)

```
red modelCheck(init,[] ~ side-collision-dngr) .
```

```
result Bool: true
```

Another important invariant is deadlock freedom:

```
red modelCheck(init,[] enabled) .
```

```
result Bool: true
```

A Traffic Lights Example (V)

```
red modelCheck(init,[] ~ side-collision-dngr) .

result Bool: true
```

Another important invariant is deadlock freedom:

```
red modelCheck(init,[] enabled) .

result Bool: true
```

A key property is that in any direction red always follows yellow:

A Traffic Lights Example (V)

```
red modelCheck(init,[] ~ side-collision-dngr) .

result Bool: true
```

Another important invariant is deadlock freedom:

```
red modelCheck(init,[] enabled) .

result Bool: true
```

A key property is that in any direction red always follows yellow:

```
red modelCheck(init,[] (on(yellow(h)) -> (on(yellow(h)) U on(red(h))))) .

result Bool: true

red modelCheck(init,[] (on(yellow(v)) -> (on(yellow(v)) U on(red(v))))) .

result Bool: true
```

A Traffic Lights Example (V)

```
red modelCheck(init,[] ~ side-collision-dngr) .

result Bool: true
```

Another important invariant is deadlock freedom:

```
red modelCheck(init,[] enabled) .

result Bool: true
```

A key property is that in any direction red always follows yellow:

```
red modelCheck(init,[] (on(yellow(h)) -> (on(yellow(h)) U on(red(h))))) .

result Bool: true

red modelCheck(init,[] (on(yellow(v)) -> (on(yellow(v)) U on(red(v))))) .

result Bool: true
```

However, yellow doesn't always follow green:

A Traffic Lights Example (VI)

```
red modelCheck(init,[] (on(green(h)) -> (on(green(h)) U on(yellow(h))))) .

result ModelCheckResult:
counterexample(nil,
                {[mt] {green(h),red(v)},'car.in}
{[car(h)] {green(h),red(v)},'car.out})
```

A Traffic Lights Example (VI)

```
red modelCheck(init,[] (on(green(h)) -> (on(green(h)) U on(yellow(h))))) .

result ModelCheckResult:
counterexample(nil,
                {[mt] {green(h),red(v)},'car.in}
{[car(h)] {green(h),red(v)},'car.out})
```

As the counterexample shows this is due to a conflict between the g2y rule and the `car.in` rules, and g2y gets forever preempted.

A Traffic Lights Example (VI)

```
red modelCheck(init,[] (on(green(h)) -> (on(green(h)) U on(yellow(h))))) .

result ModelCheckResult:
counterexample(nil,
                {[mt] {green(h),red(v)},'car.in}
{[car(h)] {green(h),red(v)},'car.out})
```

As the counterexample shows this is due to a conflict between the g2y rule and the `car.in` rules, and g2y gets forever preempted.

We can take two steps:

A Traffic Lights Example (VI)

```
red modelCheck(init,[] (on(green(h)) -> (on(green(h)) U on(yellow(h))))) .

result ModelCheckResult:
counterexample(nil,
                {[mt] {green(h),red(v)},'car.in}
{[car(h)] {green(h),red(v)},'car.out})
```

As the counterexample shows this is due to a conflict between the g2y rule and the `car.in` rules, and g2y gets forever preempted.

We can take two steps: **Step 1**. Consider `TRAFFIC-LIGHTS` a high-level design missing some details and, assuming `yellow-fair`, show that `TRAFFIC-LIGHTS` works as expected:

A Traffic Lights Example (VI)

```
red modelCheck(init,[] (on(green(h)) -> (on(green(h)) U on(yellow(h))))) .

result ModelCheckResult:
counterexample(nil,
                {[mt] {green(h),red(v)},'car.in}
{[car(h)] {green(h),red(v)},'car.out})
```

As the counterexample shows this is due to a conflict between the g2y rule and the `car.in` rules, and g2y gets forever preempted.

We can take two steps: **Step 1**. Consider `TRAFFIC-LIGHTS` a high-level design missing some details and, assuming `yellow-fair`, show that `TRAFFIC-LIGHTS` works as expected:

```
red modelCheck(init,yellow-fair ->
          ([] (on(green(h)) -> (on(green(h)) U on(yellow(h)))))) .

result Bool: true

red modelCheck(init,yellow-fair ->
          ([] (on(green(v)) -> (on(green(v)) U on(yellow(v)))))) .

result Bool: true
```

A Traffic Lights Example (VII)

Step 2. Develop a more detailed design where the traffic lights system works as expected because its design ensures fairness by construction. This second step is taken in the Appendix to this lecture.

A Traffic Lights Example (VII)

Step 2. Develop a more detailed design where the traffic lights system works as expected because its design ensures fairness by construction. This second step is taken in the Appendix to this lecture.

This example has illustrated the **State-Based** mode:

A Traffic Lights Example (VII)

Step 2. Develop a more detailed design where the traffic lights system works as expected because its design ensures fairness by construction. This second step is taken in the Appendix to this lecture.

This example has illustrated the **State-Based** mode: we didn't need to explicitly encode the taking of a conflict transition like g2y in the state because its effect could be detected by the yellow light for the relevant direction being on after it was taken.

A Traffic Lights Example (VII)

Step 2. Develop a more detailed design where the traffic lights system works as expected because its design ensures fairness by construction. This second step is taken in the Appendix to this lecture.

This example has illustrated the **State-Based** mode: we didn't need to explicitly encode the taking of a conflict transition like g2y in the state because its effect could be detected by the yellow light for the relevant direction being on after it was taken.

Next we consider the **Object/Process/Thread Fairness** mode by revisiting the `PARALLEL` programming language from Lecture 20.

A Traffic Lights Example (VII)

Step 2. Develop a more detailed design where the traffic lights system works as expected because its design ensures fairness by construction. This second step is taken in the Appendix to this lecture.

This example has illustrated the **State-Based** mode: we didn't need to explicitly encode the taking of a conflict transition like g2y in the state because its effect could be detected by the yellow light for the relevant direction being on after it was taken.

Next we consider the **Object/Process/Thread Fairness** mode by revisiting the `PARALLEL` programming language from Lecture 20. This will also allow us to illustrate the LTL formal verification of concurrent imperative programs.

PARALLEL Revisited

In PARALLEL, to verify some LTL program properties we need to be able to express **Process Fairness**. We can do so by: (1) slightly modifying the main state costructor:

PARALLEL Revisited

In PARALLEL, to verify some LTL program properties we need to be able to express **Process Fairness**. We can do so by: (1) slightly modifying the main state costructor:

```
op {_,_} : Soup Memory -> MachineState .
```

PARALLEL Revisited

In PARALLEL, to verify some LTL program properties we need to be able to express **Process Fairness**. We can do so by: (1) slightly modifying the main state costructor:

```
op {_,_} : Soup Memory -> MachineState .
```

to record the last process that modified the state. This can be achieved with the state constructor:

PARALLEL Revisited

In PARALLEL, to verify some LTL program properties we need to be able to express **Process Fairness**. We can do so by: (1) slightly modifying the main state costructor:

```
op {_,_} : Soup Memory -> MachineState .
```

to record the last process that modified the state. This can be achieved with the state constructor:

```
op {_,_,_} : Soup Memory Pid -> MachineState .
```

PARALLEL Revisited

In PARALLEL, to verify some LTL program properties we need to be able to express **Process Fairness**. We can do so by: (1) slightly modifying the main state costructor:

```
op {_,_} : Soup Memory -> MachineState .
```

to record the last process that modified the state. This can be achieved with the state constructor:

```
op {_,_,_} : Soup Memory Pid -> MachineState .
```

and (2) slightly modify the rewrite rules of PARALLEL so that they record the pid of the last executing process.

PARALLEL Revisited

In PARALLEL, to verify some LTL program properties we need to be able to express **Process Fairness**. We can do so by: (1) slightly modifying the main state costructor:

```
op {_,_} : Soup Memory -> MachineState .
```

to record the last process that modified the state. This can be achieved with the state constructor:

```
op {_,_,_} : Soup Memory Pid -> MachineState .
```

and (2) slightly modify the rewrite rules of PARALLEL so that they record the pid of the last executing process.

The only changes needed in the specification of PARALLEL in Lecture 20 are the slight modifications (1) and (2) explained above. Here is the modified specification of PARALLEL:

PARALLEL Revisited (II)

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

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

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

PARALLEL Revisited (III)

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

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

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

  crl {[I, (Q := E) ; R] | S, M, J} =>
        {[I, R] | S, [Q,eval(E,M)] M, I} if Q in M =/= true .

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

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

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

Dekker's Mutex Algorithm

Dekker's algorithm is specified extending the modified `PARALLEL`:

Dekker's Mutex Algorithm

Dekker's algorithm is specified extending the modified PARALLEL:

```
mod DEKKER is inc PARALLEL . subsort Int < Pid .
  op crit : -> UserStatement .
  op rem : -> LoopingUserStatement .
  ops p1 p2 : -> Program .
  op initialMem : -> Memory .
  op initial : -> MachineState .
  eq p1 =
        repeat
          'c1 := 1 ;
          while 'c2 = 1 do
            if 'turn = 2 then
              'c1 := 0 ;
              while 'turn = 2 do skip od ;
              'c1 := 1
            fi
          od ;
          crit ;
          'turn := 2 ;
          'c1 := 0 ;
          rem
        forever .
```

Dekker's Mutex Algorithm (II)

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

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

Dekker's Mutex Algorithm (II)

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

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

LTL Model Checking of Dekker's Algorithm

We need to define an `enabled` predicate and three predicates parameterized by the process id: `in-crit` and `in-rem`, when the process is resp. in its critical section, resp. in its remaining code fragment, and `exec`, when the process has just executed.

LTL Model Checking of Dekker's Algorithm

We need to define an `enabled` predicate and three predicates parameterized by the process id: `in-crit` and `in-rem`, when the process is resp. in its critical section, resp. in its remaining code fragment, and `exec`, when the process has just executed.

```
mod DEKKER-PREDS is inc DEKKER .  inc SATISFACTION .
  inc LTL-SIMPLIFIER .
  subsort MachineState < State .

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

  op enabled : -> Prop .

  eq {[I, U ; R] | S, M, J} |= enabled = true .
  eq {[I, L ; R] | S, M, J} |= enabled = true .
  eq {[I, (Q := E) ; R] | S, [Q, X] M, J} |= enabled = true .
  eq {[I, (Q := E) ; R] | S, M, J} |= enabled = true .
  eq {[I, if T then P fi ; R] | S, M, J} |= enabled = true .
  eq {[I, while T do P od ; R] | S, M, J} |= enabled = true .
  eq {[I, repeat P forever ; R] | S, M, J} |= enabled = true .
```

LTL Model Checking of Dekker's Algorithm (II)

```
ops in-crit in-rem exec : Pid -> Prop .

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

mod DEKKER-CHECK is inc DEKKER-PREDS .  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
endm
```

LTL Model Checking of Dekker's Algorithm (II)

```
ops in-crit in-rem exec : Pid -> Prop .

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

mod DEKKER-CHECK is inc DEKKER-PREDS .  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
endm
```

We can now verify mutual exclusion and deadlock freedom:

LTL Model Checking of Dekker's Algorithm (II)

```
ops in-crit in-rem exec : Pid -> Prop .

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

mod DEKKER-CHECK is inc DEKKER-PREDS .  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
endm
```

We can now verify mutual exclusion and deadlock freedom:

```
red modelCheck(initial,[]~ (in-crit(1) /\ in-crit(2))) .

result Bool: true

red modelCheck(initial,[] enabled) .

result Bool: true
```

LTL Model Checking of Dekker's Algorithm (III)

The strong fairness property that executing infinitely often implies entering one's critical section infinitely often does fail:

LTL Model Checking of Dekker's Algorithm (III)

The strong fairness property that executing infinitely often implies entering one's critical section infinitely often does fail:

```
red modelCheck(initial,[]<> exec(1) -> []<> in-crit(1)) .

result ModelCheckResult:
counterexample({{[1,repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then ...
```

LTL Model Checking of Dekker's Algorithm (III)

The strong fairness property that executing infinitely often implies entering one's critical section infinitely often does fail:

```
red modelCheck(initial,[]<> exec(1) -> []<> in-crit(1)) .

result ModelCheckResult:
counterexample({{[1,repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then ...
```

If p1 and p2 both get to execute infinitely often, the property that if p1 is infinitely often out of its `rem` section it enters its critical section infinitely often does hold. And the same holds for p2.

LTL Model Checking of Dekker's Algorithm (III)

The strong fairness property that executing infinitely often implies entering one's critical section infinitely often does fail:

```
red modelCheck(initial,[]<> exec(1) -> []<> in-crit(1)) .

result ModelCheckResult:
counterexample({{[1,repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2 then ...
```

If p1 and p2 both get to execute infinitely often, the property that if p1 is infinitely often out of its `rem` section it enters its critical section infinitely often does hold. And the same holds for p2.

```
red  modelCheck(initial,[]<> exec(1) /\ []<> exec(2)
                                -> []<> ~ in-rem(1) -> []<> in-crit(1)) .

result Bool: true

red  modelCheck(initial,[]<> exec(2) /\ []<> exec(1)
                                -> []<> ~ in-rem(2) -> []<> in-crit(2)) .

result Bool: true
```

LTL Model Checking of Dekker's Algorithm (IV)

The `PARALLEL` example has illustrated two main points:

LTL Model Checking of Dekker's Algorithm (IV)

The PARALLEL example has illustrated two main points:

1. **LTL Properties** of concurrent imperative programs can be model checked directly from the rewriting logic semantics of the language (no language-specific tool is needed).

LTL Model Checking of Dekker's Algorithm (IV)

The `PARALLEL` example has illustrated two main points:

1. **LTL Properties** of concurrent imperative programs can be model checked directly from the rewriting logic semantics of the language (no language-specific tool is needed).
2. The `PARALLEL` example illustrates the usefulness of the **Object/Process/Thread Fairness** mode.

LTL Model Checking of Dekker's Algorithm (IV)

The PARALLEL example has illustrated two main points:

1. **LTL Properties** of concurrent imperative programs can be model checked directly from the rewriting logic semantics of the language (no language-specific tool is needed).
2. The PARALLEL example illustrates the usefulness of the **Object/Process/Thread Fairness** mode.

For PARALLEL we only needed to record in the machine state the pid of the process that had last executed.

LTL Model Checking of Dekker's Algorithm (IV)

The `PARALLEL` example has illustrated two main points:

1. **LTL Properties** of concurrent imperative programs can be model checked directly from the rewriting logic semantics of the language (no language-specific tool is needed).
2. The `PARALLEL` example illustrates the usefulness of the **Object/Process/Thread Fairness** mode.

For `PARALLEL` we only needed to record in the machine state the `pid` of the process that had last executed. But in other **Object/Process/Thread Fairness** mode examples we often need to record more information.

LTL Model Checking of Dekker's Algorithm (IV)

The PARALLEL example has illustrated two main points:

1. **LTL Properties** of concurrent imperative programs can be model checked directly from the rewriting logic semantics of the language (no language-specific tool is needed).
2. The PARALLEL example illustrates the usefulness of the **Object/Process/Thread Fairness** mode.

For PARALLEL we only needed to record in the machine state the pid of the process that had last executed. But in other **Object/Process/Thread Fairness** mode examples we often need to record more information. For example, information of the form $l(o)$, recording that rule l was the last rule executed and that it was applied to object/process/thread o.