Where we are

We’ve covered algorithms for model checking Kripke structures

- Directly over transition graph (CTL)
- After converting to Buchi automata (LTL)
- Symbolically, representing states and transitions as predicates
Where we are

We’ve covered algorithms for model checking Kripke structures
  ▶ Directly over transition graph (CTL)
  ▶ After converting to Buchi automata (LTL)
  ▶ Symbolically, representing states and transitions as predicates

If we want to check code, we need to convert to a Kripke structure
We’ve covered algorithms for model checking Kripke structures

- Directly over transition graph (CTL)
- After converting to Buchi automata (LTL)
- Symbolically, representing states and transitions as predicates

If we want to check code, we need to convert to a Kripke structure

How do we do this?

- Manually write down a model of the software
- Time-consuming, error-prone
- Hard to get right: no obvious 1-1 mapping from code to model
So, manual modeling is unattractive
So, manual modeling is unattractive

Perhaps we should have planned ahead
So, manual modeling is unattractive

Perhaps we should have planned ahead

▶ Start by formalizing the properties we want to achieve
So, manual modeling is unattractive

Perhaps we should have planned ahead

- Start by formalizing the properties we want to achieve
- Then write a model of the system we intend to implement
So, manual modeling is unattractive

Perhaps we should have planned ahead
  ▶ Start by formalizing the properties we want to achieve
  ▶ Then write a model of the system we intend to implement
  ▶ Verify the model
So, manual modeling is unattractive

Perhaps we should have planned ahead

▶ Start by formalizing the properties we want to achieve
▶ Then write a model of the system we intend to implement
▶ Verify the model
▶ Finally, write code to match (refine) the model
So, manual modeling is unattractive

Perhaps we should have planned ahead

- Start by formalizing the properties we want to achieve
- Then write a model of the system we intend to implement
- Verify the model
- Finally, write code to match (refine) the model

This is a great idea
So, manual modeling is unattractive

Perhaps we should have planned ahead

▶ Start by formalizing the properties we want to achieve
▶ Then write a model of the system we intend to implement
▶ Verify the model
▶ Finally, write code to match (refine) the model

This is a great idea

▶ Would be even better if anyone did this
So, manual modeling is unattractive

Perhaps we should have planned ahead
- Start by formalizing the properties we want to achieve
- Then write a model of the system we intend to implement
- Verify the model
- Finally, write code to match (refine) the model

This is a great idea
- Would be even better if anyone did this

Instead, we’ll look at automatic model extraction techniques
We want to model this as a TS

```c
// assume x is 0 or 1
// x init. nondet.
while (true) {
    if (x == 0) {
        x := 1;
        // do something
        x := 0;
    }
}
```
We want to model this as a TS

- Set of states, atm. propositions

```
// assume x is 0 or 1
// x init. nondet.
while(true) {
    if(x == 0) {
        x := 1;
        //do something
        x := 0;
    }
}
```
We want to model this as a TS

- Set of states, atm. propositions
- Initial states

```c
// assume x is 0 or 1
// x init. nondet.
while(true) {
    if(x == 0) {
        x := 1;
        // do something
        x := 0;
    }
}
```
Programs as Kripke Structures

// assume x is 0 or 1
// x init. nondet.
while(true) {
  if(x == 0) {
    x := 1;
    // do something
    x := 0;
  }
}

We want to model this as a TS
  ▶ Set of states, atm. propositions
  ▶ Initial states
  ▶ Transition relation
// assume x is 0 or 1
// x init. nondet.

while (true) {
    if (x = 0) {
        x := 1;
        // do something
        x := 0;
    }
}

We want to model this as a TS
- Set of states, atm. propositions
- Initial states
- Transition relation

Recall the semantics:

\[ \langle c_1, \sigma_1 \rangle \rightarrow \langle c_2, \sigma_2 \rangle \]
// assume x is 0 or 1
// x init. nondet.
while(true) {
    if(x == 0) {
        x := 1;
        // do something
        x := 0;
    }
}

We want to model this as a TS
- Set of states, atm. propositions
- Initial states
- Transition relation

Recall the semantics:
\[ \langle c_1, \sigma_1 \rangle \rightarrow \langle c_2, \sigma_2 \rangle \]

Need states for each configuration of \( \sigma \)
\[ S = \{(x = 0), (x = 1)\} \]
We want to model this as a TS

- Set of states, atm. propositions
- Initial states
- Transition relation

Recall the semantics:

\[ \langle c_1, \sigma_1 \rangle \rightarrow \langle c_2, \sigma_2 \rangle \]

Need states for each configuration of \( \sigma \)

\[ S = \{(x = 0), (x = 1)\} \]

Nondet. initialization gives us:

\[ I = \{(x = 0), (x = 1)\} \]
// assume $x$ is 0 or 1
// $x$ init. nondet.

```
while (true) {
    if ($x = 0$) {
        $x := 1$;
        //do something
        $x := 0$;
    }
}
```

Now for the transitions

Let $S = \{(x = 0), (x = 1)\}$
Let $I = \{(x = 0), (x = 1)\}$
// assume x is 0 or 1
// x init. nondet.
while(true) {
  if(x == 0) {
    x := 1;
  // do something
    x := 0;
  }
}

Now for the transitions

Now for the transitions

\[ S = \{(x = 0), (x = 1)\} \]
\[ I = \{(x = 0), (x = 1)\} \]
// assume x is 0 or 1
// x init. nondet.
while(true) {
    if(x = 0) {
        x := 1;
        //do something
        x := 0;
    }
}

Now for the transitions

\[
S = \{(x = 0), (x = 1)\}
I = \{(x = 0), (x = 1)\}
\]
// assume $x$ is 0 or 1
// $x$ init. nondet.

while (true) {
    if ($x = 0$) {
        $x := 1$;
        // do something
        $x := 0$;
    }
}

Now for the transitions

\[ S = \{(x = 0), (x = 1)\}\]
\[ I = \{(x = 0), (x = 1)\}\]
// assume $x$ is 0 or 1
// $x$ init. nondet.
while(true) {
    if($x = 0$) {
        $x := 1$;
        //do something
        $x := 0$;
    }
}

$S = \{(x = 0), (x = 1)\}$
$I = \{(x = 0), (x = 1)\}$

Now for the transitions

\[ x = 0 \rightarrow x = 1 \rightarrow x = 0 \]
// assume $x$ is 0 or 1
// $x$ init. nondet.
while(true) {
  if($x = 0$) {
    $x := 1$;
    //do something
    $x := 0$;
  }
}

Now for the transitions

Is this right?

$S = \{(x = 0), (x = 1)\}$
$I = \{(x = 0), (x = 1)\}$
```c
// assume x is 0 or 1
// x init. nondet.
while (true) {
    if (x == 0) {
        x := 1;
        // do something
        x := 0;
    }
}

S = {(x = 0), (x = 1)}
I = {(x = 0), (x = 1)}
```

Now for the transitions

Is this right?

Do both satisfy $x = 1 \rightarrow \text{AG} (x = 1)$?
Equating states with environments isn’t enough
Equating states with environments isn’t enough

We also need to consider control flow
Modeling Control Flow

Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
Modeling Control Flow

Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
- Add transitions only between states with related locations
Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
- Add transitions only between states with related locations

We formalize this with **program graphs** \(PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \Sigma_0)\)
Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
- Add transitions only between states with related locations

We formalize this with **program graphs** \(PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \Sigma_0)\)
- \(\text{Loc}\) is a set of program locations
Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
- Add transitions only between states with related locations

We formalize this with **program graphs** \(PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \Sigma_0)\)

- \text{Loc} is a set of program locations
- \text{Var} is a set of program variables
Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: $(\ell, \sigma)$
  - Add transitions only between states with related locations

We formalize this with **program graphs** $PG = (\text{Loc}, \text{Var}, C', T, \ell_0, \Sigma_0)$
  - $\text{Loc}$ is a set of program locations
  - $\text{Var}$ is a set of program variables
  - $C' : \text{Loc} \mapsto \text{Com}$ maps locations to commands
Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
- Add transitions only between states with related locations

We formalize this with **program graphs** \(PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \Sigma_0)\)
  - \(\text{Loc}\) is a set of program locations
  - \(\text{Var}\) is a set of program variables
  - \(C: \text{Loc} \mapsto \text{Com}\) maps locations to commands
  - \(T \subseteq \text{Loc} \times \text{BExp} \times \text{Loc}\) is the **conditional** transition relation
Equating states with environments isn’t enough

We also need to consider control flow

- **Locations** and environments: \((\ell, \sigma)\)
- Add transitions only between states with related locations

We formalize this with **program graphs** \(PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \Sigma_0)\)

- \(\text{Loc}\) is a set of program locations
- \(\text{Var}\) is a set of program variables
- \(C : \text{Loc} \mapsto \text{Com}\) maps locations to commands
- \(T \subseteq \text{Loc} \times \text{BExp} \times \text{Loc}\) is the **conditional** transition relation
- \(\ell_0\) the initial location, \(\Sigma_0\) the initial environments
Program Graph: Example

// assume x is 0 or 1
// x init. nondet.
\[ \ell_0 \textbf{while}(\text{true}) \{ \]
\[ \ell_1 \quad \textbf{if}(\text{x = 0}) \{ \]
\[ \ell_2 \quad x := 1; \]
\[ \quad \text{//do something} \]
\[ \ell_3 \quad x := 0; \]
\[ \} \]
\[ \} \]
/\ assume \ x \ is \ 0 \ or \ 1
/\ x \ init. \ nondet.

\ell_0 \textbf{while} (true) \{ 
\ell_1 \quad \textbf{if} (x = 0) \{ 
\ell_2 \quad x := 1;
// \textbf{do something}
\ell_3 \quad x := 0;
\}
\}

\ell_0 \quad \ell_1 \quad \ell_2 \quad \ell_3
// assume x is 0 or 1
// x init. nondet.

\[ \ell_0 \quad \textbf{while}(\text{true}) \quad \{ \]
\[ \ell_1 \quad \textbf{if}(x = 0) \quad \{ \]
\[ \ell_2 \quad x := 1; \]
\[ \text{ //do something} \]
\[ \ell_3 \quad x := 0; \]
\[ \} \]
\[ \} \]
Program Graph: Example

// assume x is 0 or 1
// x init. nondet.
\[ \begin{align*}
\ell_0 & \quad \text{while(true) } \\
\ell_1 & \quad \text{if}(x = 0) \{ \\
\ell_2 & \quad x := 1; \\
\ell_3 & \quad x := 0;
\}
\end{align*} \]
// assume $x$ is 0 or 1
// $x$ init. nondet.

while (true) {
  if ($x = 0$) {
    $x := 1$;
    // do something
  }
  $x := 0$;
}

\[ \ell_0 \quad \ell_1 \quad \ell_2 \quad \ell_3 \]

\[ \ell_e \quad \ell_0 \quad \ell_1 \quad \ell_2 \quad \ell_3 \]

\[ x = 0 \]
// assume x is 0 or 1
// x init. nondet.

\[ \begin{align*}
\ell_0 \quad & \textbf{while}(\text{true}) \{ \\
\ell_1 \quad & \textbf{if}(x = 0) \{ \\
\ell_2 \quad & x := 1; \\
\ell_3 \quad & \text{do something} \\
\} \\
\} 
\end{align*} \]
// assume \( x \) is 0 or 1
// \( x \) init. nondet.

\[
\begin{align*}
\ell_0 & \text{ while} (\text{true}) \{ \\
\ell_1 & \quad \text{if} (x = 0) \{ \\
\ell_2 & \quad x := 1; \quad \text{// do something} \\
\ell_3 & \quad x := 0; \\
\} \\
\} 
\end{align*}
\]
// assume $x$ is 0 or 1
// $x$ init. nondet.

$\ell_0$ while(true) {
  $\ell_1$ if($x = 0$) {
    $\ell_2$ $x := 1$;
    //do something
  }
  $\ell_3$ $x := 0$;
}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{program_graph.png}
\caption{Program Graph: Example}
\end{figure}
We can faithfully convert program graphs to Kripke structures

Given $PG = (\text{Loc, Var, } C, T, \ell_0, \sigma_0)$, we derive $M = (S, P, R, I, L)$:
We can faithfully convert program graphs to Kripke structures.

Given $PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \sigma_0)$, we derive $M = (S, P, R, I, L)$:

- $S = \text{Loc} \times \text{Env}$, $P = \text{Loc} \cup \text{Env}$
We can faithfully convert program graphs to Kripke structures

Given $PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \sigma_0)$, we derive $M = (S, P, R, I, L)$:

- $S = \text{Loc} \times \text{Env}$, $P = \text{Loc} \cup \text{Env}$
- $P = \text{Loc} \cup 2^\text{Env}$
Program Graphs as Kripke Structures

We can faithfully convert program graphs to Kripke structures

Given $PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \sigma_0)$, we derive $M = (S, P, R, I, L)$:

- $S = \text{Loc} \times \text{Env}$, $P = \text{Loc} \cup \text{Env}$
- $P = \text{Loc} \cup 2^{\text{Env}}$
- We derive $R \subseteq S \times S$ using operational semantics:

$$
\begin{align*}
(l_1, b, l_2) &\in T \quad \langle b, \sigma_1 \rangle \Downarrow_b \textbf{true} \quad \langle C(l_1), \sigma_1 \rangle \Downarrow \sigma_2 \\
([l_1, \sigma_1], [l_2, \sigma_2]) &\in R
\end{align*}
$$
Program Graphs as Kripke Structures

We can faithfully convert program graphs to Kripke structures

Given $PG = (\text{Loc}, \text{Var}, C, T, \ell_0, \sigma_0)$, we derive $M = (S, P, R, I, L)$:

- $S = \text{Loc} \times \text{Env}$, $P = \text{Loc} \cup \text{Env}$
- $P = \text{Loc} \cup 2^\text{Env}$
- We derive $R \subseteq S \times S$ using operational semantics:

$$
\begin{align*}
(\ell_1, b, \ell_2) &\in T \\
\langle b, \sigma_1 \rangle &\Downarrow_b \text{true} \\
\langle C(\ell_1), \sigma_1 \rangle &\Downarrow \sigma_2 \\
([\ell_1, \sigma_1], [\ell_2, \sigma_2]) &\in R
\end{align*}
$$

- $I = \{[\ell, \sigma] \mid \sigma \in \Sigma_0\}$
We can faithfully convert program graphs to Kripke structures.

Given $PG = (\text{Loc, Var, C, T, } \ell_0, \sigma_0)$, we derive $M = (S, P, R, I, L)$:

1. $S = \text{Loc} \times \text{Env}$, $P = \text{Loc} \cup \text{Env}$
2. $P = \text{Loc} \cup 2^{\text{Env}}$
3. We derive $R \subseteq S \times S$ using operational semantics:

   $$(\ell_1, b, \ell_2) \in T \quad \langle b, \sigma_1 \rangle \downarrow_b \text{true} \quad \langle C(\ell_1), \sigma_1 \rangle \downarrow \sigma_2$$

   $$([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R$$

- $I = \{[\ell, \sigma] \mid \sigma \in \Sigma_0\}$
- $L([\ell, \sigma]) = \{\ell\} \cup \{\sigma\}$
Example: Prog. Graph to Kripke Structure

\[ \ell_0 \xrightarrow{x = 1} \ell_1 \]

\[ \ell_1 \xrightarrow{x = 0} \ell_2 \]

\[ \ell_2 \xrightarrow{true} \ell_3 \]

\[ \ell_3 \xrightarrow{true} \ell_0 \]

\[ \ell_e \xrightarrow{false} \ell_0 \]
Example: Prog. Graph to Kripke Structure

\[
\begin{array}{c}
\ell_0 \\
\ell_1 \\
\ell_2 \\
\ell_3 \\
\end{array}
\quad x = 1
\]

\[
\begin{array}{c}
\ell_e \\
\ell_0 \\
\ell_1 \\
\ell_2 \\
\ell_3 \\
\end{array}
\quad x = 0
\]

\[
\begin{array}{c}
\text{false} \\
\text{true} \\
\text{true} \\
\text{true} \\
\end{array}
\]

\[
\begin{array}{c}
\ell_0 \\
x = 0 \\
\ell_1 \\
x = 1 \\
\ell_2 \\
x = 0 \\
\ell_3 \\
x = 1 \\
\end{array}
\]

\[
\begin{array}{c}
\ell_0 \\
x = 0 \\
\ell_1 \\
x = 1 \\
\ell_2 \\
x = 0 \\
\ell_3 \\
x = 1 \\
\end{array}
\]
Example: Prog. Graph to Kripke Structure

\[ \ell_0 \]
\[ \ell_1 \]
\[ \ell_2 \]
\[ \ell_3 \]

\[ x = 0 \]
\[ x = 1 \]

false
true

\[ \ell_e \]

true

\[ x = 0 \]

\[ x = 1 \]
Example: Prog. Graph to Kripke Structure

\[
\begin{align*}
\ell_0 &\xrightarrow{x = 0} \ell_1 \\
\ell_1 &\xrightarrow{x = 1} \ell_0 \\
\ell_2 &\xrightarrow{x = 0} \ell_3 \\
\ell_3 &\xrightarrow{x = 1} \ell_2
\end{align*}
\]
Example: Prog. Graph to Kripke Structure

\[
\begin{align*}
\ell_0 & \quad x = 1 \\
\ell_1 & \quad x = 0 \\
\ell_2 & \quad x = 0 \\
\ell_3 & \quad x = 0
\end{align*}
\]
Example: Prog. Graph to Kripke Structure

\[
\begin{align*}
\ell_0 & \quad \text{false} \\
\ell_1 & \quad \text{true} \\
\ell_2 & \quad x = 0 \\
\ell_3 & \quad \text{true}
\end{align*}
\]

\[
\begin{align*}
\ell_0 & \quad x = 1 \\
\ell_1 & \quad x = 0 \\
\ell_2 & \quad x = 0 \\
\ell_3 & \quad x = 0
\end{align*}
\]
Example: Prog. Graph to Kripke Structure

\[
\begin{align*}
\ell_0 & \xrightarrow{true} \ell_1 \xrightarrow{x = 0} \ell_2 \xrightarrow{true} \ell_3 \\
\ell_e & \xrightarrow{false} \ell_0 \xrightarrow{x = 1} \ell_1 \\
\end{align*}
\]
This is sufficient to model-check software.
This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!
Software Model Checking

This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!

Consider a ~ 1000 LoC program with a few dozen 32-bit int variables
This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!

Consider a \( \sim 1000 \) LoC program with a few dozen 32-bit int variables

\[
1000 \times 36 \times 2^{32} \approx 1.5 \times 10^{14}\] states
This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!

Consider a ~ 1000 LoC program with a few dozen 32-bit int variables

▶ $1000 \times 36 \times 2^{32} \approx 1.5 \times 10^{14}$ states

▶ Can’t be *that* optimistic about optimized search
This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!

Consider a ~ 1000 LoC program with a few dozen 32-bit int variables

- \(1000 \times 36 \times 2^{32} \approx 1.5 \times 10^{14}\) states
- Can’t be *that* optimistic about optimized search
- We’ll need to be more clever

We’ll go into two prominent techniques to mitigate this complexity
This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!

Consider a ~ 1000 LoC program with a few dozen 32-bit int variables

1. $1000 \times 36 \times 2^{32} \approx 1.5 \times 10^{14}$ states
2. Can’t be *that* optimistic about optimized search
3. We’ll need to be more clever

We’ll go into two prominent techniques to mitigate this complexity

1. **Abstraction**: Build a concise approximation of the Kripke structure, ensure that it includes all the error traces
This is sufficient to model-check software

But it isn’t practical: too many states for even moderate programs!

Consider a \(\sim 1000\) LoC program with a few dozen 32-bit int variables

- \(1000 \times 36 \times 2^{32} \approx 1.5 \times 10^{14}\) states
- Can’t be *that* optimistic about optimized search
- We’ll need to be more clever

We’ll go into two prominent techniques to mitigate this complexity

1. **Abstraction**: Build a concise approximation of the Kripke structure, ensure that it includes all the error traces
2. **Bounded symbolic checking**: Compact first-order logical representation up to a fixed execution depth
Idea: Approximate the KS so that the property is preserved
Idea: Approximate the KS so that the property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$
**Idea:** Approximate the KS so that the property is preserved.

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$. 
Abstraction

**Idea:** Approximate the KS so that the property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
Idea: Approximate the KS so that the property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in $M$
**Idea:** Approximate the KS so that the property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in $M$

This preserves safety properties: if $\hat{M}$ verifies, so will $M$
**Idea:** Approximate the KS so that the property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in $M$

This preserves safety properties: if $\hat{M}$ verifies, so will $M$

But it might introduce **spurious counterexamples**
Abstraction: Example

\[ \ell_0 x = 0 \]
\[ \ell_1 x = 0 \]
\[ \ell_2 x = 0 \]
\[ \ell_3 x = 1 \]

\[ \ell_0 x = 1 \]
\[ \ell_1 x = 0 \]
\[ \ell_2 x = 0 \]

Does this preserve the formula?

AG x = 1
Abstraction: Example

Does this preserve the formula?

\[ x = 1 \]

\[ AG x = 1 \]
Abstraction: Example

Does this preserve the formula?

\[x = 1\]
Abstraction: Example

\[ \ell_0 x = 0 \]
\[ \ell_1 x = 0 \]
\[ \ell_2 x = 0 \]
\[ \ell_3 x = 1 \]
\[ x = 0 \]
\[ x = 1 \]

Does this preserve the formula?

\[ x = 1 \]
Abstraction: Example

\[ \ell_0 \quad x = 0 \]
\[ \ell_1 \quad x = 0 \]
\[ \ell_2 \quad x = 0 \]
\[ \ell_3 \quad x = 1 \]

Does this preserve the formula?

\[ x = 1 \rightarrow \text{AG} \quad x = 1 \]
Abstraction: Another Example

\[ \ell_0 \quad x = 0 \]
\[ \ell_1 \quad x = 0 \]
\[ \ell_2 \quad x = 0 \]
\[ \ell_3 \quad x = 1 \]

\[ \ell_0 \quad x = 1 \]
\[ \ell_1 \quad x = 1 \]

Does this preserve the formula? No, we saw this before.

\[ x = 1; (x = 0; x = 1) \]

is spurious.
Abstraction: Another Example

Does this preserve the formula? No, we saw this before.
Abstraction: Another Example

\[ \ell_0 x = 0 \]
\[ \ell_1 x = 0 \]
\[ \ell_2 x = 0 \]
\[ \ell_3 x = 1 \]

\[ \ell_0 x = 1 \]
\[ \ell_1 x = 1 \]

Does this preserve the formula?

\[ x = 1 \rightarrow \text{AG} \ x = 1 \]
Abstraction: Another Example

Does this preserve the formula?

\[ x = 1 \rightarrow \text{AG } x = 1 \]

No, we saw this before.
Abstraction: Another Example

Does this preserve the formula? 

\[ x = 1 \rightarrow \textbf{AG} \; x = 1 \]

No, we saw this before.

\[ x = 1, (x = 0, x = 1)^\omega \text{ is spurious} \]
How do we know which abstraction to use?
How do we know which abstraction to use?

**Idea:** Only track *predicates* on program state
How do we know which abstraction to use?

Idea: Only track **predicates** on program state

- Predicates relevant to the property, control flow
Predicate Abstraction

How do we know which abstraction to use?

**Idea:** Only track *predicates* on program state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values
How do we know which abstraction to use?

**Idea:** Only track *predicates* on program state
- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We’re given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define *abstraction function* $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$$\alpha(\sigma) = (\phi_1(\sigma), \ldots, \phi_n(\sigma))$$
How do we know which abstraction to use?

Idea: Only track **predicates** on program state
- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We’re given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$$\alpha(\sigma) = (\phi_1(\sigma), \ldots, \phi_n(\sigma))$$

Think: $\alpha$ ranges over conjunctions of $\phi_i, \neg\phi_i$
How do we know which abstraction to use?

**Idea:** Only track **predicates** on program state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We’re given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$$\alpha(\sigma) = (\phi_1(\sigma), \ldots, \phi_n(\sigma))$$

Think: $\alpha$ ranges over conjunctions of $\phi_i, \neg\phi_i$

The states in our abstraction will be: $S = \text{Loc} \times \{0, 1\}^m$
Existential Abstraction

How do we abstract transitions?
Existential Abstraction

How do we abstract transitions?

Remember, we want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]

A transition is in the abstraction \( \hat{M} \) if and only if:

1. There exist corresponding states \((s_1, s_2)\) in \( M \), where \( s_1, s_2 \) are the endpoints of a transition in \( M \).

Why is this conservative?
Existential Abstraction

How do we abstract transitions?

Remember, we want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]

We’ll define an **existential abstraction**: 

\[
(\hat{s}_1, \hat{s}_2) \in \hat{R} \iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2
\]

\[
\hat{s} \in \hat{I} \iff \exists s. s \in I \land h(s) = \hat{s}
\]
Existential Abstraction

How do we abstract transitions?

Remember, we want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]

We’ll define an existential abstraction:

\[
(\hat{s}_1, \hat{s}_2) \in \hat{R} \iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2
\]

\[
\hat{s} \in \hat{I} \iff \exists s. s \in I \land h(s) = \hat{s}
\]

A transition is in the abstraction \( \hat{M} \) if and only if:
Existential Abstraction

How do we abstract transitions?

Remember, we want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]

We’ll define an **existential abstraction**:

\[
\begin{align*}
(\hat{s}_1, \hat{s}_2) \in \hat{R} &\iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2 \\
\hat{s} \in \hat{I} &\iff \exists s. s \in I \land h(s) = \hat{s}
\end{align*}
\]

A transition is in the abstraction \( \hat{M} \) if and only if:

1. There **exist** corresponding states \((s_1, s_2)\) in \( M \),
Existential Abstraction

How do we abstract transitions?

Remember, we want an over-approximation that gives us:

\[
\widehat{M} \models \phi \Rightarrow M \models \phi
\]

We’ll define an existential abstraction:

\[
(\hat{s}_1, \hat{s}_2) \in \hat{R} \iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2
\]

\[
\hat{s} \in \hat{I} \iff \exists s. s \in I \land h(s) = \hat{s}
\]

A transition is in the abstraction \( \hat{M} \) if and only if:

1. There exist corresponding states \((s_1, s_2)\) in \( M \),
2. where \( s_1, s_2 \) are the endpoints of a transition in \( M \)
Existential Abstraction

How do we abstract transitions?

Remember, we want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]

We’ll define an existential abstraction:

\[
(\hat{s}_1, \hat{s}_2) \in \hat{R} \iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2 \\
\hat{s} \in \hat{I} \iff \exists s. s \in I \land h(s) = \hat{s}
\]

A transition is in the abstraction \( \hat{M} \) if and only if:

1. There exist corresponding states \((s_1, s_2)\) in \( M \),
2. where \( s_1, s_2 \) are the endpoints of a transition in \( M \)

Why is this conservative?
Example: Existential Abstraction

Suppose we use:

\[ p_0, (c_1 \leq c_2) \quad p_1, (y = 1) \]
Example: Existential Abstraction

Suppose we use:

\[ p_0 \iff (c_1 \lor c_2) \]
\[ p_1 \iff (y = 1) \]
Example: Existential Abstraction

Suppose we use:

\[ p_0 \Leftrightarrow (c_1 \lor c_2) \]
\[ p_1 \Leftrightarrow (y = 1) \]
Example: Existential Abstraction

Suppose we use:

\[ p_0 \iff (c_1 \lor c_2) \]
\[ p_1 \iff (y = 1) \]
Example: Existential Abstraction

Suppose we use:

\[ p_0 \iff (c_1 \lor c_2) \]
\[ p_1 \iff (y = 1) \]
Suppose we use:

\[ p_0 \Leftrightarrow (c_1 \lor c_2) \]
\[ p_1 \Leftrightarrow (y = 1) \]
Example: Existential Abstraction

Suppose we use:

\[ p_0 \Leftrightarrow (c_1 \lor c_2) \]
\[ p_1 \Leftrightarrow (y = 1) \]
Example: Existential Abstraction

Suppose we use:

\[ p_0 \Leftrightarrow (c_1 \lor c_2) \]
\[ p_1 \Leftrightarrow (y = 1) \]
How do we compute program approximations?
How do we compute program approximations?

The key issue: how do we compute transitions
How do we compute program approximations?

The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

$$\left(\ell_1, b, \ell_2\right) \in T \quad \left\langle b, \sigma_1 \right\rangle \Downarrow_b \text{true} \quad \left\langle C(\ell_1), \sigma_1 \right\rangle \Downarrow \sigma_2$$

$$\left(\text{[}\ell_1, \sigma_1\text{]}, \text{[}\ell_2, \sigma_2\text{]}\right) \in R$$
How do we compute program approximations?

The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

\[
(\ell_1, b, \ell_2) \in T \quad \langle b, \sigma_1 \rangle \Downarrow_b \text{true} \quad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2
\]

\[
([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R
\]

We don’t have concrete states \( \sigma \) to work with anymore
How do we compute program approximations?

The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

\[
(\ell_1, b, \ell_2) \in T \quad \langle b, \sigma_1 \rangle \downarrow_b \text{true} \quad \langle C(\ell_1), \sigma_1 \rangle \downarrow \sigma_2
\]

\[
([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R
\]

We don’t have concrete states \(\sigma\) to work with anymore

Just predicates.
How do we compute program approximations?

The key issue: how do we compute **transitions**

Recall our construction of KS from program graphs:

\[
\begin{align*}
(l_1, b, l_2) &\in T \quad \langle b, \sigma_1 \rangle \downarrow_b \text{true} \quad \langle C(l_1), \sigma_1 \rangle \downarrow \sigma_2 \\
([l_1, \sigma_1], [l_2, \sigma_2]) &\in R
\end{align*}
\]

We don’t have concrete states $\sigma$ to work with anymore

Just predicates. **Idea:** Use predicate transformers
Predicate Transformers (Refresher)

Given an assertion $Q$ and program $c$, we described describe a function:

- That is a **predicate transformer**: produces another assertion
- Assertion for the corresponding precondition $P$ for $c$
- Guaranteed to be the **weakest** such assertion
Given an assertion $Q$ and program $c$, we described a function:

- That is a *predicate transformer*: produces another assertion
- Assertion for the corresponding precondition $P$ for $c$
- Guaranteed to be the *weakest* such assertion

This is the *weakest precondition* predicate transformer $wp(c, Q)$
Predicate Transformers (Refresher)

Given an assertion $Q$ and program $c$, we described a function:

- That is a **predicate transformer**: produces another assertion
- Assertion for the corresponding precondition $P$ for $c$
- Guaranteed to be the **weakest** such assertion

This is the **weakest precondition** predicate transformer $wp(c, Q)$

The weakest precondition satisfies the following conditions:
1. The triple $[wp(c, Q)] c [Q]$ is valid
2. For any $P$ where $[P] c [Q]$ is valid, $P \Rightarrow wp(c, Q)$
We used Hoare triples to define \( \text{wlp} \)
We used Hoare triples to define \( \text{wp} \)

Recall the rule for assignment:

\[
\text{Asgn} \quad \frac{\{Q[a/x]\} \quad x := a\{Q\}}{} \]

If \( P \not\models Q[a/x] \), then \( f \models g \) won't hold
We used Hoare triples to define \( \text{wlp} \).

Recall the rule for assignment:

\[
\text{Asgn} \quad \begin{array}{c}
\{Q[a/x]\} \quad x := a \{Q\}
\end{array}
\]

The corresponding transformer is:

\[
\text{wp}(x := a, Q) = Q[a/x]
\]
We used Hoare triples to define \( wlp \)

Recall the rule for assignment:

\[
\text{Asgn} \quad \begin{array}{ll}
\{ Q[a/x] \} & x := a \{ Q \}
\end{array}
\]

The corresponding transformer is:

\[
wp(x := a, Q) = Q[a/x]
\]

If \( P \not\models Q[a/x] \), then \( \{ P \} \ c \ \{ Q \} \) won’t hold
Abstracting Program Transitions

Suppose we have a very simple program:

\[ \ell_0 : \ x := x + 1 \]
\[ \ell_1 : \ \text{skip} \]
Abstracting Program Transitions

Suppose we have a very simple program:

\[\ell_0 : \ x := x + 1\]
\[\ell_1 : \ \text{skip}\]

Predicate \(\phi\) is true after \(\ell_0\) iff \(wp(\ell_0, \phi)\) is true at \(\ell_0\)
Suppose we have a very simple program:

\[
\ell_0 : \quad x := x + 1 \\
\ell_1 : \quad \text{skip}
\]

Predicate \( \phi \) is true after \( \ell_0 \) iff \( \wp(\ell_0, \phi) \) is true at \( \ell_0 \)

- We can add a transition from \((\ell_0, \wp(\ell_0, \phi))\) to \((\ell_1, \phi)\)
Suppose we have a very simple program:

\[ \begin{align*}
\ell_0 : & \quad x := x + 1 \\
\ell_1 : & \quad \text{skip}
\end{align*} \]

Predicate \( \phi \) is true after \( \ell_0 \) iff \( \text{wp}(\ell_0, \phi) \) is true at \( \ell_0 \)

- We can add a transition from \((\ell_0, \text{wp}(\ell_0, \phi))\) to \((\ell_1, \phi)\)
- Any problems with this?
Suppose we have a very simple program:

\[
\ell_0 : \ x := x + 1 \\
\ell_1 : \ \text{skip}
\]

Predicate \( \phi \) is true after \( \ell_0 \) iff \( \text{wp}(\ell_0, \phi) \) is true at \( \ell_0 \)

- We can add a transition from \((\ell_0, \text{wp}(\ell_0, \phi))\) to \((\ell_1, \phi)\)
- Any problems with this?

Our predicate set probably doesn’t consist of this one predicate
Suppose we have a very simple program:

\[
\ell_0 : \quad x := x + 1 \\
\ell_1 : \quad \text{skip}
\]

Predicate \( \phi \) is true after \( \ell_0 \) iff \( \text{wp}(\ell_0, \phi) \) is true at \( \ell_0 \)

- We can add a transition from \( (\ell_0, \text{wp}(\ell_0, \phi)) \) to \( (\ell_1, \phi) \)
- Any problems with this?

Our predicate set probably doesn’t consist of this one predicate

- Worse yet: \( \text{wp}(\ell_0, \phi) \) may not be expressible in terms of it
Suppose we have a very simple program:

\[
\ell_0 : \quad x := x + 1 \\
\ell_1 : \quad \text{skip}
\]

Predicate \( \phi \) is true after \( \ell_0 \) iff \( \text{wp}(\ell_0, \phi) \) is true at \( \ell_0 \)

- We can add a transition from \( (\ell_0, \text{wp}(\ell_0, \phi)) \) to \( (\ell_1, \phi) \)
- Any problems with this?

Our predicate set probably doesn’t consist of this one predicate

- Worse yet: \( \text{wp}(\ell_0, \phi) \) may not be expressible in terms of it

We need to introduce another approximation: \( \text{wp} \) in terms of \( E \)
Strengthening Predicates

Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:
Strengthening Predicates

Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$, 

$\text{Env} \phi \text{ Pred}(\phi, E)$
Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$,
- that is at least as strong as $\phi$, 

> $\phi$ 

> $\text{Pred}(\phi, E)$ 

> $\text{Env}$
Strengthening Predicates

Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$,
- that is at least as strong as $\phi$,
- where each clause has $n$ literals

Notice: $\text{Pred}(\phi, E) \Rightarrow \phi$
Strengthening Predicates

Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$,
- that is at least as strong as $\phi$,
- where each clause has $n$ literals

Notice: $\text{Pred}(\phi, E) \Rightarrow \phi$

Compute this by querying SMT solver
Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$,
- that is at least as strong as $\phi$,
- where each clause has $n$ literals

Notice: $\text{Pred}(\phi, E) \Rightarrow \phi$

Compute this by querying SMT solver

- What’s the complexity of this?
Strengthening Predicates

Given \( E = \{\phi_1, \ldots, \phi_n\} \), let \( \text{Pred}(\phi, E) \):

- The \textbf{weakest} DNF over \( E \),
- that is at least as strong as \( \phi \),
- where each clause has \( n \) literals

Notice: \( \text{Pred}(\phi, E) \Rightarrow \phi \)

Compute this by querying SMT solver

- What’s the complexity of this?
- \( O(2^n) \)
Strengthening Predicates

Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$,
- that is at least as strong as $\phi$,
- where each clause has $n$ literals

Notice: $\text{Pred}(\phi, E) \Rightarrow \phi$

Compute this by querying SMT solver

- What’s the complexity of this?
- $O(2^n)$
- Need to query each:
  \[ p_1 \land \cdots \land p_n \Rightarrow \phi \]
  where $p_i$ is $\phi_i$ or $\neg \phi_i$
Example: Strengthening Predicates

Let \( E = \{ x = 1, x = 2, x < 3, x < 4, x > 4, false \} \)
Let $E = \{ x = 1, x = 2, x < 3, x < 4, x > 4, false \}$

How do we strengthen the following:
Example: Strengthening Predicates

Let $E = \{x = 1, x = 2, x < 3, x < 4, x > 4, \text{false}\}$

How do we strengthen the following:

- $x \leq 2$
Let $E = \{x = 1, x = 2, x < 3, x < 4, x > 4, false\}$

How do we strengthen the following:

$\rightarrow x \leq 2 \quad x = 1 \lor x = 2$
Example: Strengthening Predicates

Let \( E = \{ x = 1, x = 2, x < 3, x < 4, x > 4, \text{false} \} \)

How do we strengthen the following:

- \( x \leq 2 \quad x = 1 \lor x = 2 \)
- \( x \neq 0 \)
Example: Strengthening Predicates

Let $E = \{x = 1, x = 2, x < 3, x < 4, x > 4, \text{false}\}$

How do we strengthen the following:

- $x \leq 2 \quad x = 1 \lor x = 2$
- $x \neq 0 \quad x = 1 \lor x = 2 \lor \neg(x < 3)$
Let $E = \{x = 1, x = 2, x < 3, x < 4, x > 4, \text{false}\}$

How do we strengthen the following:

- $x \leq 2 \quad x = 1 \lor x = 2$
- $x \neq 0 \quad x = 1 \lor x = 2 \lor \neg(x < 3)$
- $x = 0$
Example: Strengthening Predicates

Let $E = \{ x = 1, x = 2, x < 3, x < 4, x > 4, \text{false} \}$

How do we strengthen the following:

- $x \leq 2 \quad x = 1 \lor x = 2$
- $x \neq 0 \quad x = 1 \lor x = 2 \lor \neg (x < 3)$
- $x = 0 \quad \text{false}$
Let $E = \{x = 1, x = 2, x < 3, x < 4, x > 4, \text{false}\}$

How do we strengthen the following:

- $x \leq 2 \Rightarrow x = 1 \lor x = 2$
- $x \neq 0 \Rightarrow x = 1 \lor x = 2 \lor \neg(x < 3)$
- $x = 0 \Rightarrow \text{false}$
- $x = 3 \lor x = 4$
Let $E = \{x = 1, x = 2, x < 3, x < 4, x > 4, \text{false}\}$

How do we strengthen the following:

- $x \leq 2 \quad x = 1 \lor x = 2$
- $x \neq 0 \quad x = 1 \lor x = 2 \lor \neg (x < 3)$
- $x = 0 \quad \text{false}$
- $x = 3 \lor x = 4 \quad \neg (x < 3) \land x < 4 \lor \neg (x < 4) \land \neg (x > 4)$
Recall the strengthening rule:

\[
\text{Pre} \quad \frac{\vdash \{P'\} \ c \ \{Q\} \quad P \Rightarrow P'}{\{P\} \ c \ \{Q\}}
\]
Recall the strengthening rule:

\[
\text{Pre} \quad \vdash \{P'\} c \{Q\} \quad P \Rightarrow P' \\
\{P\} c \{Q\}
\]

1. We know that \(\{\text{wp}(c, \phi)\} c \{\phi\}\)
Recall the strengthening rule:

\[
\text{Pre} \quad \frac{\vdash \{ P' \} \ c \ \{ Q \}}{} \quad P \Rightarrow P' \\
\{ P \} \ c \ \{ Q \}
\]

1. We know that \( \{ \text{wp}(c, \phi) \} \ c \ \{ \phi \} \)

2. And we know that \( \text{Pred}(\text{wp}(c, \phi), E) \Rightarrow \text{wp}(c, \phi) \)
Recall the strengthening rule:

\[
\begin{align*}
\text{Pre} & \quad \vdash \{P'\} \ c \ \{Q\} \\
& \quad P \Rightarrow P' \\
\{P\} & \quad c \ \{Q\}
\end{align*}
\]

1. We know that \{wp(c, \phi)\} \ c \ \{\phi\}

2. And we know that \text{Pred}(wp(c, \phi), E) \Rightarrow wp(c, \phi)

So, we have that \{\text{Pred}(wp(c, \phi), E)\} \ c \ \{\phi\}
Recall the strengthening rule:

\[
\text{Pre} \quad \vdash \{P'\} \ c \ \{Q\} \quad P \Rightarrow P'
\]

\[
\{P\} \ c \ \{Q\}
\]

1. We know that \(\{\wp(c, \phi) \ c \ \{\phi\}\}\)
2. And we know that \(\text{Pred}(\wp(c, \phi), E) \Rightarrow \wp(c, \phi)\)

So, we have that \(\{\text{Pred}(\wp(c, \phi), E)\} \ c \ \{\phi\}\)

- If \(\text{Pred}(\wp(c, \phi_i), E)\) is true before \(c\), then \(\phi_i\) is true after
Recall the strengthening rule:

\[ \text{Pre} \vdash \{P'\} c \{Q\} \quad P \Rightarrow P' \]

\[ \{P\} c \{Q\} \]

1. We know that \( \{\text{wp}(c, \phi)\} \ c \ \{\phi\} \)

2. And we know that \( \text{Pred}(\text{wp}(c, \phi), E) \Rightarrow \text{wp}(c, \phi) \)

So, we have that \( \{\text{Pred}(\text{wp}(c, \phi), E)\} \ c \ \{\phi\} \)

- If \( \text{Pred}(\text{wp}(c, \phi_i), E) \) is true before \( c \), then \( \phi_i \) is true after
- If \( \text{Pred}(\neg\text{wp}(c, \phi_i), E) \) is true before \( c \), then \( \neg\phi_i \) is true after
Recall the strengthening rule:

\[
\text{Pre} \quad \frac{\vdash \{P'\} \ c \ \{Q\}}{\{P\} \ c \ \{Q\}} \quad P \Rightarrow P'
\]

1. We know that \(\{\text{wp}(c, \phi)\} \ c \ \{\phi\}\)

2. And we know that \(\text{Pred}(\text{wp}(c, \phi), E) \Rightarrow \text{wp}(c, \phi)\)

So, we have that \(\{\text{Pred}(\text{wp}(c, \phi), E)\} \ c \ \{\phi\}\)

- If \(\text{Pred}(\text{wp}(c, \phi_i), E)\) is true before \(c\), then \(\phi_i\) is true after
- If \(\text{Pred}(\neg\text{wp}(c, \phi_i), E)\) is true before \(c\), then \(\neg\phi_i\) is true after

Now we know how to compute transitions
For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [\ldots]))$: 

1. If $\wedge_{1 \leq i \leq n} b_i$ 
2. If $\wedge_{1 \leq i \leq n} \neg b_i$ 
3. Otherwise 
4. Draw transition $((\ell, [b_1, \ldots, b_n]), (\ell', [b_1', \ldots, b_n']))$
Abstracting Transitions

For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [\ldots]))$:

1. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(wp(C(\ell), \phi_i), E)$, set $b_i' := \text{true}$
Abstracting Transitions

For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [...]))$:

1. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\text{wp}(C(\ell), \phi_i), E)$, set $b_i' := true$
2. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\neg \text{wp}(C(\ell), \phi_i), E)$, set $b_i' := false$
Abstracting Transitions

For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [\ldots]))$:

1. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\text{wp}(C(\ell), \phi_i), E)$, set $b'_i := \text{true}$
2. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\neg\text{wp}(C(\ell), \phi_i), E)$, set $b'_i := \text{false}$
3. Otherwise, set $b'_i := \ast$ (could be either)
Abstracting Transitions

For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [\ldots]))$:

1. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(wp(C(\ell), \phi_i), E)$, set $b_i' := \text{true}$
2. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\neg wp(C(\ell), \phi_i), E)$, set $b_i' := \text{false}$
3. Otherwise, set $b_i' := *$ (could be either)
4. Draw transition $((\ell, [b_1, \ldots, b_n]), (\ell', [b_1', \ldots, b_n']))$
Abstracting Transitions

For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [\ldots]))$:

1. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\text{wp}(C(\ell), \phi_i), E)$, set $b'_i := \text{true}$
2. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\neg\text{wp}(C(\ell), \phi_i), E)$, set $b'_i := \text{false}$
3. Otherwise, set $b'_i := \ast$ (could be either)
4. Draw transition $((\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]))$

\[
\ell_0 : \quad x := x + 1
\]
\[
\ell_1 : \quad \text{skip}
\]

\[
E = \left\{ x = y \right\}_{p_0}
\]
Abstracting Transitions

For each $\phi_i$ in $E$, each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [\ldots]))$: 

1. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\text{wp}(C(\ell), \phi_i), E)$, set $b_i' := \text{true}$
2. If $\bigwedge_{1 \leq i \leq n} b_i \Rightarrow \text{Pred}(\neg\text{wp}(C(\ell), \phi_i), E)$, set $b_i' := \text{false}$
3. Otherwise, set $b_i' := \ast$ (could be either)
4. Draw transition $((\ell, [b_1, \ldots, b_n]), (\ell', [b_1', \ldots, b_n']))$

$\ell_0 : \ x := x + 1$
$\ell_1 : \ \text{skip}$

$E = \{x = y\}$
We have that $\phi$ is overapproximated by $\neg\text{Pred}(\neg\phi, E)$.
We have that $\phi$ is overapproximated by $\neg\text{Pred}(\neg\phi, E)$

Observe: everything outside the rightmost circle is $\neg\text{Pred}(\neg\phi)$
Abstracting Conditionals

What about conditional statements?
Abstracting Conditionals

What about conditional statements?

We treat them like **assume**
What about conditional statements?

We treat them like **assume**

We have that $\phi$ is overapproximated by $\neg \text{Pred}(\neg \phi, E)$

For $\text{if}(b)\{\ldots\}$, for each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]))$: 
Abstracting Conditionals

What about conditional statements?

We treat them like assume

We have that $\phi$ is overapproximated by $\neg\text{Pred}(\neg\phi, E)$

For if$(b)\{\ldots\}$, for each trans. $((\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]))$:
  - If $\bigwedge_{1 \leq i \leq n} b'_i \Rightarrow \neg\text{Pred}(\neg b, E)$, add $((\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]))$ (i.e., then case)
What about conditional statements?

We treat them like **assume**

We have that \( \phi \) is overapproximated by \( \neg \text{Pred}(\neg \phi, E) \)

For \( \text{if}(b)\{\ldots\} \), for each trans. \(( (\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]) )\):

- If \( \bigwedge_{1 \leq i \leq n} b'_i \Rightarrow \neg \text{Pred}(\neg b, E) \), add \(( (\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]) )\) (i.e., **then** case)
- If \( \bigwedge_{1 \leq i \leq n} b'_i \Rightarrow \neg \text{Pred}(b, E) \), add \(( (\ell, [b_1, \ldots, b_n]), (\ell', [b'_1, \ldots, b'_n]) )\) (i.e., **else** case)
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G \left( \neg \ell_0 \rightarrow 0 \leq i \right) \]

Using:
\[ E = \{0 \leq i\} \]
\[ p_0 \]
Example: Predicate Abstraction

ℓ₀ : i := 1;
ℓ₁ : while (0 ≤ x < 1) {
  ℓ₂ : i := i - 1;
  ℓ₃ : x := x + 1;
}

Suppose we check:

\[ G (¬ℓ₀ → 0 ≤ i) \]

Using:

\[ E = \{0 ≤ i\} \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ while(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{ 0 \leq i \} \]
\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0: \quad i := 1; \]
\[ \ell_1: \quad \textbf{while}(0 \leq x < 1) \quad \{ \]
\[ \ell_2: \quad i := i - 1; \]
\[ \ell_3: \quad x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{ 0 \leq i \} \]
\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:

\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:

\[ E = \{0 \leq i\} \]
Example: Predicate Abstraction

\[ \ell_0 : \; i := 1; \]
\[ \ell_1 : \; \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \; i := i - 1; \]
\[ \ell_3 : \; x := x + 1; \]
\[ \} \]

Suppose we check:
\[ \mathbf{G} \left( \neg \ell_0 \rightarrow 0 \leq i \right) \]

Using:
\[ E = \{0 \leq i\} \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:

\[ \textbf{G} \ (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:

\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while} (0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G \ ((\neg \ell_0 \rightarrow 0 \leq i)) \]

Using:
\[ E = \{0 \leq i\} \]
\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:
\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[
\ell_0 : \quad i := 1;
\]
\[
\ell_1 : \quad \textbf{while}(0 \leq x < 1) \quad \{
\]
\[
\ell_2 : \quad i := i - 1;
\]
\[
\ell_3 : \quad x := x + 1;
\]
\[
\}
\]

Suppose we check:

\[ G \left( \neg \ell_0 \rightarrow 0 \leq i \right) \]

Using:

\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]
Example: Predicate Abstraction

\[\ell_0 : \quad i := 1;\]
\[\ell_1 : \quad \textbf{while}(0 \leq x < 1) \{\]
\[\ell_2 : \quad i := i - 1;\]
\[\ell_3 : \quad x := x + 1;\]
\[\}\]

Does the property hold?
\[\textbf{G} \ (\neg \ell_0 \rightarrow 0 \leq i)\]
Does the property hold?

\[ \mathcal{G} \left( \neg \ell_0 \rightarrow 0 \leq i \right) \]

No.
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Does the property hold?

\[ \textbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

\textbf{No.} Should it?
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \ \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Does the property hold?

\[ \mathbf{G} \ (\neg \ell_0 \rightarrow 0 \leq i) \]

\textbf{No.} Should it? \textbf{Yes.}
Example: Predicate Abstraction

\[
\ell_0 : \quad i := 1;
\ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 : \quad i := i - 1;
\ell_3 : \quad x := x + 1;
\}
\]

Does the property hold?

\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

\textbf{No.} Should it? \textbf{Yes.}

All counterexamples are \textit{spurious}. 
Example: Predicate Abstraction

\[\ell_0 : \ i := 1;\]
\[\ell_1 : \ \textbf{while}(0 \leq x < 1) \{\]
\[\ell_2 : \ i := i - 1;\]
\[\ell_3 : \ x := x + 1; \}

Does the property hold?

\[G(\neg \ell_0 \rightarrow 0 \leq i)\]

\textbf{No.} Should it? \textbf{Yes.}

All counterexamples are \textit{spurious}.

Our abstraction is too coarse.
We need to \textit{refine} it.
Abstraction refinement

- Leverage counterexamples to find new predicates
- Automatic construction of good-enough abstractions
- Lazy, on-demand refinement

Next Lecture
Abstraction refinement
  ▶ Leverage counterexamples to find new predicates
  ▶ Automatic construction of good-enough abstractions
  ▶ Lazy, on-demand refinement

Bounded model checking using SAT/SMT
Abstraction refinement
- Leverage counterexamples to find new predicates
- Automatic construction of good-enough abstractions
- Lazy, on-demand refinement

Bounded model checking using SAT/SMT

Start the last assignment today (if you haven’t already)!