Automated Program Verification and Testing 15414/15614 Fall 2016 Lecture 24: Symbolic Model Checking 2, Spin

Matt Fredrikson
mfredrik@cs.cmu.edu

November 29, 2016

Symbolic Transition Systems (Recap)

We'll represent states by their atomic propositions:

Symbolic Transition Systems (Recap)

We'll represent states by their atomic propositions:

- Need to assume that states are uniquely determined by their propositions

Symbolic Transition Systems (Recap)

We'll represent states by their atomic propositions:

- Need to assume that states are uniquely determined by their propositions
- I.e., for any $s, s' \in S$ where $s \neq s'$, $L(s) \neq L(s')$

Symbolic Transition Systems (Recap)

We'll represent states by their atomic propositions:

- Need to assume that states are uniquely determined by their propositions
- I.e., for any $s, s' \in S$ where $s \neq s'$, $L(s) \neq L(s')$
- Then if $L(s) = p_1, \ldots, p_n$, we'll refer to s by writing:

$$p_1 \wedge \cdots \wedge p_n$$

Symbolic Transition Systems (Recap)

We'll represent states by their atomic propositions:

- Need to assume that states are uniquely determined by their propositions
- I.e., for any $s, s' \in S$ where $s \neq s'$, $L(s) \neq L(s')$
- Then if $L(s) = p_1, \ldots, p_n$, we'll refer to s by writing:

$$p_1 \wedge \cdots \wedge p_n$$

- If ϕ is a formula over atomic propositions, then

ϕ refers to the set $\{s \in S \mid s \models \phi\}$

Symbolic Transition Systems (Recap)

We'll represent states by their atomic propositions:

- Need to assume that states are uniquely determined by their propositions
- I.e., for any $s, s' \in S$ where $s \neq s'$, $L(s) \neq L(s')$
- Then if $L(s) = p_1, \ldots, p_n$, we'll refer to s by writing:

$$p_1 \wedge \cdots \wedge p_n$$

- If ϕ is a formula over atomic propositions, then

ϕ refers to the set $\{s \in S \mid s \models \phi\}$

Recall: this is similar to how we treated assertions in Hoare logic

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

The transition relation is just a set of these pairs, so as a predicate,

$$R(s, s') = 1 \Leftrightarrow (s, s') \in R$$

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

The transition relation is just a set of these pairs, so as a predicate,

$$R(s, s') = 1 \Leftrightarrow (s, s') \in R$$

We'll represent transition predicates using atomic propositions:

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

The transition relation is just a set of these pairs, so as a predicate,

$$R(s, s') = 1 \Leftrightarrow (s, s') \in R$$

We'll represent transition predicates using atomic propositions:

- To refer to "next state", prime the proposition symbols

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

The transition relation is just a set of these pairs, so as a predicate,

$$R(s, s') = 1 \Leftrightarrow (s, s') \in R$$

We'll represent transition predicates using atomic propositions:

- To refer to "next state", prime the proposition symbols
- So the predicate $(p_1 \wedge \neg p_2) \wedge (p'_1 \wedge p'_2)$:

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

The transition relation is just a set of these pairs, so as a predicate,

$$R(s, s') = 1 \Leftrightarrow (s, s') \in R$$

We'll represent transition predicates using atomic propositions:

- To refer to "next state", prime the proposition symbols
- So the predicate $(p_1 \wedge \neg p_2) \wedge (p'_1 \wedge p'_2)$:
  1. Begins in the state where p_1 is true and p_2 is false

Symbolic Transition Systems (Recap)

We also represent transitions as predicates

Transitions reference **ordered pairs** of states (s, s')

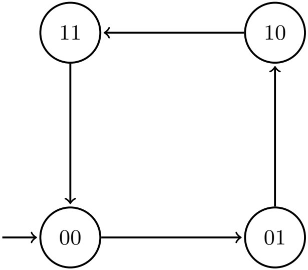
The transition relation is just a set of these pairs, so as a predicate,

$$R(s, s') = 1 \Leftrightarrow (s, s') \in R$$

We'll represent transition predicates using atomic propositions:

- To refer to "next state", prime the proposition symbols
- So the predicate $(p_1 \wedge \neg p_2) \wedge (p_1' \wedge p_2')$:
  1. Begins in the state where p_1 is true and p_2 is false
  2. Ends in the state where both p_1 and p_2 are true

Example: Symbolic Representation



Symbolic transitions:

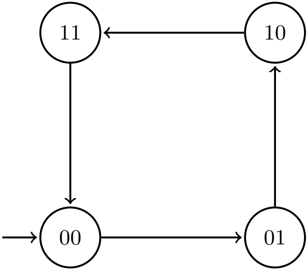
$$
\begin{array}{ll}
 & (v_0 = 0 \wedge v_1 = 0 \wedge v_0' = 0 \wedge v_1' = 1) \\
\vee & (v_0 = 0 \wedge v_1 = 1 \wedge v_0' = 1 \wedge v_1' = 0) \\
\vee & (v_0 = 1 \wedge v_1 = 0 \wedge v_0' = 1 \wedge v_1' = 1) \\
\vee & (v_0 = 1 \wedge v_1 = 1 \wedge v_0' = 0 \wedge v_1' = 0)
\end{array}
$$

Initial state: $v_0 = 0 \wedge v_0 = 1$

The transitions are a predicate

$$\psi_R(v_0, v_1, v_0', v_1')$$

Example: Symbolic Representation



Symbolic transitions:

$$
\begin{array}{ll}
 & (v_0 = 0 \wedge v_1 = 0 \wedge v'_0 = 0 \wedge v'_1 = 1) \\
\vee & (v_0 = 0 \wedge v_1 = 1 \wedge v'_0 = 1 \wedge v'_1 = 0) \\
\vee & (v_0 = 1 \wedge v_1 = 0 \wedge v'_0 = 1 \wedge v'_1 = 1) \\
\vee & (v_0 = 1 \wedge v_1 = 1 \wedge v'_0 = 0 \wedge v'_1 = 0)
\end{array}
$$

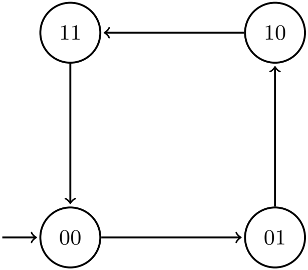
Initial state: $v_0 = 0 \wedge v_0 = 1$

The transitions are a predicate

$$\psi_R(v_0, v_1, v'_0, v'_1)$$

- Over four Boolean $\{0, 1\}$ variables

Example: Symbolic Representation



Symbolic transitions:

$$\begin{array}{ll} & (v_0 = 0 \land v_1 = 0 \land v_0' = 0 \land v_1' = 1) \\ \lor & (v_0 = 0 \land v_1 = 1 \land v_0' = 1 \land v_1' = 0) \\ \lor & (v_0 = 1 \land v_1 = 0 \land v_0' = 1 \land v_1' = 1) \\ \lor & (v_0 = 1 \land v_1 = 1 \land v_0' = 0 \land v_1' = 0) \end{array}$$

Initial state: $v_0 = 0 \land v_0 = 1$

The transitions are a predicate

$$\psi_R(v_0, v_1, v_0', v_1')$$

- Over four Boolean $\{0, 1\}$ variables
- Variables completely determine state of system

Same for the initial state: $\psi_I(v_0, v_1)$

Fixpoints

Let $\tau : 2^S \mapsto 2^S$ be a predicate transfomer

- τ is **monotonic** iff $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$
- A **fixpoint** of τ is a predicate (set) Z where $\tau(Z) = Z$
- A **least fixpoint** of τ, written $\mu Z.\tau(Z)$, is:
  1. A fixpoint of τ, so $\tau(\mu Z.\tau(Z)) = Z$
  2. A subset of any other fixpoint
- A **greatest fixpoint** of τ, written $\nu Z.\tau(Z)$, is:
  1. A fixpoint of τ, so $\tau(\nu Z.\tau(Z)) = Z$
  2. A superset of any other fixpoint

Computing Fixpoints

We have a simple algorithm that gives us fixpoints

Computing Fixpoints

We have a simple algorithm that gives us fixpoints

$$
\begin{array}{l}
\textbf{function } \textit{lfp}(\tau) \; \{ \\
\quad Q := \textit{false}; \\
\quad Q' := \tau(Q); \\
\quad \textbf{while}(Q \neq Q') \; \{ \\
\qquad Q := Q'; \\
\qquad Q := \tau(Q'); \\
\quad \} \\
\quad \textbf{return } Q; \\
\}
\end{array}
$$

Computing Fixpoints

We have a simple algorithm that gives us fixpoints

```
function lfp(τ) {
  Q := false;
  Q′ := τ(Q);
  while(Q ≠ Q′) {
    Q := Q′;
    Q := τ(Q′);
  }
  return Q;
}
```

```
function gfp(τ) {
  Q := true;
  Q′ := τ(Q);
  while(Q ≠ Q′) {
    Q := Q′;
    Q := τ(Q′);
  }
  return Q;
}
```

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**
- Greatest fixpoints correspond to **global assertions**

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**
- Greatest fixpoints correspond to **global assertions**

Identify a CTL formula f with the predicate $\{s \in S \mid M, s \models f\}$

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**
- Greatest fixpoints correspond to **global assertions**

Identify a CTL formula f with the predicate $\{s \in S \mid M, s \models f\}$

Our "base" operator is **EX** ϕ, given by the predicate transformer:

$$\tau(\mathbf{v}) = \exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')$$

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**
- Greatest fixpoints correspond to **global assertions**

Identify a CTL formula f with the predicate $\{s \in S \mid M, s \models f\}$

Our "base" operator is **EX** ϕ, given by the predicate transformer:

$$\tau(\mathbf{v}) = \exists \mathbf{v}'.\phi(\mathbf{v}') \land R(\mathbf{v}, \mathbf{v}')$$

Then we define a sufficient set of operators using fixpoints:

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**
- Greatest fixpoints correspond to **global assertions**

Identify a CTL formula f with the predicate $\{s \in S \mid M, s \models f\}$

Our “base” operator is **EX** ϕ, given by the predicate transformer:

$$\tau(\mathbf{v}) = \exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')$$

Then we define a sufficient set of operators using fixpoints:

- **EG** $\phi = \nu Z.\phi \wedge$ **EX** Z

Fixpoint Semantics of CTL

We can define the semantics of CTL in terms of fixpoints and predicate transformers

- Least fixpoints correspond to **eventualities**
- Greatest fixpoints correspond to **global assertions**

Identify a CTL formula f with the predicate $\{s \in S \mid M, s \models f\}$

Our "base" operator is **EX** ϕ, given by the predicate transformer:

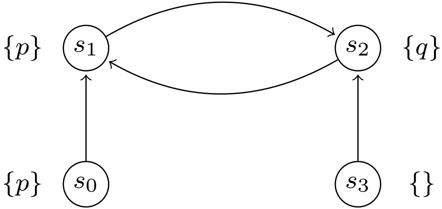
$$\tau(\mathbf{v}) = \exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')$$

Then we define a sufficient set of operators using fixpoints:

- **EG** $\phi = \nu Z.\phi \wedge \textbf{EX}\ Z$
- **E** $(\phi_1\ \textbf{U}\ \phi_2) = \mu Z.\phi_2 \vee (\phi_1 \wedge \textbf{EX}\ Z)$

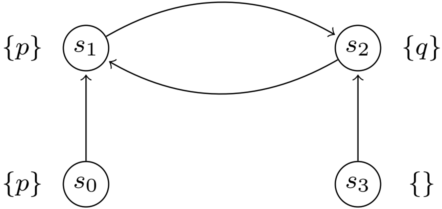
Example: **E** $(p$ **U** $q)$

$$\tau(Z) = q \vee (p \wedge \textbf{EX}\ Z)$$



Example: **E** $(p$ **U** $q)$

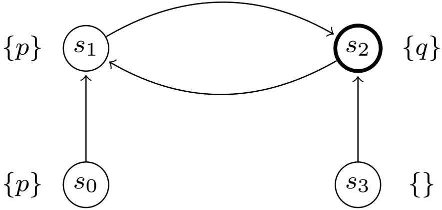
$$\tau(Z) = q \vee (p \wedge \textbf{EX}\ Z)$$



First compute $\tau($*false*$) = \tau(\varnothing)$

Example: **E** $(p$ **U** $q)$

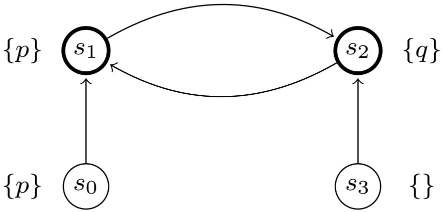
$$\tau(Z) = q \vee (p \wedge \textbf{EX}\ Z)$$



Then $\tau^1(\textit{false}) = \tau(\{s_2\})$

Example: **E** $(p \; \mathbf{U} \; q)$

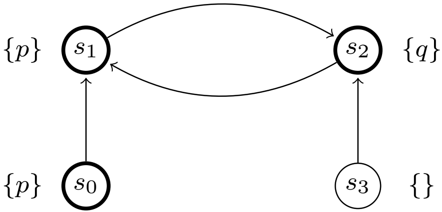
$$\tau(Z) = q \vee (p \wedge \mathbf{EX}\; Z)$$



Then $\tau^2(\textit{false}) = \tau(\{s_1, s_2\})$

Example: **E** $(p$ **U** $q)$

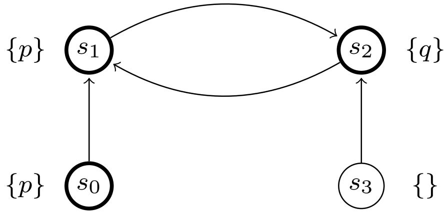
$$\tau(Z) = q \vee (p \wedge \textbf{EX}\ Z)$$



Then $\tau^3(\textit{false}) = \tau(\{s_0, s_1, s_2\})$

Example: **E** $(p$ **U** $q)$

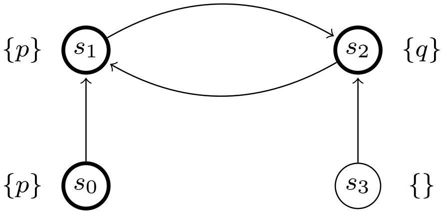
$$\tau(Z) = q \vee (p \wedge \textbf{EX}\ Z)$$



Then $\tau^4(\textit{false}) = \tau(\{s_0, s_1, s_2\}) = \tau^3(\textit{false})$

Example: **E** $(p$ **U** $q)$

$$\tau(Z) = q \vee (p \wedge \textbf{EX}\ Z)$$



Then $\tau^4(\textit{false}) = \tau(\{s_0, s_1, s_2\}) = \tau^3(\textit{false})$

We've reached the fixpoint $\mu Z.\tau(Z)$

Symbolic Model Checking (**EX**)

Checking **EX** ϕ is fairly straightforward

Symbolic Model Checking (**EX**)

Checking **EX** ϕ is fairly straightforward

Recall: We want to know if an initial state I satisfies **EX** ϕ

Symbolic Model Checking (**EX**)

Checking **EX** ϕ is fairly straightforward

Recall: We want to know if an initial state I satisfies **EX** ϕ

Our predicate transformer was: $\exists \mathbf{v}'.\phi(\mathbf{v}') \land R(\mathbf{v}, \mathbf{v}')$

Symbolic Model Checking (**EX**)

Checking **EX** ϕ is fairly straightforward

Recall: We want to know if an initial state I satisfies **EX** ϕ

Our predicate transformer was: $\exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')$

Then we check that the following formula is satisfiable:

$$\psi_I(\mathbf{v}) \wedge (\exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}'))$$

Symbolic Model Checking (**EX**)

Checking **EX** ϕ is fairly straightforward

Recall: We want to know if an initial state I satisfies **EX** ϕ

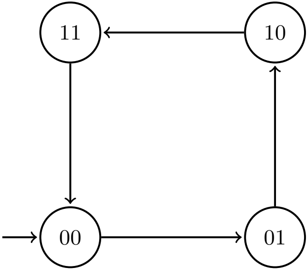
Our predicate transformer was: $\exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}')$

Then we check that the following formula is satisfiable:

$$\psi_I(\mathbf{v}) \wedge (\exists \mathbf{v}'.\phi(\mathbf{v}') \wedge R(\mathbf{v}, \mathbf{v}'))$$

If it is, then the corresponding set is non-empty, and ϕ holds

Symbolic Model Checking (**EX**): Example

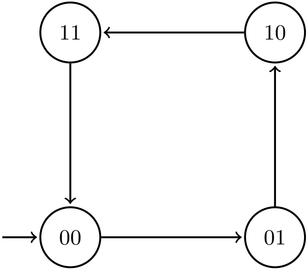


Suppose we want to check **EX** $v_0 = 1$

$$\psi_I(v_0, v_1) \Leftrightarrow v_0 = 0 \wedge v_1 = 0$$

$$\begin{aligned} \psi_R(v_0, v_1, v_0', v_1') \Leftrightarrow & \\ & (v_0 = 0 \wedge v_1 = 0 \wedge v_0' = 0 \wedge v_1' = 1) \\ \vee & (v_0 = 0 \wedge v_1 = 1 \wedge v_0' = 1 \wedge v_1' = 0) \\ \vee & (v_0 = 1 \wedge v_1 = 0 \wedge v_0' = 1 \wedge v_1' = 1) \\ \vee & (v_0 = 1 \wedge v_1 = 1 \wedge v_0' = 0 \wedge v_1' = 0) \end{aligned}$$

Symbolic Model Checking (**EX**): Example



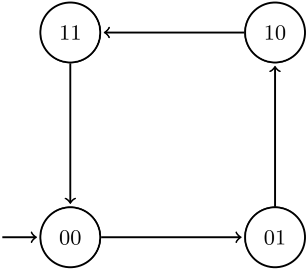
$\psi_I(v_0, v_1) \Leftrightarrow v_0 = 0 \land v_1 = 0$

$$\begin{aligned}
\psi_R(v_0, v_1, v_0', v_1') \Leftrightarrow & \\
& (v_0 = 0 \land v_1 = 0 \land v_0' = 0 \land v_1' = 1) \\
\lor\ & (v_0 = 0 \land v_1 = 1 \land v_0' = 1 \land v_1' = 0) \\
\lor\ & (v_0 = 1 \land v_1 = 0 \land v_0' = 1 \land v_1' = 1) \\
\lor\ & (v_0 = 1 \land v_1 = 1 \land v_0' = 0 \land v_1' = 0)
\end{aligned}$$

Suppose we want to check **EX** $v_0 = 1$

We apply the transformer for **EX** :

Symbolic Model Checking (**EX**): Example



$\psi_I(v_0, v_1) \Leftrightarrow v_0 = 0 \wedge v_1 = 0$

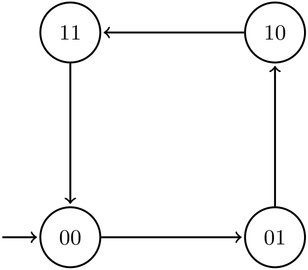
$$\begin{aligned}\psi_R(v_0, v_1, v_0', v_1') \Leftrightarrow \\ &(v_0 = 0 \wedge v_1 = 0 \wedge v_0' = 0 \wedge v_1' = 1) \\ \vee\ &(v_0 = 0 \wedge v_1 = 1 \wedge v_0' = 1 \wedge v_1' = 0) \\ \vee\ &(v_0 = 1 \wedge v_1 = 0 \wedge v_0' = 1 \wedge v_1' = 1) \\ \vee\ &(v_0 = 1 \wedge v_1 = 1 \wedge v_0' = 0 \wedge v_1' = 0)\end{aligned}$$

Suppose we want to check **EX** $v_0 = 1$

We apply the transformer for **EX** :

$$\exists v_0', v_1'. v_0' = 1 \wedge \psi_R(v_0, v_1, v_0', v_1')$$

Symbolic Model Checking (**EX**): Example



$\psi_I(v_0, v_1) \Leftrightarrow v_0 = 0 \wedge v_1 = 0$

$$
\begin{aligned}
\psi_R(v_0, v_1, v_0', v_1') \Leftrightarrow \\
& (v_0 = 0 \wedge v_1 = 0 \wedge v_0' = 0 \wedge v_1' = 1) \\
\vee \; & (v_0 = 0 \wedge v_1 = 1 \wedge v_0' = 1 \wedge v_1' = 0) \\
\vee \; & (v_0 = 1 \wedge v_1 = 0 \wedge v_0' = 1 \wedge v_1' = 1) \\
\vee \; & (v_0 = 1 \wedge v_1 = 1 \wedge v_0' = 0 \wedge v_1' = 0)
\end{aligned}
$$

Suppose we want to check **EX** $v_0 = 1$

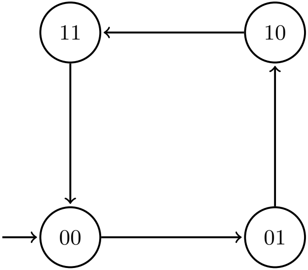
We apply the transformer for **EX** :

$$\exists v_0', v_1'. v_0' = 1 \wedge \psi_R(v_0, v_1, v_0', v_1')$$

Then conjoin the initial states:

$$
\begin{aligned}
v_0 = 0 \wedge v_1 = 0 \wedge \\
& \exists v_0', v_1'. v_0' = 1 \wedge \psi_R(v_0, v_1, v_0', v_1')
\end{aligned}
$$

Symbolic Model Checking (**EX**): Example



$\psi_I(v_0, v_1) \Leftrightarrow v_0 = 0 \land v_1 = 0$

$$\psi_R(v_0, v_1, v_0', v_1') \Leftrightarrow$$
$$(v_0 = 0 \land v_1 = 0 \land v_0' = 0 \land v_1' = 1)$$
$$\lor\ (v_0 = 0 \land v_1 = 1 \land v_0' = 1 \land v_1' = 0)$$
$$\lor\ (v_0 = 1 \land v_1 = 0 \land v_0' = 1 \land v_1' = 1)$$
$$\lor\ (v_0 = 1 \land v_1 = 1 \land v_0' = 0 \land v_1' = 0)$$

Suppose we want to check **EX** $v_0 = 1$

We apply the transformer for **EX** :

$$\exists v_0', v_1'. v_0' = 1 \land \psi_R(v_0, v_1, v_0', v_1')$$

Then conjoin the initial states:

$$v_0 = 0 \land v_1 = 0 \land$$
$$\exists v_0', v_1'. v_0' = 1 \land \psi_R(v_0, v_1, v_0', v_1')$$

This formula is *false*, so there are no states that satisfy

Symbolic Model Checking (**EG**)

We have that **EG** $\phi = \nu Z.\phi \wedge$ **EX** Z

Symbolic Model Checking (**EG**)

We have that **EG** $\phi = \nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

Symbolic Model Checking (**EG**)

We have that **EG** ϕ = $\nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of τ = $\nu Z.\phi \wedge$ **EX** Z

Symbolic Model Checking (**EG**)

We have that **EG** $\phi = \nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of $\tau = \nu Z.\phi \wedge$ **EX** Z
2. Conjoin ψ_I

Symbolic Model Checking (**EG**)

We have that **EG** ϕ = $\nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of τ = $\nu Z.\phi \wedge$ **EX** Z
2. Conjoin ψ_I
3. Check for satisfiability

Symbolic Model Checking (**EG**)

We have that **EG** $\phi = \nu Z.\phi \land$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of $\tau = \nu Z.\phi \land$ **EX** Z
2. Conjoin ψ_I
3. Check for satisfiability

We know that we can compute greatest fixpoints by:

Symbolic Model Checking (**EG**)

We have that **EG** $\phi = \nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of $\tau = \nu Z.\phi \wedge$ **EX** Z
2. Conjoin ψ_I
3. Check for satisfiability

We know that we can compute greatest fixpoints by:

1. Applying the predicate transformer to *true*

Symbolic Model Checking (**EG**)

We have that **EG** $\phi = \nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of $\tau = \nu Z.\phi \wedge$ **EX** Z
2. Conjoin ψ_I
3. Check for satisfiability

We know that we can compute greatest fixpoints by:

1. Applying the predicate transformer to *true*
2. Repeating, until the predicate doesn't change

Symbolic Model Checking (**EG**)

We have that **EG** ϕ = $\nu Z.\phi \wedge$ **EX** Z

So to check **EG** ϕ:

1. Find the fixpoint of τ = $\nu Z.\phi \wedge$ **EX** Z
2. Conjoin ψ_I
3. Check for satisfiability

We know that we can compute greatest fixpoints by:

1. Applying the predicate transformer to *true*
2. Repeating, until the predicate doesn't change

But before we can do this, must show $\nu Z.\phi \wedge$ **EX** Z is monotonic

Symbolic Model Checking (**E** $(\phi_1$ **U** $\phi_2))$

We have that **E** $(\phi_1$ **U** $\phi_2) = \mu Z.\phi_2 \vee (\phi_1 \wedge$ **EX** $Z)$

Symbolic Model Checking (**E** $(\phi_1$ **U** $\phi_2))$

We have that **E** $(\phi_1$ **U** $\phi_2) = \mu Z.\phi_2 \vee (\phi_1 \wedge$ **EX** $Z)$

We proceed exactly as we did for **EG** , but compute *lfp* instead

Symbolic Model Checking (**E** $(\phi_1$ **U** $\phi_2))$

We have that **E** $(\phi_1$ **U** $\phi_2)$ = $\mu Z.\phi_2 \vee (\phi_1 \wedge$ **EX** $Z)$

We proceed exactly as we did for **EG** , but compute *lfp* instead

Notice: this algorithm is very similar to the explicit-state one

Symbolic Model Checking (**E** $(\phi_1$ **U** $\phi_2))$

We have that **E** $(\phi_1$ **U** $\phi_2)$ = $\mu Z.\phi_2 \vee (\phi_1 \wedge$ **EX** $Z)$

We proceed exactly as we did for **EG** , but compute *lfp* instead

Notice: this algorithm is very similar to the explicit-state one

1. Compute the set of states satisfying the CTL formula

Symbolic Model Checking (**E** $(\phi_1$ **U** $\phi_2))$

We have that **E** $(\phi_1$ **U** $\phi_2) = \mu Z.\phi_2 \vee (\phi_1 \wedge$ **EX** $Z)$

We proceed exactly as we did for **EG** , but compute *lfp* instead

Notice: this algorithm is very similar to the explicit-state one

1. Compute the set of states satisfying the CTL formula
2. Check that an initial state is in the result

Symbolic Model Checking (**E** (ϕ_1 **U** ϕ_2))

We have that **E** $(\phi_1$ **U** $\phi_2)$ = $\mu Z.\phi_2 \vee (\phi_1 \wedge$ **EX** $Z)$

We proceed exactly as we did for **EG** , but compute *lfp* instead

Notice: this algorithm is very similar to the explicit-state one

1. Compute the set of states satisfying the CTL formula
2. Check that an initial state is in the result

But what have we gained by doing it this way?

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

- Internal nodes corresponding to variables $x_1, \ldots, x_n$

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

- Internal nodes corresponding to variables $x_1, \ldots, x_n$
- Leaf nodes corresponding to Boolean values of $\phi(x_1, \ldots, x_n)$

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

- Internal nodes corresponding to variables $x_1, \ldots, x_n$
- Leaf nodes corresponding to Boolean values of $\phi(x_1, \ldots, x_n)$
- Edges corresponding to Boolean values of x_i

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

- Internal nodes corresponding to variables $x_1, \ldots, x_n$
- Leaf nodes corresponding to Boolean values of $\phi(x_1, \ldots, x_n)$
- Edges corresponding to Boolean values of x_i

Given a fixed ordering of $x_1, \ldots, x_n$, these are **canonical**

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

- Internal nodes corresponding to variables $x_1, \ldots, x_n$
- Leaf nodes corresponding to Boolean values of $\phi(x_1, \ldots, x_n)$
- Edges corresponding to Boolean values of x_i

Given a fixed ordering of $x_1, \ldots, x_n$, these are **canonical**

- Isomorphic trees $T_1, T_2 \implies$ Equivalent predicates ϕ_1, ϕ_2

Efficient Propositional Encodings

Given a predicate $\phi(x_1, \ldots, x_n) \mapsto \{0, 1\}$

An **ordered binary decision tree** consists of:

- Internal nodes corresponding to variables $x_1, \ldots, x_n$
- Leaf nodes corresponding to Boolean values of $\phi(x_1, \ldots, x_n)$
- Edges corresponding to Boolean values of x_i

Given a fixed ordering of $x_1, \ldots, x_n$, these are **canonical**

- Isomorphic trees $T_1, T_2 \implies$ Equivalent predicates ϕ_1, ϕ_2

This gives us an easy way to test fixpoints

Ordered Binary Decision Trees

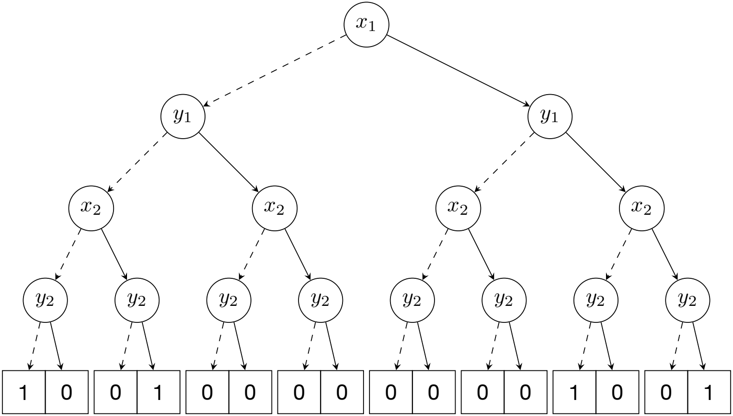
Consider the two-bit comparator:

$$\phi(x_1, x_2, y_1, y_2) = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$$

Ordered Binary Decision Trees

Consider the two-bit comparator:

$$\phi(x_1, x_2, y_1, y_2) = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$$



More efficient representations

Ordered binary trees are canonical, but as large as truth tables

More efficient representations

Ordered binary trees are canonical, but as large as truth tables

Idea: remove redundant information

More efficient representations

Ordered binary trees are canonical, but as large as truth tables

Idea: remove redundant information

- Merge duplicate leaves: only one terminal with each label

More efficient representations

Ordered binary trees are canonical, but as large as truth tables

Idea: remove redundant information

- Merge duplicate leaves: only one terminal with each label
- Eliminate redundant internal nodes: if both edges give same result, redirect incoming edges to successors

More efficient representations

Ordered binary trees are canonical, but as large as truth tables

Idea: remove redundant information

- Merge duplicate leaves: only one terminal with each label
- Eliminate redundant internal nodes: if both edges give same result, redirect incoming edges to successors
- Remove duplicate internal nodes: two nodes for same variable, whose successors give same result

More efficient representations

Ordered binary trees are canonical, but as large as truth tables

Idea: remove redundant information

- Merge duplicate leaves: only one terminal with each label
- Eliminate redundant internal nodes: if both edges give same result, redirect incoming edges to successors
- Remove duplicate internal nodes: two nodes for same variable, whose successors give same result

The result is no longer a tree, but a DAG

More efficient representations

Ordered binary trees are canonical, but as large as truth tables

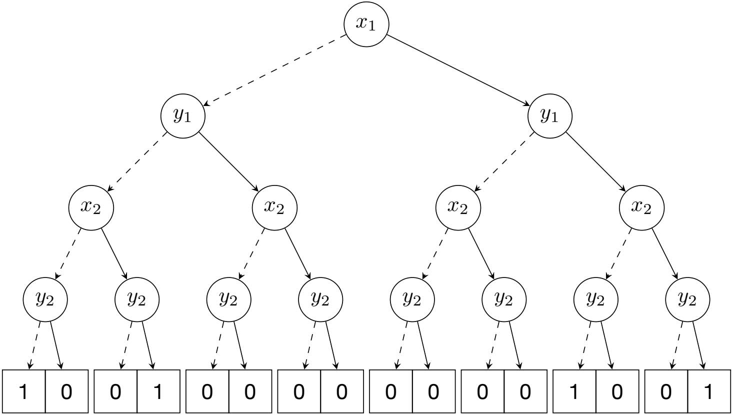
Idea: remove redundant information

- Merge duplicate leaves: only one terminal with each label
- Eliminate redundant internal nodes: if both edges give same result, redirect incoming edges to successors
- Remove duplicate internal nodes: two nodes for same variable, whose successors give same result

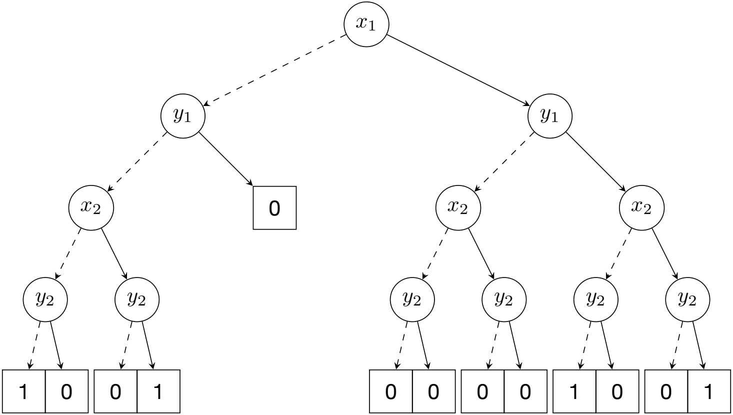
The result is no longer a tree, but a DAG

These are called **Ordered Binary Decision Diagrams** (OBDDs)

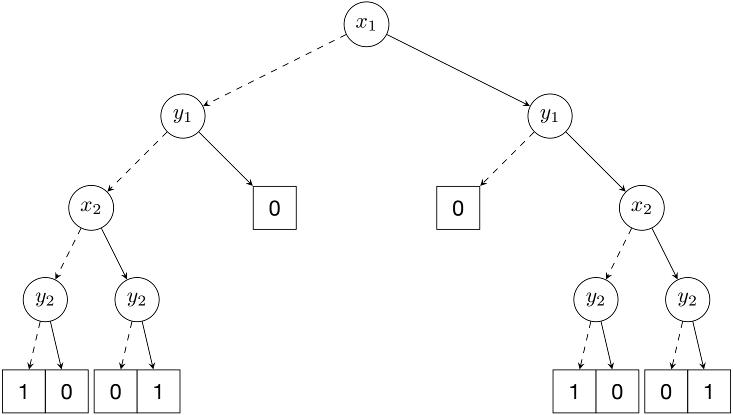
Ordered Binary Decision Trees



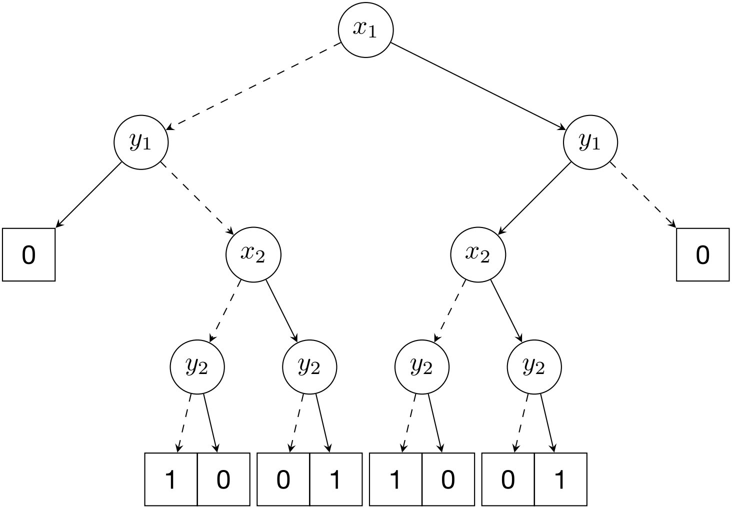
Ordered Binary Decision Trees



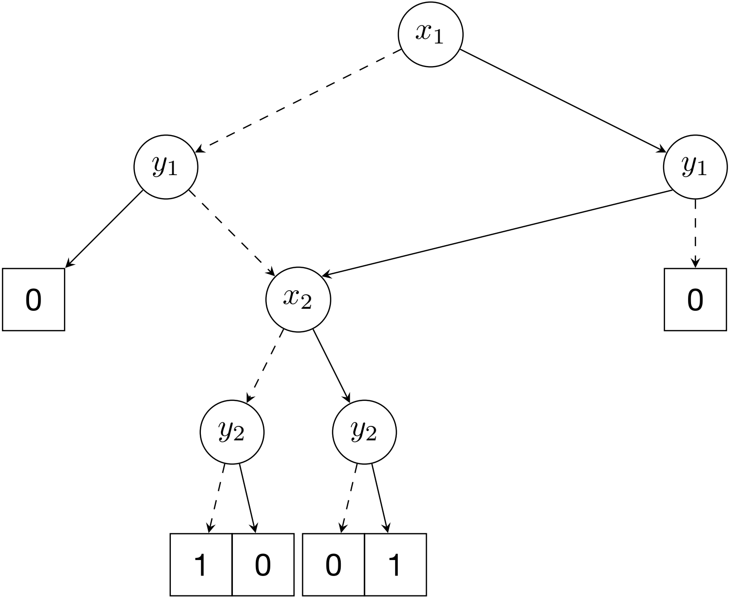
Ordered Binary Decision Trees



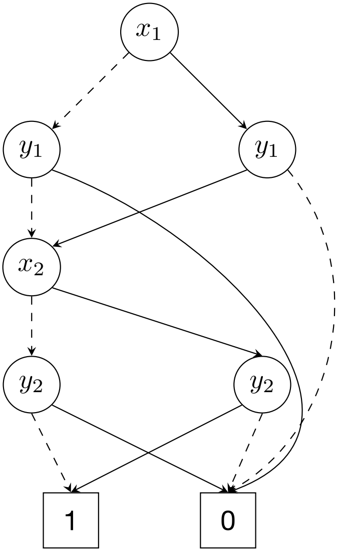
Ordered Binary Decision Trees



Ordered Binary Decision Trees



Ordered Binary Decision Diagrams



OBDDs and Ordering

Variable ordering matters for OBDD size

OBDDs and Ordering

Variable ordering matters for OBDD size

For an n-bit comparator:

OBDDs and Ordering

Variable ordering matters for OBDD size

For an n-bit comparator:

- $x_1, y_1, \ldots, x_n, y_n$: $3n + 2$ vertices

OBDDs and Ordering

Variable ordering matters for OBDD size

For an n-bit comparator:

- $x_1, y_1, \ldots, x_n, y_n$: $3n + 2$ vertices
- $x_1, x_2, \ldots, y_{n-1}, y_n$: $3 \times 2^n - 1$ vertices

OBDDs and Ordering

Variable ordering matters for OBDD size

For an n-bit comparator:

- $x_1, y_1, \ldots, x_n, y_n$: $3n + 2$ vertices
- $x_1, x_2, \ldots, y_{n-1}, y_n$: $3 \times 2^n - 1$ vertices

Some predicates have exponential size for **any** ordering

OBDDs and Ordering

Variable ordering matters for OBDD size

For an n-bit comparator:

- $x_1, y_1, \ldots, x_n, y_n$: $3n + 2$ vertices
- $x_1, x_2, \ldots, y_{n-1}, y_n$: $3 \times 2^n - 1$ vertices

Some predicates have exponential size for **any** ordering

OBDDs **typically** introduce drastic savings on time and space

OBDDs and Ordering

Variable ordering matters for OBDD size

For an n-bit comparator:

- $x_1, y_1, \ldots, x_n, y_n$: $3n + 2$ vertices
- $x_1, x_2, \ldots, y_{n-1}, y_n$: $3 \times 2^n - 1$ vertices

Some predicates have exponential size for **any** ordering

OBDDs **typically** introduce drastic savings on time and space

- $\sim$ order of magnitude savings on many real examples

Introduction to Spin

Spin is a prominent model checking tool & simulator

Introduction to Spin

Spin is a prominent model checking tool & simulator

- **S**imple **P**romela **In**terpreter

Introduction to Spin

Spin is a prominent model checking tool & simulator

- **S**imple **P**romela **In**terpreter
- Gerard Holzmann at Bell Labs' Unix group, starting c. 1980

Introduction to Spin

Spin is a prominent model checking tool & simulator

- **S**imple **P**romela **In**terpreter
- Gerard Holzmann at Bell Labs' Unix group, starting c. 1980
- Applied to Mars Rovers, Deep Impact, Cassini, Toyota control software, medical devices, …

Introduction to Spin

Spin is a prominent model checking tool & simulator

- **S**imple **P**romela **In**terpreter
- Gerard Holzmann at Bell Labs' Unix group, starting c. 1980
- Applied to Mars Rovers, Deep Impact, Cassini, Toyota control software, medical devices, ...
- Accepts LTL, converts to Buchi automata

Introduction to Spin

Spin is a prominent model checking tool & simulator

- **S**imple **P**romela **In**terpreter
- Gerard Holzmann at Bell Labs' Unix group, starting c. 1980
- Applied to Mars Rovers, Deep Impact, Cassini, Toyota control software, medical devices, …
- Accepts LTL, converts to Buchi automata
- Implements partial order reduction, on-the-fly checking, state compression, BDD-like representations

Introduction to Spin

Spin is a prominent model checking tool & simulator

- **S**imple **P**romela **In**terpreter
- Gerard Holzmann at Bell Labs' Unix group, starting c. 1980
- Applied to Mars Rovers, Deep Impact, Cassini, Toyota control software, medical devices, …
- Accepts LTL, converts to Buchi automata
- Implements partial order reduction, on-the-fly checking, state compression, BDD-like representations

Tool you'll use for the final homework

Why Spin?

Mature implementation

1. Under development since 1980, freely-available since 1991
2. Winner of ACM Software Systems Award (others include Unix, TCP/IP, GCC, LLVM, `make`, ...)
3. Lots of real applications and successes (see previous slide)
4. Several projects extend Spin with frontends and other utilities
5. Based on concepts we've covered: ω-automata and LTL

Why Spin?

Mature implementation

1. Under development since 1980, freely-available since 1991
2. Winner of ACM Software Systems Award (others include Unix, TCP/IP, GCC, LLVM, `make`, ...)
3. Lots of real applications and successes (see previous slide)
4. Several projects extend Spin with frontends and other utilities
5. Based on concepts we've covered: ω-automata and LTL

Good documentation

1. Several books (see Holzmann 2003, Ben-Ari 2008)
2. Annual workshops since 1995
3. Used extensively in other courses
4. Google turns up many hits when looking for specific info

Spin

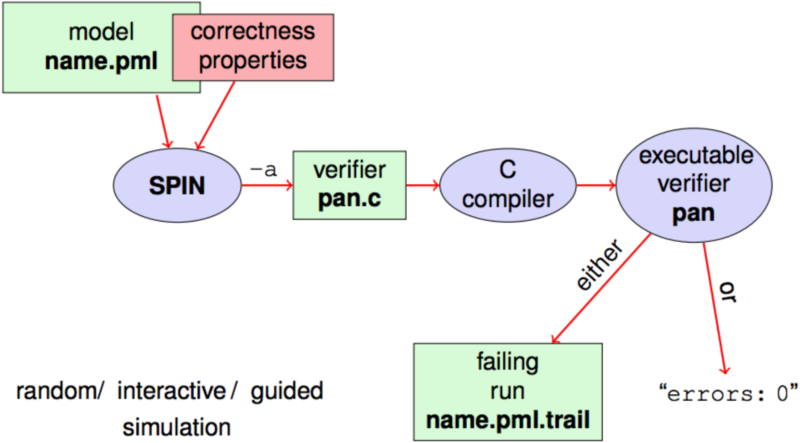


Image credit: Bernhard Beckert and Vladimir Klebanov

Promela

Process **Me**ta **La**nguage

Promela

Process **Me**ta **La**nguage

Modeling language used by Spin

Promela

Process **Me**ta **La**nguage

Modeling language used by Spin

- Just a few statement types

Promela

Process **Me**ta **La**nguage

Modeling language used by Spin

- Just a few statement types
- Multi-threaded interleaving semantics

Promela

Process **Me**ta **La**nguage

Modeling language used by Spin

- Just a few statement types
- Multi-threaded interleaving semantics
- Synchronization and message passing facilities

Promela

Process **Me**ta **La**nguage

Modeling language used by Spin

- Just a few statement types
- Multi-threaded interleaving semantics
- Synchronization and message passing facilities
- Support for finite data structures

Promela

Process **Me**ta **La**nguage

Modeling language used by Spin

- Just a few statement types
- Multi-threaded interleaving semantics
- Synchronization and message passing facilities
- Support for finite data structures

Not an implementation language

- No libraries
- No pointers
- No standard input
- ...

Promela: Hello World

```
active proctype P() {
  printf("Hello world!");
}
```

Promela: Hello World

```
active proctype P() {
  printf("Hello world!");
}
```

1. `proctype` declares a new process named `P`

Promela: Hello World

```
active proctype P() {
  printf("Hello world!");
}
```

1. `proctype` declares a new process named `P`
2. Promela programs consist of a finite set of concurrent processes

Promela: Hello World

```
active proctype P() {
  printf("Hello world!");
}
```

1. `proctype` declares a new process named `P`
2. Promela programs consist of a finite set of concurrent processes
3. `active` denotes that `P` is run immediately

Promela: Hello World

```
active proctype P() {
  printf("Hello world!");
}
```

1. `proctype` declares a new process named `P`
2. Promela programs consist of a finite set of concurrent processes
3. `active` denotes that `P` is run immediately
4. C-like `printf` for debugging

Promela: Hello World

```
active proctype P() {
  printf("Hello world!");
}
```

1. `proctype` declares a new process named `P`
2. Promela programs consist of a finite set of concurrent processes
3. `active` denotes that `P` is run immediately
4. C-like `printf` for debugging

To run:

```
> spin hellow.pml
 Hello world!
```

Data types

```
bit                         {0,1}
bool                        {0,1}
byte                      [0..255]
short              [-2^15..2^15-1]
int                [-2^31..2^31-1]

#define N 10
byte array[N];
array[0] = array[1];

typedef Msg {
  byte header[16];
  int payload;
}
Msg x;
x.payload = 1;
```

Data types

```
bit                        {0,1}
bool                       {0,1}
byte                     [0..255]
short             [-2^15..2^15-1]
int               [-2^31..2^31-1]

#define N 10
byte array[N];
array[0] = array[1];

typedef Msg {
  byte header[16];
  int payload;
}
Msg x;
x.payload = 1;
```

Basic types

Data types

```
bit                          {0,1}
bool                         {0,1}
byte                      [0..255]
short              [-2^15..2^15-1]
int                [-2^31..2^31-1]

#define N 10
byte array[N];
array[0] = array[1];

typedef Msg {
  byte header[16];
  int payload;
}
Msg x;
x.payload = 1;
```

Basic types

C-style preprocessor directives

Data types

```
bit                          {0,1}
bool                         {0,1}
byte                      [0..255]
short              [-2^15..2^15-1]
int                [-2^31..2^31-1]

#define N 10
byte array[N];
array[0] = array[1];

typedef Msg {
  byte header[16];
  int payload;
}
Msg x;
x.payload = 1;
```

Basic types

C-style preprocessor directives
array declarations

Data types

```
bit                       {0,1}
bool                      {0,1}
byte                   [0..255]
short          [-2^15..2^15-1]
int            [-2^31..2^31-1]

#define N 10
byte array[N];
array[0] = array[1];

typedef Msg {
  byte header[16];
  int payload;
}
Msg x;
x.payload = 1;
```

Basic types

C-style preprocessor directives
array declarations
array access

Data types

```
bit                          {0,1}
bool                         {0,1}
byte                       [0..255]
short               [-2^15..2^15-1]
int                 [-2^31..2^31-1]

#define N 10
byte array[N];
array[0] = array[1];

typedef Msg {
  byte header[16];
  int payload;
}
Msg x;
x.payload = 1;
```

Basic types

C-style preprocessor directives
array declarations
array access

structured data

Basic Statements

Expressions are statements

Basic Statements

Expressions are statements

- No side effects

Basic Statements

Expressions are statements

- No side effects
- Standard arithmetic operations

Basic Statements

Expressions are statements

- No side effects
- Standard arithmetic operations
- Conditional expression: `(x >= 0 -> x : -x)`

Basic Statements

Expressions are statements

- No side effects
- Standard arithmetic operations
- Conditional expression: `(x >= 0 -> x : -x)`

Assignments have the usual meaning

- `x = x * 5;`

Basic Statements

Expressions are statements

- No side effects
- Standard arithmetic operations
- Conditional expression: `(x >= 0 -> x : -x)`

Assignments have the usual meaning

- `x = x * 5;`
- Promela supports increment `++` and decrement `--` assignments

Basic Statements

Expressions are statements

- No side effects
- Standard arithmetic operations
- Conditional expression: `(x >= 0 -> x : -x)`

Assignments have the usual meaning

- `x = x * 5;`
- Promela supports increment `++` and decrement `--` assignments

The no-op statment `skip` is supported

Basic Statements

Expressions are statements

- No side effects
- Standard arithmetic operations
- Conditional expression: `(x >= 0 -> x : -x)`

Assignments have the usual meaning

- `x = x * 5;`
- Promela supports increment `++` and decrement `--` assignments

The no-op statment `skip` is supported

Control transfer via `goto label` is supported

Compound Statements

Sequential composition via the usual semicolon `;` syntax

Compound Statements

Sequential composition via the usual semicolon `;` syntax

- The arrow `->` can be used interchangably with `;`

Selection via the computing `if..fi` statement

Compound Statements

Sequential composition via the usual semicolon `;` syntax

- The arrow `->` can be used interchangably with `;`

Selection via the computing `if..fi` statement

- Expressions guard each case

Compound Statements

Sequential composition via the usual semicolon `;` syntax

- The arrow `->` can be used interchangably with `;`

Selection via the computing `if..fi` statement

- Expressions guard each case
- Can be non-deterministic by omitting guard

Compound Statements

Sequential composition via the usual semicolon `;` syntax

- The arrow `->` can be used interchangably with `;`

Selection via the computing `if..fi` statement

- Expressions guard each case
- Can be non-deterministic by omitting guard

```
if
:: (a == b) -> state = state + 1
:: else -> state = state - 1
fi

if
:: x = 0
:: x = 1
fi
```

Blocking

All statements are either **blocked** or **enabled**

Blocking

All statements are either **blocked** or **enabled**

If an expression-statement evaluates to 0, then it is blocked

Blocking

All statements are either **blocked** or **enabled**

If an expression-statement evaluates to 0, then it is blocked

```
byte state = 1;

proctype A()
{ byte tmp;
  (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}

proctype B()
{ byte tmp;
  (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}

init
{ run A(); run B()
}
```

Repetition

Syntax for repetition is similar to `if .. fi`

Repetition

Syntax for repetition is similar to `if .. fi`

Keyword `do .. od` denote repetition block

Can also have non-deterministic behavior by omitting guards

Repetition

Syntax for repetition is similar to `if .. fi`

Keyword `do .. od` denote repetition block

Can also have non-deterministic behavior by omitting guards

```
proctype Euclid(int x, y)
  {
    do
    :: (x > y) -> x = x - y
    :: (x < y) -> y = y - x
    :: (x == y) -> break
    od;
  }
```

More on guards

```
:: guard -> command
```

When this appears in `if` or `do`:

More on guards

```
:: guard -> command
```

When this appears in `if` or `do`:

- `command` is optional: can write `:: guard;`

More on guards

```
:: guard -> command
```

When this appears in `if` or `do`:

- `command` is optional: can write `:: guard;`
- Guards can overlap: any alternative that is *true* is non-deterministically selected

More on guards

```
:: guard -> command
```

When this appears in `if` or `do`:

- `command` is optional: can write `:: guard;`
- Guards can overlap: any alternative that is *true* is non-deterministically selected
- When no guards are true, the statement (and process) block until one becomes true

Communication Channels

Processes can communicate by passing messages

Communication Channels

Processes can communicate by passing messages

- Asynchronously via a buffered FIFO queue

Communication Channels

Processes can communicate by passing messages

- Asynchronously via a buffered FIFO queue
- Synchronously via rendez-vous ports

Can declare an enumerated message type `mtype`

Communication Channels

Processes can communicate by passing messages

- Asynchronously via a buffered FIFO queue
- Synchronously via rendez-vous ports

Can declare an enumerated message type `mtype`

- One `mtype` per program

Communication Channels

Processes can communicate by passing messages

- Asynchronously via a buffered FIFO queue
- Synchronously via rendez-vous ports

Can declare an enumerated message type `mtype`

- One `mtype` per program
- Useful for abstract protocol specifications

Communication Channels

Processes can communicate by passing messages

- Asynchronously via a buffered FIFO queue
- Synchronously via rendez-vous ports

Can declare an enumerated message type `mtype`

- One `mtype` per program
- Useful for abstract protocol specifications

```
mtype = {ack, err, accept};

chan c1 = [16] of { mtype };          // store up to 16 messages
chan c2 = [16] of { int, mtype };     // two fields per message

// rendez-vous channel for synchronous communication
// size 0: can transmit but not store a message
chan port = [0] of { short };
```

Process Communications

Sending a message: `channel!expr`

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`
- If `channel` is full, statement blocks

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`
- If `channel` is full, statement blocks

Receiving a message: `channel?var`

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`
- If `channel` is full, statement blocks

Receiving a message: `channel?var`

- Can specify multiple fields with `channel?expr1,expr2`

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`
- If `channel` is full, statement blocks

Receiving a message: `channel?var`

- Can specify multiple fields with `channel?expr1,expr2`
- Reads the head of `channel` into `var`

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`
- If `channel` is full, statement blocks

Receiving a message: `channel?var`

- Can specify multiple fields with `channel?expr1,expr2`
- Reads the head of `channel` into `var`
- If `channel` is empty, statement blocks

Process Communications

Sending a message: `channel!expr`

- Can specify multiple fields with `channel!expr1,expr2`
- Appends the value of `expr` to the end of `channel`
- If `channel` is full, statement blocks

Receiving a message: `channel?var`

- Can specify multiple fields with `channel?expr1,expr2`
- Reads the head of `channel` into `var`
- If `channel` is empty, statement blocks

The expression `len(channel)` returns # of messages on `channel`

Channels: Example

```
#define msgtype 33

chan name = [0] of { byte, byte };

active proctype A()
{ name!msgtype,124;
  // synchronous channel, no second receive in B
  // process will block here forever
  name!msgtype,121;
}

active proctype B()
{ byte state;
  name?msgtype(state)
}
```

Atomicity

Basic statements execute atomically

- Assignments, expressions, `goto`, `skip`

Atomicity

Basic statements execute atomically

- Assignments, expressions, `goto`, `skip`

Guarded commands are **not atomic**

Atomicity

Basic statements execute atomically

- Assignments, expressions, `goto`, `skip`

Guarded commands are **not atomic**

```
int a, b, c;

active proctype P1() {
  a = 1; b = 5;
  if
  :: a != 0 -> c = b / a;  // this can be #div0!
  :: else -> c = b;
  fi
}

active proctype P2() {
  a = 0;
}
```

Atomicity

Use an `atomic` block to prevent bad interleavings

Atomicity

Use an `atomic` block to prevent bad interleavings

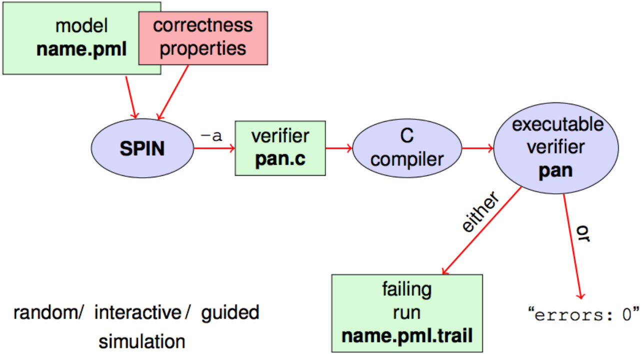
```
int a, b, c;

active proctype P1() {
  a = 1; b = 5;
  atomic {
    if
    :: a != 0 -> c = b / a;
    :: else -> c = b;
    fi
  }
}

active proctype P2() {
  a = 0;
}
```

Stating Correctness Properties: `assert`

Option 1: `assert` statements

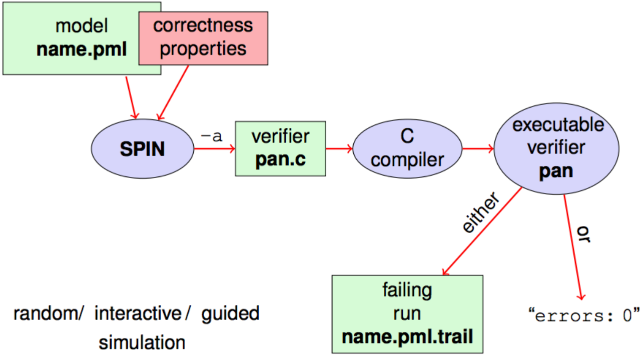
```
bool flag[2];
bool turn;
byte cnt = 0;

active [2] proctype user()
{
  flag[_pid] = true;
  turn = _pid;
  (flag[1-_pid] == false || turn == 1-_pid);

  cnt++;
crit: assert(cnt == 1);  // critical section
  cnt--;

  flag[_pid] = false;
}
```

Checking the property



random/ interactive / guided
simulation

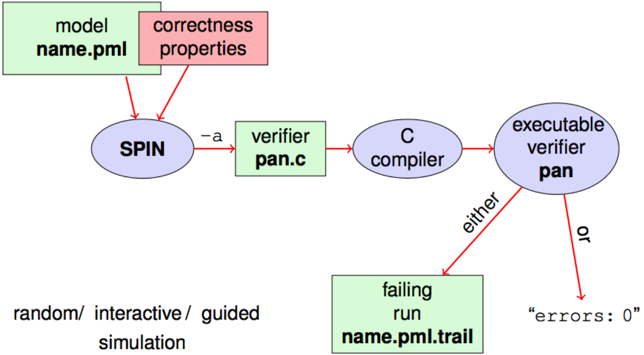
Checking the property



Step 1: Generate a verifier

```
> spin -a mutex.pml          // spin generates pan.c
```

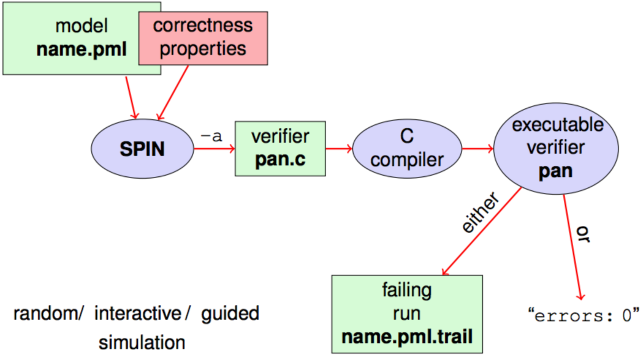
Checking the property



Step 2: Compile the verifier

```
> gcc -o pan pan.c        // output in pan
```

Checking the property



Step 3: Run the verifier to do exhaustive model checking

```
> ./pan
```

Verification Results

```
(Spin Version 6.4.5 -- 1 January 2016)
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  acceptance   cycles   - (not selected)
  invalid end states  +

State-vector 28 byte, depth reached 16, errors: 0
       56 states, stored
       21 states, matched
       77 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.003 equivalent memory usage for states
    0.292 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  128.730 total actual memory usage

unreached in proctype user
  (0 of 8 states)
```

Stating Correctness Properties: LTL

Option 2: Write an LTL formula

```
bool flag[2];
bool turn;
byte cnt = 0;

active [2] proctype user()
{
  flag[_pid] = true;
  turn = _pid;
  (flag[1-_pid] == false || turn == 1-_pid);

crit: skip;  // critical section

  flag[_pid] = false;
}

ltl mutex { [] (!p[0]@crit || !p[1]@crit) }
```

LTL in Spin

```
Grammar:
  ltl ::= opd | ( ltl ) | ltl binop ltl | unop ltl

Operands (opd):
  true, false, user-defined names starting with a lower-case letter,
  or embedded expressions inside curly braces, e.g.,: { a+b>n }.

Unary Operators (unop):
  []  (the temporal operator always)
  <>  (the temporal operator eventually)
  !   (the boolean operator for negation)

Binary Operators (binop):
  U   (the temporal operator strong until)
  W (the temporal operator weak until
  V   (the dual of U): (p V q) means !(!p U !q))
  &&  (the boolean operator for logical and)
  ||  (the boolean operator for logical or)
  /\  (alternative form of &&)
  \/  (alternative form of ||)
  ->  (the boolean operator for logical implication)
  <-> (the boolean operator for logical equivalence)
```

Counterexamples

Let's introduce the bug from the previous homework

```
bool flag[2];
bool turn;
byte cnt = 0;

active [2] proctype user()
{
  turn = _pid;
  flag[_pid] = true;
  (flag[1-_pid] == false || turn == 1-_pid);

crit: skip;  // critical section

  flag[_pid] = false;
}

ltl mutex { [] (!p[0]@crit || !p[1]@crit) }
```

Generating counterexmaples

```
> spin -a mutex.pml; gcc -o pan pan.c; ./pan
> spin -t -p -l mutex.pml

using statement merging
  1:  proc  1 (user:1) mutex.pml:8 (state 1)  [turn = _pid]
  2:  proc  0 (user:1) mutex.pml:8 (state 1)  [turn = _pid]
  3:  proc  0 (user:1) mutex.pml:9 (state 2)  [flag[_pid] = 1]
  4:  proc  0 (user:1) mutex.pml:10 (state 3) [(((flag[(1-_pid)]==0)||(
       turn==(1-_pid))))]
  5:  proc  1 (user:1) mutex.pml:9 (state 2)  [flag[_pid] = 1]
  6:  proc  1 (user:1) mutex.pml:10 (state 3) [(((flag[(1-_pid)]==0)||(
       turn==(1-_pid))))]
  7:  proc  1 (user:1) mutex.pml:12 (state 4) [cnt = (cnt+1)]
  8:  proc  1 (user:1) mutex.pml:13 (state 5) [assert((cnt==1))]
  9:  proc  0 (user:1) mutex.pml:12 (state 4) [cnt = (cnt+1)]
spin: mutex.pml:13, Error: assertion violated
spin: text of failed assertion: assert((cnt==1))
 10:  proc  0 (user:1) mutex.pml:13 (state 5) [assert((cnt==1))]
spin: trail ends after 10 steps
#processes: 2
    flag[0] = 1
    flag[1] = 1
    turn = 0
    cnt = 2
 10:  proc  1 (user:1) mutex.pml:14 (state 6)
 10:  proc  0 (user:1) mutex.pml:14 (state 6)
2 processes created
```

Generating counterexmaples

```
> spin -t -p -l mutex.pml
```

Generating counterexmaples

```
> spin -t -p -l mutex.pml
```

- Failed verification produces `mutex.pml.trail`

Generating counterexmaples

```
> spin -t -p -l mutex.pml
```

- Failed verification produces `mutex.pml.trail`
- `-t` option tells Spin to use `mutex.pml.trail` to guide simulation

Generating counterexmaples

```
> spin -t -p -l mutex.pml
```

- Failed verification produces `mutex.pml.trail`
- `-t` option tells Spin to use `mutex.pml.trail` to guide simulation
- Basically, inject the discovered fault into execution

Generating counterexmaples

```
> spin -t -p -l mutex.pml
```

- Failed verification produces `mutex.pml.trail`
- `-t` option tells Spin to use `mutex.pml.trail` to guide simulation
- Basically, inject the discovered fault into execution
- `-p` option prints all statements in the execution

Generating counterexmaples

```
> spin -t -p -l mutex.pml
```

- Failed verification produces `mutex.pml.trail`
- `-t` option tells Spin to use `mutex.pml.trail` to guide simulation
- Basically, inject the discovered fault into execution
- `-p` option prints all statements in the execution
- `-l` option prints the values of local variables

Next Lecture

Last assignment goes out today

Due at midnight on last day of classes

Next class: Software Model Checking