Predicate Abstraction: A Tutorial
Predicate Abstraction

Daniel Kroening

May 28 2012
Outline

Introduction

Existential Abstraction

Predicate Abstraction for Software

Counterexample-Guided Abstraction Refinement

Computing Existential Abstractions of Programs

Checking the Abstract Model

Simulating the Counterexample

Refining the Abstraction
Model Checking with Predicate Abstraction

- A heavy-weight formal analysis technique

- Recent successes in software verification, e.g., SLAM at Microsoft

- The abstraction reduces the size of the model by removing irrelevant detail

- The abstract model is then small enough for an analysis with a BDD-based Model Checker

- Idea: only track predicates on data, and remove data variables from model

- Mostly works with control-flow dominated properties
Reminder Abstract Interpretation

Abstract Domain

Approximate representation of sets of concrete values

\[ S \xrightarrow{\alpha} \hat{S} \xleftarrow{\gamma}\hat{S} \]
We are given a set of predicates over $S$, denoted by $\Pi_1, \ldots, \Pi_n$.

An abstract state is a valuation of the predicates:

$$\hat{S} = \mathbb{B}^n$$

The abstraction function:

$$\alpha(s) = \langle \Pi_1(s), \ldots, \Pi_n(s) \rangle$$
Predicate Abstraction: the Basic Idea

Concrete states over variables $x, y$:

- $x = 2$
  - $y = 0$  $ightarrow$  $x = 2$
  - $y = 1$  $ightarrow$  $x = 0$
  - $y = 0$

- $x = 1$
  - $y = 0$  $ightarrow$  $x = 1$
  - $y = 1$  $ightarrow$  $x = 1$
  - $y = 2$
Predicate Abstraction: the Basic Idea

Concrete states over variables $x$, $y$:

Predicates:

$p_1 \iff x > y$

$p_2 \iff y = 0$
Predicate Abstraction: the Basic Idea

Concrete states over variables $x, y$:

Predicates:

$p_1 \iff x > y$

$p_2 \iff y = 0$
Predicate Abstraction: the Basic Idea

Concrete states over variables $x, y$:

```
Concrete states over variables $x, y$:

- $x = 1, y = 0$
- $x = 1, y = 1$
- $x = 1, y = 2$
- $x = 2, y = 0$
- $x = 2, y = 1$
- $x = 0, y = 0$

Predicates:

- $p_1 \iff x > y$
- $p_2 \iff y = 0$
```

Abstract Transitions?
Existential Abstraction\(^1\)

<table>
<thead>
<tr>
<th>Definition (Existential Abstraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A model ( \hat{M} = (\hat{S}, \hat{S}_0, \hat{T}) ) is an <strong>existential abstraction</strong> of ( M = (S, S_0, T) ) with respect to ( \alpha : S \to \hat{S} ) iff</td>
</tr>
<tr>
<td>( \exists s \in S_0. \alpha(s) = \hat{s} \Rightarrow \hat{s} \in \hat{S}_0 ) and</td>
</tr>
<tr>
<td>( \exists (s, s') \in T. \alpha(s) = \hat{s} \land \alpha(s') = \hat{s}' \Rightarrow (\hat{s}, \hat{s}') \in \hat{T} ).</td>
</tr>
</tbody>
</table>

---

\(^1\)Clarke, Grumberg, Long: *Model Checking and Abstraction*, ACM TOPLAS, 1994
Minimal Existential Abstractions

There are obviously many choices for an existential abstraction for a given $\alpha$.

Definition (Minimal Existential Abstraction)

A model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{T})$ is the *minimal existential abstraction* of $M = (S, S_0, T)$ with respect to $\alpha : S \rightarrow \hat{S}$ iff

1. $\exists s \in S_0. \alpha(s) = \hat{s} \iff \hat{s} \in \hat{S}_0$ and
2. $\exists (s, s') \in T. \alpha(s) = \hat{s} \land \alpha(s') = \hat{s}' \iff (\hat{s}, \hat{s}') \in \hat{T}$.

This is the most precise existential abstraction.
Existential Abstraction

We write $\alpha(\pi)$ for the abstraction of a path $\pi = s_0, s_1, \ldots$:

$$\alpha(\pi) = \alpha(s_0), \alpha(s_1), \ldots$$
Existential Abstraction

We write $\alpha(\pi)$ for the abstraction of a path $\pi = s_0, s_1, \ldots$:

$$\alpha(\pi) = \alpha(s_0), \alpha(s_1), \ldots$$

Lemma

Let $\hat{M}$ be an existential abstraction of $M$. The abstraction of every path (trace) $\pi$ in $M$ is a path (trace) in $\hat{M}$.

$$\pi \in M \implies \alpha(\pi) \in \hat{M}$$

Proof by induction.
We say that $\hat{M}$ overapproximates $M$. 
Abstracting Properties

Reminder: we are using

- a set of atomic propositions (predicates) $A$, and
- a state-labelling function $L : S \rightarrow \mathcal{P}(A)$

in order to define the meaning of propositions in our properties.
Abstracting Properties

We define an abstract version of it as follows:

- First of all, the negations are pushed into the atomic propositions.
  E.g., we will have

  \[ x = 0 \in A \]

  and

  \[ x \neq 0 \in A \]
Abstracting Properties

- An abstract state \( \hat{s} \) is labelled with \( a \in A \) iff all of the corresponding concrete states are labelled with \( a \).

\[
a \in \hat{L}(\hat{s}) \iff \forall s | \alpha(s) = \hat{s}. a \in L(s)
\]

- This also means that an abstract state may have neither the label \( x = 0 \) nor the label \( x \neq 0 \) – this may happen if it concretizes to concrete states with different labels!
Conservative Abstraction

The keystone is that existential abstraction is conservative for certain properties:

Theorem (Clarke/Grumberg/Long 1994)

Let $\phi$ be a $\forall$CTL* formula where all negations are pushed into the atomic propositions, and let $\hat{M}$ be an existential abstraction of $M$. If $\phi$ holds on $\hat{M}$, then it also holds on $M$.

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

We say that an existential abstraction is conservative for $\forall$CTL* properties. The same result can be obtained for LTL properties.

The proof uses the lemma and is by induction on the structure of $\phi$. The converse usually does not hold.
Conservative Abstraction

We hope: computing $\hat{M}$ and checking $\hat{M} \models \phi$ is easier than checking $M \models \phi$. 
Back to the Example

$p_1, p_2$

$p_1, \neg p_2$

$\neg p_1, p_2$

$p_1, \neg p_2$

$\neg p_1, \neg p_2$
Back to the Example

\[ x = 2 \]
\[ y = 0 \]
\[ p_1, p_2 \]

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

\[ x = 2 \]
\[ y = 1 \]
\[ p_1, \neg p_2 \]

\[ x = 0 \]
\[ y = 0 \]
\[ \neg p_1, p_2 \]

\[ x = 1 \]
\[ y = 1 \]

\[ x = 1 \]
\[ y = 2 \]
\[ \neg p_1, \neg p_2 \]
Back to the Example

\[
\begin{align*}
x &= 2 \quad \Rightarrow \quad y &= 0 \\
x &= 2 \quad \Rightarrow \quad y &= 1 \\
x &= 0 \quad \Rightarrow \quad y &= 0 \\
x &= 1 \quad \Rightarrow \quad y &= 0 \\
x &= 1 \quad \Rightarrow \quad y &= 1 \\
x &= 1 \quad \Rightarrow \quad y &= 2 \\
\end{align*}
\]

\[
\begin{align*}
p_1, p_2 \\
p_1, \neg p_2 \\
\neg p_1, p_2 \\
\neg p_1, \neg p_2
\end{align*}
\]
Back to the Example

\[ x = 2 \]
\[ y = 0 \]

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

\[ p_1, p_2 \]

\[ x = 2 \]
\[ y = 1 \]

\[ p_1, \neg p_2 \]

\[ x = 0 \]
\[ y = 0 \]

\[ \neg p_1, p_2 \]

\[ x = 1 \]
\[ y = 1 \]

\[ \neg p_1, \neg p_2 \]
Back to the Example

$p_1, p_2$

$p_1, \neg p_2$

$\neg p_1, \neg p_2$
Let’s try a Property

Property:

\[ x > y \lor y \neq 0 \iff p_1 \lor \neg p_2 \]
Let’s try a Property

Property:

\[ x > y \lor y \neq 0 \iff p_1 \lor \neg p_2 \]
Another Property

Property:
\[ x > y \iff p_1 \]

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Another Property

Property:

\[ x > y \iff \neg p_1 \]

But: the counterexample is spurious
Another Property

Property:
\[ x > y \iff p_1 \]

But: the counterexample is spurious
Another Property

Property:
\[ x > y \iff p_1 \]

But: the counterexample is spurious.
Microsoft blames most Windows crashes on third party device drivers.

The Windows device driver API is quite complicated.

Drivers are low level C code.

SLAM: Tool to automatically check device drivers for certain errors.

SLAM is shipped with Device Driver Development Kit.

Full detail available at http://research.microsoft.com/slam/
SLIC

- **Finite state language** for defining properties
  - Monitors behavior of C code
  - Temporal safety properties (security automata)
  - familiar C syntax

- Suitable for expressing control-dominated properties
  - e.g., proper sequence of events
  - can track data values
SLIC Example

\[
\text{state} \{ \\
\quad \text{enum} \ {\text{Locked, Unlocked}} \\
\quad \quad s = \text{Unlocked} \\
\}
\]

\[
\text{KeAcquireSpinLock}.\text{entry} \{ \\
\quad \text{if} (s==\text{Locked}) \text{abort}; \\
\quad \text{else} \ s = \text{Locked}; \\
\}
\]

\[
\text{KeReleaseSpinLock}.\text{entry} \{ \\
\quad \text{if} (s==\text{Unlocked}) \text{abort}; \\
\quad \text{else} \ s = \text{Unlocked}; \\
\}
\]
```cpp
state {
    enum {Locked, Unlocked}
    s = Unlocked;
}

KeAcquireSpinLock.entry {
    if (s==Locked) abort;
    else s = Locked;
}

KeReleaseSpinLock.entry {
    if (s==Unlocked) abort;
    else s = Unlocked;
}
```
Refinement Example

do {
        KeAcquireSpinLock ();
        nPacketsOld = nPackets;
        if (request) {
            request = request->Next;
            KeReleaseSpinLock ();
            nPackets++;
        }
    } while(nPackets != nPacketsOld);

KeReleaseSpinLock ();
Refinement Example

Does this code obey the locking rule?

do {
   KeAcquireSpinLock ();
   nPacketsOld = nPackets;
   if (request) {
      request = request->Next;
      KeReleaseSpinLock ();
      nPackets++;
   }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock ();
Refinement Example

do {
    KeAcquireSpinLock();

    if (*) {
        KeReleaseSpinLock();
    }
} while(*);

KeReleaseSpinLock();
Refinement Example

\[
\text{do} \{ \\
    \text{KeAcquireSpinLock}(); \\
    \text{if} \ (*) \{ \\
        \text{KeReleaseSpinLock}(); \\
    \} \\
\} \text{ while}(*) ;
\]

\text{KeReleaseSpinLock}();
Refinement Example

do {
    KeAcquireSpinLock();
    if (*) {
        KeReleaseSpinLock();
    }
} while(*);

KeReleaseSpinLock();
Refinement Example

\[
\begin{align*}
\text{do } & \{ \\
& \text{KeAcquireSpinLock ();} \\
& \text{if ( \ast ) } \{ \\
& \quad \text{KeReleaseSpinLock ();} \\
& \} \text{ \textbf{while( \ast );}} \\
& \text{KeReleaseSpinLock ();} \\
\end{align*}
\]

Is this path concretizable?

Refinement Example

do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
Refinement Example

\[
\text{do } \{
\begin{align*}
\text{KeAcquireSpinLock ();} \\
n\text{PacketsOld} &= n\text{Packets;} \\
\text{if (request) } \{
\begin{align*}
\text{request} &= \text{request} \rightarrow \text{Next}; \\
\text{KeReleaseSpinLock ();} \\
n\text{Packets++;}
\end{align*}
\}
\end{align*}
\} \text{ while (nPackets } \neq \text{ nPacketsOld);}
\]

This path is spurious!
Refinement Example

```
do {
    KeAcquireSpinLock ();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock ();
        nPackets++;
    }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock ();
```

Let’s add the predicate

\[\text{nPacketsOld} == \text{nPackets}\]
Refinement Example

\[
\text{do } \{
\text{KeAcquireSpinLock}();
\text{nPacketsOld} = \text{nPackets};
\text{if}\ (\text{request}) \{ \\
\quad \text{request} = \text{request} \rightarrow \text{Next};
\quad \text{KeReleaseSpinLock}();
\quad \text{nPackets}++; \\
\}
\} \text{while}(\text{nPackets} \neq \text{nPacketsOld});
\]

Let's add the predicate \(\text{nPacketsOld}==\text{nPackets}\)
Refinement Example

do {
    KeAcquireSpinLock ();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock ();
        nPackets++;
    }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock ();

Let’s add the predicate nPacketsOld == nPackets
Refinement Example

\[
\text{do}\{ \\
\text{KeAcquireSpinLock}(); \\
b=true; \\
\text{if} (\ast) \{ \\
\text{KeReleaseSpinLock}(); \\
b=b?false:\ast; \\
\} \\
\} \text{ while}(!b); \\
\text{KeReleaseSpinLock}();
\]
Refinement Example

```c
do {
    KeAcquireSpinLock ();
    b=true;
    if (*) {
        KeReleaseSpinLock ();
        b=b?false:*;
    }
}
while(!b);
KeReleaseSpinLock ();
```
Refinement Example

```c
do {
    KeAcquireSpinLock();
    b=true;
    if (∗) {
        KeReleaseSpinLock();
        b=b?false:∗;
    }
} while( !b );
KeReleaseSpinLock();
```

The property holds!

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Refinement Example

do {
    KeAcquireSpinLock ();
    b=true;
    if (∗) {
        KeReleaseSpinLock ();
        b=b?false:∗;
    }
}
while ( !b );

KeReleaseSpinLock ();
Refinement Example

```c
refinement
{
  KeAcquireSpinLock();
  b=true;
  if (b) {
    KeReleaseSpinLock();
    b=b?false:*;
  }
}
while(!b);
KeReleaseSpinLock();
```
Refinement Example

```
do {
    KeAcquireSpinLock();
    b=true;
    if (*) {
        KeReleaseSpinLock();
        b=b?false:*;
    }
    KeReleaseSpinLock();
} while (!b);

The property holds!
```
Counterexample-guided Abstraction Refinement

- "CEGAR"

- An iterative method to compute a sufficiently precise abstraction

- Initially applied in the context of hardware [Kurshan]
CEGAR Overview

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Counterexample-guided Abstraction Refinement

Claims:

1. This never returns a false error.
2. This never returns a false proof.

3. This is complete for finite-state models.
4. But: no termination guarantee in case of infinite-state systems
Computing Existential Abstractions of Programs

1.) Compute Abstraction
2.) Check Abstraction
3.) Check Feasibility
4.) Refine Predicates

[no error]
OK

feasible
report counterexample

C program
Computing Existential Abstractions of Programs

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i ++;
}
```

C Program
Computing Existential Abstractions of Programs

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i ++;
}
```

C Program

```
void main() {
    bool p1, p2;
    p1 = TRUE;
    p2 = TRUE;
    while (p2)
    {
        p1 = p1 ? FALSE : *;
        p2 = !p2;
    }
}
```

Predicates

\[
p_1 \iff i = 0\]
\[
p_2 \iff \text{even}(i)\]
Computing Existential Abstractions of Programs

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i ++;
}
```

```c
void main() {
    bool p1, p2;
    p1 = TRUE;
    p2 = TRUE;
    while (p2)
        p1 = p1 ? FALSE : *;
        p2 = !p2;
}
```

$\mathbf{p_1} \iff i = 0$

$\mathbf{p_2} \iff \text{even}(i)$
Computing Existential Abstractions of Programs

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i ++;
}
```

C Program

```c
#define even(i) ((i) % 2 == 0)
```

```c
void main() {
    bool p1, p2;
    p1 = TRUE;
    p2 = TRUE;
    while (p2) {
        p1 = p1 ? FALSE : *;
        p2 = !p2;
    }
}
```

Boolean Program

```
p1 ⇐⇒ i = 0
p2 ⇐⇒ even(i)
```

Predicates

Minimal?
Predicate Images

Reminder:

$$\text{Image}(X) = \{s' \in S | \exists s \in X. T(s, s')\}$$

We need

$$\text{Image}(\hat{X}) = \{\hat{s}' \in \hat{S} | \exists \hat{s} \in \hat{X}. \hat{T}(\hat{s}, \hat{s}')\}$$

$$\text{Image}(\hat{X})$$ is equivalent to

$$\{\hat{s}, \hat{s}' \in \hat{S}^2 | \exists s, s' \in S^2. \alpha(s) = \hat{s} \land \alpha(s') = \hat{s}' \land T(s, s')\}$$

This is called the predicate image of $T$.
Let’s take existential abstraction seriously

Basic idea: with $n$ predicates, there are $2^n \cdot 2^n$ possible abstract transitions

Let’s just check them!
Enumeration: Example

Predicates

\[ p_1 \iff i = 1 \]
\[ p_2 \iff i = 2 \]
\[ p_3 \iff \text{even}(i) \]
Enumeration: Example

Predicates

\begin{align*}
p_1 & \iff i = 1 \\
p_2 & \iff i = 2 \\
p_3 & \iff \text{even}(i)
\end{align*}

Basic Block

\begin{align*}
i & \leftarrow i + 1 \\
\end{align*}
Enumeration: Example

Predicates

\[ p_1 \iff i = 1 \]
\[ p_2 \iff i = 2 \]
\[ p_3 \iff \text{even}(i) \]

Basic Block

\[ i++; \]

\[ T \]

\[ i' = i + 1 \]
**Enumeration: Example**

**Predicates**

- $p_1 \iff i = 1$
- $p_2 \iff i = 2$
- $p_3 \iff \text{even}(i)$

**Basic Block**

```
i++;```

**$T$**

```
i' = i + 1```

---

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p'_1$</th>
<th>$p'_2$</th>
<th>$p'_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## Enumeration: Example

### Predicates

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Basic Block

```plaintext
i++;
```

### Transformation

\[ i' = i + 1 \]
Enumeration: Example

Predicates

$p_1 \iff i = 1$
$p_2 \iff i = 2$
$p_3 \iff \text{even}(i)$

Basic Block

$i \text{++}$;

$T$

$i' = i + 1$

Query to Solver

$i \neq 1 \land i \neq 2 \land \overline{\text{even}(i)} \land$

$i' = i + 1 \land$

$i' \neq 1 \land i' \neq 2 \land \overline{\text{even}(i')}$
Enumeration: Example

Predicates

- \( p_1 \iff i = 1 \)
- \( p_2 \iff i = 2 \)
- \( p_3 \iff \text{even}(i) \)

Basic Block

\[ i++; \]

Basic Block 

\[ T \]

\[ i' = i + 1 \]

Query to Solver

\[ i \neq 1 \land i \neq 2 \land \overline{\text{even}(i)} \land \\
  i' = i + 1 \land \\
  i' \neq 1 \land i' \neq 2 \land \overline{\text{even}(i')} \]
Enumeration: Example

Predicates

\[
\begin{align*}
    p_1 & \iff i = 1 \\
    p_2 & \iff i = 2 \\
    p_3 & \iff \text{even}(i)
\end{align*}
\]

Basic Block

\[
i++;
\]

\[
T
\]

\[
i' = i + 1
\]

Query to Solver

\[
i \neq 1 \land i \neq 2 \land \text{even}(i) \land \\
i' = i + 1 \land \\
i' \neq 1 \land i' \neq 2 \land \text{even}(i')
\]
**Enumeration: Example**

**Predicates**

\[
p_1 \iff i = 1 \\
p_2 \iff i = 2 \\
p_3 \iff \text{even}(i)
\]

**Basic Block**

\[
i++; \\
i' = i + 1
\]

**Query to Solver**

\[
i \neq 1 \land i \neq 2 \land \text{even}(i) \land \\
i' = i + 1 \land \\
i' \neq 1 \land i' \neq 2 \land \text{even}(i')
\]
Enumeration: Example

_predicates_

\[ p_1 \iff i = 1 \]
\[ p_2 \iff i = 2 \]
\[ p_3 \iff \text{even}(i) \]

Basic Block

\[ i++; \]

\[ T \]

\[ i' = i + 1 \]

Query to Solver

\[ i \neq 1 \land i \neq 2 \land \text{even}(i) \land i' = i + 1 \land i' \neq 1 \land i' \neq 2 \land \text{even}(i') \]

… and so on …
Predicat Images

- Computing the minimal existential abstraction can be way too slow

  - Use an over-approximation instead
    - ✔ Fast(er) to compute
    - ✗ But has additional transitions

- Examples:
  - Cartesian approximation (SLAM)
  - FastAbs (SLAM)
  - Lazy abstraction (Blast)
  - Predicate partitioning (VCEGAR)
Checking the Abstract Model

C program

1.) Compute Abstraction
2.) Check Abstraction
3.) Check Feasibility
4.) Refine Predicates

[no error]
OK

[feasible]
report counterexample
Checking the Abstract Model

- No more integers!

- But:
  - All control flow constructs, including function calls
  - (more) non-determinism

✔ BDD-based model checking now scales
Variables

VAR b0_argc_ge_1: boolean;   -- argc >= 1
VAR b1_argc_le_2147483646: boolean;   -- argc <= 2147483646
VAR b2: boolean;   -- argv[argc] == NULL
VAR b3_nmemb_ge_r: boolean;   -- nmemb >= r
VAR b4: boolean;   -- p1 == &array[0]
VAR b5_i_ge_8: boolean;   -- i >= 8
VAR b6_i_ge_s: boolean;   -- i >= s
VAR b7: boolean;   -- 1 + i >= 8
VAR b8: boolean;   -- 1 + i >= s
VAR b9_s_gt_0: boolean;   -- s > 0
VAR b10_s_gt_1: boolean;   -- s > 1
Finite-State Model Checkers: SMV

② Control Flow

— program counter: 56 is the "terminating" PC
VAR PC: 0..56;
ASSIGN init(PC):=0; — initial PC

ASSIGN next(PC):=case
  PC=0: 1; — other
  PC=1: 2; — other
  . . .
  PC=19: case — goto (with guard)
    guard19: 26;
    1: 20;
    esac;
  . . .
Finite-State Model Checkers: SMV

**Data**

\[
\text{TRANS } (PC=0) \rightarrow \text{next}(b0_{\text{argc} \geq 1}) = b0_{\text{argc} \geq 1}
\& \text{next}(b1_{\text{argc} \leq 213646}) = b1_{\text{argc} \leq 21646}
\& \text{next}(b2) = b2
\& (!b30 \mid b36)
\& (!b17 \mid !b30 \mid b42)
\& (!b30 \mid !b42 \mid b48)
\& (!b17 \mid !b30 \mid !b42 \mid b54)
\& (!b54 \mid b60)
\]

\[
\text{TRANS } (PC=1) \rightarrow \text{next}(b0_{\text{argc} \geq 1}) = b0_{\text{argc} \geq 1}
\& \text{next}(b1_{\text{argc} \leq 214646}) = b1_{\text{argc} \leq 214746}
\& \text{next}(b2) = b2
\& \text{next}(b3_{\text{nmemb} \geq r}) = b3_{\text{nmemb} \geq r}
\& \text{next}(b4) = b4
\& \text{next}(b5_{i \geq 8}) = b5_{i \geq 8}
\& \text{next}(b6_{i \geq s}) = b6_{i \geq s}
\]

...
Finite-State Model Checkers: SMV

Property

— the specification

— file main.c line 20 column 12
— function c::very_buggy_function

SPEC AG ((PC=51) ⇒ !b23)
Finite-State Model Checkers: SMV

- If the property holds, we can terminate

- If the property fails, SMV generates a **counterexample** with an assignment for all variables, including the PC
Simulating the Counterexample

1.) Compute Abstraction
2.) Check Abstraction
3.) Check Feasibility
4.) Refine Predicates

report counterexample

C program

[feasible]

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Lazy Abstraction

- The progress guarantee is only valid if the minimal existential abstraction is used.

- Thus, distinguish spurious transitions from spurious prefixes.

- Refine spurious transitions separately to obtain minimal existential abstraction

- SLAM: Constrain
Lazy Abstraction

▶ One more observation: each iteration only causes only minor changes in the abstract model

▶ Thus, use “incremental Model Checker”, which retains the set of reachable states between iterations (BLAST)
Example Simulation

```c
int main() {
    int x, y;
    y=1;
    x=1;
    if (y > x)
        y--;  
    else
        y++;
    assert(y > x);
}
```

```c
bool b0; // y > x
b0=*
if (b0)
    b0=*;
else
    b0=*;
assert(b0);
}
```
Example Simulation

```c
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y--; 
    else
        y++;
    assert(y>x);
}
```

```c
main() {
    bool b0; // y>x
    b0=*
    if (b0)
        b0=*
    else
        b0=*
    assert(b0);
}
```
Example Simulation

```c
int main() {
    int x, y;
    y = 1;
    x = 1;
    if (y > x)
        y--;  // This line is not executed
    else
        y++;  // This line is executed
    assert(y > x);
}
```
Example Simulation

```c
int main() {
    int x, y;
    y = 1;
    x = 1;
    if (y > x)
        y--; // y = 0
    else
        y++; // y = 2
    assert(y > x);
}
```

We now do a path test, so convert to SSA.
Example Simulation

```c
int main() {
    int x, y;
    y = 1;
    x = 1;
    if (y > x) {
        y2 = y1 - 1;
    } else {
        y++;
    }
    assert(y2 > x);
}
```

This is UNSAT, so \( \hat{\pi} \) is spurious.
Example Simulation

```c
int main() {
    int x, y;
    y = 1;
    x = 1;
    if (y > x)
        y = y - 1;
    else
        y++;
    assert(y > x);
}
```

\[
y_1 = 1 \land x_1 = 1 \land y_1 > x_1 \land y_2 = y_1 - 1 \land \neg (y_2 > x_0)
\]

This is UNSAT, so \( \hat{\pi} \) is spurious.
Example Simulation

```c
int main() {
    int x, y;
y_1=1;
x_1=1;
    if (y_1 > x_1)
        y_2 = y_1 - 1;
    else
        y++;
    assert(y_2 > x_1);
}
```

This is UNSAT, so \( \hat{\pi} \) is spurious.
Refining the Abstraction

1.) Compute Abstraction
2.) Check Abstraction
3.) Check Feasibility
4.) Refine Predicates

[no error]
OK

C program

[feasible]
report counterexample
Manual Proof!

```c
int main() {
    int x, y;
    y = 1;
    x = 1;

    if (y > x)
        y--;  
    else
        y++;

    assert(y > x);
}
```

This proof uses strongest post-conditions

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Manual Proof!

```c
int main() {
    int x, y;
    y = 1;
    { y = 1 }
    x = 1;
    if (y > x)
        y--; 
    else
        y++; 
    assert(y > x);
}
```

This proof uses strongest post-conditions

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Manual Proof!

```c
int main() {
    int x, y;
    y = 1;
    \{ y = 1 \}
    x = 1;
    \{ x = 1 \land y = 1 \}
    if (y > x)
        y--;  
    else
        y++;  

    assert(y > x);
}
```

This proof uses strongest post-conditions.
int main() {
    int x, y;
    y=1;
    {y = 1}
    x=1;
    {x = 1 ∧ y = 1}
    if (y>x)
        y--;  // Original: y--; changed to y- to fix the typo
    else
        {x = 1 ∧ y = 1 ∧ ¬y > x}
        y++;

    assert(y>x);
}

This proof uses strongest post-conditions.

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
Manual Proof!

```c
int main() {
    int x, y;
    y=1;
    { y = 1 }
    x=1;
    { x = 1 ∧ y = 1 }
    if (y > x)
        y--;  
    else
        { x = 1 ∧ y = 1 ∧ ¬y > x }
        y++;
        { x = 1 ∧ y = 2 ∧ y > x }
    assert(y>x);
}
```

This proof uses strongest post-conditions
An Alternative Proof

```c
int main() {
    int x, y;
    y = 1;

    x = 1;

    if (y > x)
        y--;  
    else
        y++;

    assert(y > x);
}
```
An Alternative Proof

```c
int main() {
    int x, y;
    y = 1;
    x = 1;
    if (y > x)
        y--;  
    else
        y++;  
    {y > x}
    assert(y > x);
}
```

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
An Alternative Proof

```c
int main() {
    int x, y;
    y = 1;
    x = 1;

    if (y > x)
        y--; 
    else
        { y + 1 > x }
        y++; 
        { y > x }
    assert(y>x);
}
```
An Alternative Proof

```c
int main() {
    int x, y;
    y = 1;

    x = 1;
    {¬y > x ⇒ y + 1 > x}
    if (y>x)
        y--;  
    else
        {y + 1 > x}
        y++;  
    {y > x}
    assert(y>x);
}
```

D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial
An Alternative Proof

```c
int main() {
    int x, y;
    y = 1;
    {¬y > 1 ⇒ y + 1 > 1}
    x = 1;
    {¬y > x ⇒ y + 1 > x}
    if (y > x)
        y--;  
    else
        {y + 1 > x}
        y++;  
        {y > x}
    assert(y > x);
}
```
An Alternative Proof

```c
int main() {
    int x, y;
    y = 1;
    {¬y > 1 ⇒ y + 1 > 1}
    x = 1;
    {¬y > x ⇒ y + 1 > x}
    if (y > x)
        y--;    
    else
        {y + 1 > x}
        y++;
    {y > x}
    assert(y > x);
}
```

We are using weakest pre-conditions here

- $wp(x := E, P) = P[x/E]$
- $wp(S; T, Q) = wp(S, wp(T, Q))$
- $wp(if(c) A else B, P) = (B ⇒ wp(A, P)) ∧ (¬B ⇒ wp(B, P))$

The proof for the "true" branch is missing
Refinement Algorithms

Using WP

1. Start with failed guard $G$
2. Compute $wp(G)$ along the path

Using SP

1. Start at beginning
2. Compute $sp(...)$ along the path

- Both methods eliminate the trace
- Advantages/disadvantages?
Recall the decision problem we build for simulating paths:

\[ x_1 = 10 \land y_1 = x_1 + 10 \land y_2 = y_1 + 10 \land y_2 \neq 30 \]
Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

\[ x_1 = 10 \land y_1 = x_1 + 10 \land y_2 = y_1 + 10 \land y_2 \neq 30 \]

\[ \Rightarrow x_1 = 10 \]
Recall the decision problem we build for simulating paths:

\[
x_1 = 10 \land y_1 = x_1 + 10 \land y_2 = y_1 + 10 \land y_2 \neq 30
\]

\[\Rightarrow x_1 = 10 \quad \Rightarrow y_1 = 20\]
Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

\[ x_1 = 10 \land y_1 = x_1 + 10 \land y_2 = y_1 + 10 \land y_2 \neq 30 \]

\[ \Rightarrow x_1 = 10 \quad \Rightarrow y_1 = 20 \quad \Rightarrow y_2 = 30 \]
Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

\[ x_1 = 10 \land y_1 = x_1 + 10 \land y_2 = y_1 + 10 \land y_2 \neq 30 \]

\[ \Rightarrow x_1 = 10 \land \Rightarrow y_1 = 20 \land \Rightarrow y_2 = 30 \land \Rightarrow \text{false} \]
Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

\[ A_1 \quad \begin{array}{c} x_1 = 10 \\ \Rightarrow x_1 = 10 \end{array} \]
\[ A_2 \quad \begin{array}{c} y_1 = x_1 + 10 \\ \Rightarrow y_1 = 20 \end{array} \]
\[ A_3 \quad \begin{array}{c} y_2 = y_1 + 10 \\ \Rightarrow y_2 = 30 \end{array} \]
\[ A_4 \quad y_2 \neq 30 \quad \Rightarrow \text{false} \]
Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

\[
\begin{align*}
A_1 & : x_1 = 10 \\
A_2 & : y_1 = x_1 + 10 \\
A_3 & : y_2 = y_1 + 10 \\
A_4 & : y_2 \neq 30
\end{align*}
\]

\[
\begin{align*}
\Rightarrow x_1 & = 10 \\
\Rightarrow y_1 & = 20 \\
\Rightarrow y_2 & = 30 \\
\Rightarrow \text{false}
\end{align*}
\]
For a path with $n$ steps:

\[
\begin{align*}
A_1 & \quad \Rightarrow A'_1 \\
A_2 & \quad \Rightarrow A'_2 \\
A_3 & \quad \Rightarrow A'_3 \\
\vdots & \quad \vdots \\
A_n & \quad \Rightarrow A'_{n-1} \\
\text{true} & \quad \Rightarrow \text{false}
\end{align*}
\]
Predicate Refinement for Paths

For a path with \( n \) steps:

\[
\begin{array}{ccccccc}
A_1 & | & A_2 & | & A_3 & | & \ldots & | & A_n \\
\text{true} & \Rightarrow & A'_1 & \Rightarrow & A'_2 & \Rightarrow & A'_3 & \Rightarrow & A'_{n-1} & \Rightarrow \text{false}
\end{array}
\]

- Given \( A_1, \ldots, A_n \) with \( \bigwedge_i A_i = \text{false} \)
- \( A'_0 = \text{true} \) and \( A'_n = \text{false} \)
- \( (A'_{i-1} \land A_i) \Rightarrow A'_i \) for \( i \in \{1, \ldots, n\} \)
Predicate Refinement for Paths

For a path with $n$ steps:

\[
A_1 \quad | \quad A_2 \quad | \quad A_3 \quad | \quad \ldots \quad | \quad A_n
\]

true $\Rightarrow A_1' \Rightarrow A_2' \Rightarrow A_3' \Rightarrow A_{n-1}' \Rightarrow false$

- Given $A_1, \ldots, A_n$ with $\bigwedge_i A_i = false$
- $A'_0 = true$ and $A'_n = false$
- $(A'_{i-1} \land A_i) \Rightarrow A'_i$ for $i \in \{1, \ldots, n\}$
- Finally, $\text{Vars}(A'_i) \subseteq (\text{Vars}(A_1 \ldots A_i) \cap \text{Vars}(A_{i+1} \ldots A_n))$
Predicate Refinement for Paths

Special case $n = 2$:

- $A \land B = \text{false}$
- $A \Rightarrow A'$
- $A' \land B = \text{false}$
- $\text{Vars}(A') \subseteq (\text{Vars}(A) \cap \text{Vars}(B))$
Predicate Refinement for Paths

Special case $n = 2$:

- $A \land B = \text{false}$
- $A \Rightarrow A'$
- $A' \land B = \text{false}$
- $\text{Vars}(A') \subseteq (\text{Vars}(A) \cap \text{Vars}(B))$

W. Craig’s Interpolation theorem (1957): such an $A'$ exists for any first-order, inconsistent $A$ and $B$. 
Predicate Refinement with Craig Interpolants

✓ For propositional logic, a propositional Craig Interpolant can be extracted from a resolution proof (→ SAT!) in linear time.

✓ Interpolating solvers available for linear arithmetic over the reals and integer difference logic with uninterpreted functions.

✗ Not possible for every fragment of FOL:

\[ x = 2y \quad \text{and} \quad x = 2z + 1 \quad \text{with} \quad x, y, z \in \mathbb{Z} \]
Predicate Refinement with Craig Interpolants

For propositional logic, a propositional Craig Interpolant can be extracted from a resolution proof ($\rightarrow$ SAT!) in linear time.

Interpolating solvers available for linear arithmetic over the reals and integer difference logic with uninterpreted functions.

Not possible for every fragment of FOL:

$$x = 2y \quad \text{and} \quad x = 2z + 1 \quad \text{with} \quad x, y, z \in \mathbb{Z}$$

The interpolant is “$x$ is even”
Craig Interpolation for Linear Inequalities

\[ 0 \leq x \leq y \]
\[ 0 \leq c_1 x + c_2 y \quad \text{with} \quad 0 \leq c_1, c_2 \]

- “Cutting-planes”
- Naturally arise in Fourier-Motzkin or Simplex
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]

\[ 0 \leq y - z - 1 \]

\[ 0 \leq z - x \]
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]

\[ 0 \leq y - z - 1 \quad 0 \leq z - x \]

\[ 0 \leq y - x - 1 \]
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]

\[
\begin{align*}
0 &\leq y - z - 1 \\
0 &\leq z - x \\
0 &\leq y - x - 1 \\
0 &\leq x - y \\
0 &\leq -1
\end{align*}
\]
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]

\[ 0 \leq y - z - 1 \]
\[ 0 \leq z - x \]
\[ 0 \leq y - z - 1 \]
\[ 0 \leq 0 \]
\[ 0 \leq y - x - 1 \]
\[ 0 \leq x - y \]
\[ 0 \leq -1 \]
\[ 0 \leq x - z - 1 \]
\[ 0 \leq x - y \]
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]

\[ 0 \leq y - z - 1 \quad 0 \leq z - x \]

\[ 0 \leq y - x - 1 \quad 0 \leq y - z - 1 \]

\[ 0 \leq x - y \quad 0 \leq x - y \]

\[ 0 \leq -1 \quad 0 \leq -1 \]

\[ 0 \leq x - z - 1 \quad z - x \leq -1 \]
Example

\[ A = (0 \leq x - y) \land (0 \leq y - z - 1) \]

\[ B = (0 \leq z - x) \]

\[ 0 \leq y - z - 1 \]
\[ 0 \leq z - x \]
\[ 0 \leq y - z - 1 \]
\[ 0 \leq 0 \]

\[ 0 \leq y - x - 1 \]
\[ 0 \leq z - x \]
\[ 0 \leq y - z - 1 \]
\[ 0 \leq 0 \]
\[ 0 \leq x - y \]
\[ 0 \leq x - y \]
\[ 0 \leq x - z - 1 \]
\[ \iff z - x \leq -1 \]

Just sum the inequalities from \( A \), and you get an interpolant!
Approximating Loop Invariants: SP

```c
int x, y;

x = y = 0;

while (x != 10) {
    x ++;
    y ++;
}

assert (y == 10);
```

The SP refinement results in

\[
sp(x=y=0, \text{true}) = x = 0 \land y = 0
\]

It won't work if we replace 10 by \( n \).
Approximating Loop Invariants: SP

```c
int x, y;

x=y=0;

while (x != 10) {
    x++;
    y++;
}

assert (y == 10);
```

The SP refinement results in

\[
\begin{align*}
sp(x=y=0, true) &= x = 0 \land y = 0 \\
sp(x++; y++, \ldots) &= x = 1 \land y = 1
\end{align*}
\]

... 10 iterations required to prove the property.

It won't work if we replace 10 by \(n\).
Approximating Loop Invariants: SP

```c
int x, y;

x = y = 0;

while (x != 10) {
    x ++;
    y ++;
}

assert (y == 10);
```

The SP refinement results in

\[
\begin{align*}
sp(x=y=0, \text{true}) & = x = 0 \land y = 0 \\
sp(x++; y++; \ldots) & = x = 1 \land y = 1 \\
sp(x++; y++; \ldots) & = x = 2 \land y = 2 \\
& \quad \ldots
\end{align*}
\]

10 iterations required to prove the property.

It won't work if we replace 10 by \( n \).
The SP refinement results in

\[ sp(x=y=0, \text{true}) = x = 0 \land y = 0 \]
\[ sp(x++; y++, \ldots) = x = 1 \land y = 1 \]
\[ sp(x++; y++, \ldots) = x = 2 \land y = 2 \]
\[ sp(x++; y++, \ldots) = x = 3 \land y = 3 \]
\[ \ldots \]

★ 10 iterations required to prove the property.
★ It won’t work if we replace 10 by \( n \).
Approximating Loop Invariants: WP

```c
int x, y;

x = y = 0;

while (x != 10) {
    x ++;
    y ++;
}

assert (y == 10);
```

The WP refinement results in

\[
wp(x == 10, y \neq 10) = y \neq 10 \land x = 10
\]

Also requires 10 iterations.

It won't work if we replace 10 by \( n \).
The WP refinement results in

\[
wp(x==10, y \neq 10) = y \neq 10 \land x = 10
\]

\[
wp(x++; y++, \ldots) = y \neq 9 \land x = 9
\]
Approximating Loop Invariants: WP

```
i nt x, y;
x = y = 0;
while (x != 10) {
    x ++;
    y ++;
}
assert (y == 10);
```

The WP refinement results in

\[
\begin{align*}
wp(x == 10, y \neq 10) &= y \neq 10 \land x = 10 \\
wp(x++; y++; \ldots) &= y \neq 9 \land x = 9 \\
wp(x++; y++; \ldots) &= y \neq 8 \land x = 8
\end{align*}
\]

Also requires 10 iterations.

It won't work if we replace 10 by \( n \).
int x, y;

x=y=0;

while (x!=10) {
    x ++;
    y ++;
}

assert (y==10);

The WP refinement results in

\[ wp(x==10, y \neq 10) = y \neq 10 \land x = 10 \]
\[ wp(x++; y++, \ldots) = y \neq 9 \land x = 9 \]
\[ wp(x++; y++, \ldots) = y \neq 8 \land x = 8 \]
\[ wp(x++; y++, \ldots) = y \neq 7 \land x = 7 \]
approximating loop invariants: wp

```c
int x, y;
x = y = 0;
while (x != 10) {
    x ++;
y ++;
}
assert (y == 10);
```

the wp refinement results in

\[
wp(x == 10, y \ne 10) = y \ne 10 \land x = 10
\]

\[
wp(x++; y++; \ldots) = y \ne 9 \land x = 9
\]

\[
wp(x++; y++; \ldots) = y \ne 8 \land x = 8
\]

\[
wp(x++; y++; \ldots) = y \ne 7 \land x = 7
\]

\ldots

also requires 10 iterations.

\times it won't work if we replace 10 by \(n\).
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

\[
\begin{align*}
x_1 &= 0 \\
y_1 &= 0
\end{align*}
\]
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

1st It.

\[ x_1 = 0 \]
\[ y_1 = 0 \]
\[ x_2 = x_1 + 1 \]
\[ y_2 = y_1 + 1 \]
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

<table>
<thead>
<tr>
<th>x₁ = 0</th>
<th>y₁ = 0</th>
<th>1st It.</th>
<th>2nd It.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₂ = x₁ + 1</td>
<td>y₂ = y₁ + 1</td>
<td>x₂ ≠ 10</td>
<td>x₃ = x₂ + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y₃ = y₂ + 1</td>
<td></td>
</tr>
</tbody>
</table>
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

\[
\begin{align*}
x_1 &= 0 \\
y_1 &= 0
\end{align*}
\]

1st It. 
\[
\begin{align*}
x_1 &\neq 10 \\
x_2 &= x_1 + 1 \\
y_2 &= y_1 + 1
\end{align*}
\]

2nd It. 
\[
\begin{align*}
x_2 &\neq 10 \\
x_3 &= x_2 + 1 \\
y_3 &= y_2 + 1
\end{align*}
\]

3rd It. 
\[
\begin{align*}
x_3 &\neq 10 \\
x_4 &= x_3 + 1 \\
y_4 &= y_3 + 1
\end{align*}
\]

Assertion 
\[
\begin{align*}
x_1 &= 0 \\
y_1 &= 0 \\
x_2 &= 1 \\
y_2 &= 1 \\
x_3 &= 2 \\
y_3 &= 2 \\
x_4 &= 3 \\
y_4 &= 3
\end{align*}
\]
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

\[
\begin{align*}
\text{1st It.} & : & x_1 & \neq 10 & & & & x_1 = 0 \\
& & x_2 & = x_1 + 1 & & & & x_2 = x_1 + 1 \\
& & y_2 & = y_1 + 1 & & & & y_2 = y_1 + 1 \\
\text{2nd It.} & : & x_2 & \neq 10 & & & & x_2 = x_1 + 1 \\
& & x_3 & = x_2 + 1 & & & & x_3 = x_2 + 1 \\
& & y_3 & = y_2 + 1 & & & & y_3 = y_2 + 1 \\
\text{3rd It.} & : & x_3 & \neq 10 & & & & x_3 = x_2 + 1 \\
& & x_4 & = x_3 + 1 & & & & x_4 = x_3 + 1 \\
& & y_4 & = y_3 + 1 & & & & y_4 = y_3 + 1 \\
\text{Assertion} & : & x_4 & = 10 & & & & x_4 = 10 \\
& & y_4 & \neq 10 & & & & y_4 \neq 10
\end{align*}
\]
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

<table>
<thead>
<tr>
<th>x₁</th>
<th>y₁</th>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₁ = 0</td>
<td>y₁ = 0</td>
<td>x₁ ≠ 10</td>
<td>x₂ = x₁ + 1</td>
<td>x₃ ≠ 10</td>
<td>x₄ = x₃ + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x₂ = x₁ + 1</td>
<td>x₃ = x₂ + 1</td>
<td>y₄ = y₃ + 1</td>
<td>x₄ = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y₂ = y₁ + 1</td>
<td>y₃ = y₂ + 1</td>
<td></td>
<td>y₄ ≠ 10</td>
</tr>
</tbody>
</table>
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

<table>
<thead>
<tr>
<th></th>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_1 \neq 10$</td>
<td>$x_2 \neq 10$</td>
<td>$x_3 \neq 10$</td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$x_4 = x_3 + 1$</td>
<td></td>
</tr>
<tr>
<td>$y_2$</td>
<td>$y_2 = y_1 + 1$</td>
<td>$y_3 = y_2 + 1$</td>
<td>$y_4 = y_3 + 1$</td>
<td></td>
</tr>
<tr>
<td>$y_1$</td>
<td>$y_1 = 0$</td>
<td>$y_2 = 1$</td>
<td>$y_4 \neq 10$</td>
<td></td>
</tr>
</tbody>
</table>
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

<table>
<thead>
<tr>
<th></th>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_1 \neq 10$</td>
<td>$x_2 \neq 10$</td>
<td>$x_3 \neq 10$</td>
<td>$x_4 = 10$</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$x_4 = x_3 + 1$</td>
<td>$y_4 \neq 10$</td>
</tr>
<tr>
<td></td>
<td>$y_2 = y_1 + 1$</td>
<td>$y_3 = y_2 + 1$</td>
<td>$y_4 = y_3 + 1$</td>
<td></td>
</tr>
<tr>
<td>$x_1$</td>
<td>$x_1 = 0$</td>
<td>$x_2 = 1$</td>
<td>$x_3 = 2$</td>
<td></td>
</tr>
<tr>
<td>$y_1$</td>
<td>$y_1 = 0$</td>
<td>$y_2 = 1$</td>
<td>$y_3 = 2$</td>
<td></td>
</tr>
</tbody>
</table>
Consider an SSA-unwinding with 3 loop iterations:

\[
\begin{align*}
\text{1st It.} & : & x_1 & \neq 10 \\
& & x_2 & = x_1 + 1 \\
& & y_2 & = y_1 + 1 \\
\text{2nd It.} & : & x_2 & \neq 10 \\
& & x_3 & = x_2 + 1 \\
& & y_3 & = y_2 + 1 \\
\text{3rd It.} & : & x_3 & \neq 10 \\
& & x_4 & = x_3 + 1 \\
& & y_4 & = y_3 + 1 \\
\text{Assertion} & : & x_4 & = 10 \\
& & y_4 & \neq 10
\end{align*}
\]
What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

<table>
<thead>
<tr>
<th></th>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = 0$</td>
<td>$x_1 \neq 10$</td>
<td>$x_2 \neq 10$</td>
<td>$x_3 \neq 10$</td>
<td>$x_4 = 10$</td>
</tr>
<tr>
<td>$y_1 = 0$</td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$x_4 = x_3 + 1$</td>
<td>$y_4 \neq 10$</td>
</tr>
<tr>
<td></td>
<td>$y_2 = y_1 + 1$</td>
<td>$y_3 = y_2 + 1$</td>
<td>$y_4 = y_3 + 1$</td>
<td></td>
</tr>
<tr>
<td>$x_1 = 0$</td>
<td>$x_2 = 1$</td>
<td>$x_3 = 2$</td>
<td>$x_4 = 3$</td>
<td></td>
</tr>
<tr>
<td>$y_1 = 0$</td>
<td>$y_2 = 1$</td>
<td>$y_3 = 2$</td>
<td>$y_4 = 3$</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{This proof will produce the same predicates as SP.} \]
Split Provers

Idea:

- Each prover \( \mathcal{P}_i \) only knows \( A_i \), but they exchange facts
- We require that each prover only exchanges facts with common symbols
- Plus, we restrict the facts exchanged to some language \( \mathcal{L} \)
Back to the Example

Restriction to language $\mathcal{L} = \text{“no new constants”}$:

<table>
<thead>
<tr>
<th>$x_1 = 0$</th>
<th>$y_1 = 0$</th>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x_1 \neq 10$</td>
<td>$x_2 \neq 10$</td>
<td>$x_3 \neq 10$</td>
<td>$x_4 = 10$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$x_4 = x_3 + 1$</td>
<td>$y_4 \neq 10$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y_2 = y_1 + 1$</td>
<td>$y_3 = y_2 + 1$</td>
<td>$y_4 = y_3 + 1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Back to the Example

Restriction to language $\mathcal{L} = \text{“no new constants”}:

<table>
<thead>
<tr>
<th></th>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_1 \neq 10$</td>
<td>$x_2 \neq 10$</td>
<td>$x_3 \neq 10$</td>
<td>$x_4 = 10$</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$x_4 = x_3 + 1$</td>
<td>$y_4 \neq 10$</td>
</tr>
<tr>
<td></td>
<td>$y_2 = y_1 + 1$</td>
<td>$y_3 = y_2 + 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x_1 = 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y_1 = 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Back to the Example

Restriction to language $\mathcal{L} =$ “no new constants”:

$\begin{align*}
x_1 &= 0 \\
y_1 &= 0
\end{align*}$

$\begin{align*}
x_1 &\neq 10 \\
x_2 &= x_1 + 1 \\
y_2 &= y_1 + 1
\end{align*}$

$\begin{align*}
x_2 &\neq 10 \\
x_3 &= x_2 + 1 \\
y_3 &= y_2 + 1
\end{align*}$

$\begin{align*}
x_3 &\neq 10 \\
x_4 &= x_3 + 1 \\
y_4 &= y_3 + 1
\end{align*}$

$\begin{align*}
x_4 &= 10 \\
y_4 &\neq 10
\end{align*}$
Back to the Example

Restriction to language $\mathcal{L} = “\text{no new constants}”: 

<table>
<thead>
<tr>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = 0$</td>
<td>$x_1 \neq 10$</td>
<td>$x_2 \neq 10$</td>
<td>$x_3 \neq 10$</td>
</tr>
<tr>
<td>$y_1 = 0$</td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$x_4 = x_3 + 1$</td>
</tr>
<tr>
<td>$x_1 = 0$</td>
<td>$x_2 = 1$</td>
<td>$x_3 = 2$</td>
<td></td>
</tr>
<tr>
<td>$y_1 = 0$</td>
<td>$y_2 = 1$</td>
<td>$y_3 = 2$</td>
<td></td>
</tr>
</tbody>
</table>
Back to the Example

Restriction to language $\mathcal{L} = \text{“no new constants”}$:

1st It.  \[
\begin{align*}
x_1 &= 0 \\
y_1 &= 0 \\
x_2 &= x_1 + 1 \\
y_2 &= y_1 + 1
\end{align*}
\]

2nd It.  \[
\begin{align*}
x_2 &= 1 \\
y_2 &= 1 \\
x_3 &= x_2 + 1 \\
y_3 &= y_2 + 1
\end{align*}
\]

3rd It.  \[
\begin{align*}
x_3 &= 2 \\
y_3 &= 2 \\
x_4 &= x_3 + 1 \\
y_4 &= y_3 + 1
\end{align*}
\]

Assertion  \[
\begin{align*}
x_4 &= 10 \\
y_4 &\neq 10
\end{align*}
\]
Back to the Example

Restriction to language $\mathcal{L} =$ “no new constants”:

1st It.

\[
\begin{align*}
x_1 &= 0 \\
y_1 &= 0 \\
x_2 &= x_1 + 1 \\
y_2 &= y_1 + 1
\end{align*}
\]

2nd It.

\[
\begin{align*}
x_2 &= x_1 + 1 \\
y_2 &= y_1 + 1 \\
x_3 &= x_2 + 1 \\
y_3 &= y_2 + 1
\end{align*}
\]

3rd It.

\[
\begin{align*}
x_3 &= x_2 + 1 \\
y_3 &= y_2 + 1 \\
x_4 &= x_3 + 1 \\
y_4 &= y_3 + 1
\end{align*}
\]

Assertion

\[
\begin{align*}
x_4 &= 10 \\
y_4 &\neq 10
\end{align*}
\]
Back to the Example

Restriction to language $\mathcal{L} = \text{"no new constants"}:

<table>
<thead>
<tr>
<th>1st It.</th>
<th>2nd It.</th>
<th>3rd It.</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = 0$</td>
<td>$x_1 \not= 10$</td>
<td>$x_2 \not= 10$</td>
<td>$x_3 \not= 10$</td>
</tr>
<tr>
<td>$y_1 = 0$</td>
<td>$x_2 = x_1 + 1$</td>
<td>$x_3 = x_2 + 1$</td>
<td>$y_3 = y_2 + 1$</td>
</tr>
<tr>
<td>$x_1 = 0$</td>
<td>$x_2 = 1$</td>
<td>$x_3 = y_3$</td>
<td>$x_4 = y_4$</td>
</tr>
</tbody>
</table>
The language restriction forces the solver to generalize!

Algorithm:

- If the proof fails, increase $L$!
- If we fail to get a sufficiently strong invariant, increase $n$.

This does work if we replace $10$ by $n$!
Invariants from Restricted Proofs

✔️ The language restriction forces the solver to generalize!

▷ Algorithm:
  
  ▶ If the proof fails, increase $L$!
  
  ▶ If we fail to get a sufficiently strong invariant, increase $n$.

✔️ This does work if we replace 10 by $n$!

❓ Which $L_1, L_2, \ldots$ is complete for which programs?