Model Checking (so far)

The promise of model checking

- Exhaustive exploration of the state space of a program
- Push-button verification of arbitrary temporal logic formulas
- Dramatic performance improvements from state-space reduction techniques (e.g. partial order reduction)

But

- It only works for programs with finite state space
Abstraction to the Rescue

- We can abstract the infinite state space into a finite one
- Every abstract state corresponds to an infinite set of states
- Is this the same thing as abstract interpretation?

```c
void main(){
    int x = *;(l_1)
    while(*){
        (l_2)
        if(x>0)
            x = 2*x;
        else
            (l_3)
            x = x-1;
        (l_5)
        x = abs(*)/x;
    }
}
```
Abstraction to the Rescue

- Abstractions usually have to be tailored to the program and property of interest
- Imprecision on the abstraction can lead to spurious paths

```c
void main()
{
    int x = *(l1);
    while(*)
    {
        if(x>1)
            x = 2*x;
        else
            x = x − 2;
        x = abs(*)/x;
    }
}
```
Abstractions usually have to be tailored to the program and property of interest.

Imprecision on the abstraction can lead to spurious paths.

```c
void main(){
    int x = *; (l_1)
    while(*){
        (l_2)
        if(x>1)
            (l_3)
            x = 2*x;
        else
            (l_4)
            x = x-2;
        (l_5)
        x = abs(*)/x;
    } (l_2,+) (l_2,0) (l_2,-) (l_5,+) (l_5,-) (l_5,0) (x,x)
}```
Abstraction Refinement

- We need a simple way to come up with abstractions
- Our abstractions must be flexible
- We need to be able to refine them on demand
- This is how we identify spurious paths and eliminate them
 Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data:
  \[(p_1(s), \cdots, p_n(s))\]
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int \(x = *;\)
if \((x \geq 0)\) then
  \(x++;\)
else \(x = -x\)
assert \((x \neq -1);\)

state:  \((x: \text{Int})\)
Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: 
  \( (p_1(s), \ldots, p_n(s)) \)
- Transition function can be computed by a theorem prover
- **Big idea**: We can refine the abstraction by introducing more predicates!

```plaintext
int x = *;
if (x ≥ 0) then
  x++;
else x = −x
assert(x ≠ -1);
```

State: \((x: \text{Int})\)

\( p_1 \equiv (x ≥ 0) \)
\( p_2 \equiv (x < 0) \)
Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: 
  \[(p_1(s), \cdots, p_n(s))\]
- Transition function can be computed by a theorem prover
- **Big idea**: We can refine the abstraction by introducing more predicates!

```plaintext
int x = *;
if (x \geq 0) then
  x++;
else x = -x
assert(x \neq -1);
```

```
state: (x: Int)
```

```
\begin{align*}
  x' &= * \\
  p_1 &\equiv (x \geq 0) \\
  p_2 &\equiv (x < 0)
\end{align*}
```

```
\begin{align*}
  x' &= x + 1 \\
  x' &= -x \\
  x &= -1
\end{align*}
```

```
q_0, x
q_1, x
q_2, x
q_3, x
q_4, x
```

```
q_0, \emptyset
q_1, \emptyset
```

```
state: (p_1: Bool, p_2: Bool)
p_1(x) \equiv (x \geq 0), p_2(x) \equiv (x < 0)
```
Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

```plaintext
int x = *;
if (x ≥ 0) then
    x++;
else x = -x
assert(x ≠ -1);
```

```
\[
\begin{align*}
q_0, x & \quad x' = * \\
q_1, x & \quad x ≥ 0 & x < 0 \\
q_2, x & \quad x' = x + 1 \\
q_3, x & \quad x' = -x \\
q_4, x & \quad x = -1 \\
\text{err} & \quad
\end{align*}
\]
```

```plaintext
state: (x: Int)
```

```
\[
\begin{align*}
p_1(x) & \equiv (x ≥ 0) \\
p_2(x) & \equiv (x < 0)
\end{align*}
\]
```

```plaintext
state: (p_1: Bool, p_2: Bool)
```

```
\[
\begin{align*}
q_0, \emptyset & \quad x' = * \\
q_1, \emptyset & \quad x < 0 \\
q_3, p_2 & \quad
\end{align*}
\]
Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data:
  \[(p_1(s), \cdots, p_n(s))\]
- Transition function can be computed by a theorem prover
- **Big idea**: We can refine the abstraction by introducing more predicates!

```plaintext
int x = *;
if (x ≥ 0) then
  x++;
else x = −x
assert(x ≠ -1);
```

**state:** \((x: \text{Int})\)

**state:** \((p_1: \text{Bool}, p_2: \text{Bool})\)

\[ p_1(x) \equiv (x \geq 0), p_2(x) \equiv (x < 0) \]
Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: 
  \[(p_1(s), \ldots, p_n(s))\]
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

\[
\begin{align*}
\text{int } x &= *; \\
\text{if } (x \geq 0) \text{ then} \\
& \quad x++; \\
\text{else } x &= -x \\
\text{assert}(x \neq -1);
\end{align*}
\]

\[
\begin{align*}
\text{state: } (x: \text{Int}) \\
p_1(x) &\equiv (x \geq 0), p_2(x) \equiv (x < 0)
\end{align*}
\]
Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: 
  \((p_1(s), \cdots, p_n(s))\)
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

```python
int x = *;
if (x \geq 0) then
  x++;
else x = -x
assert(x \neq -1);
```

```
state: (x: Int)  state: (p_1: Bool, p_2: Bool)
p_1(x) \equiv (x \geq 0), p_2(x) \equiv (x < 0)
```
• Safety verification: Prove no path from initial to final state.
• Problem: Huge (infinite) state graph
- Safety verification: Prove no path from initial to final state.
- Problem: Huge (infinite) state graph
Predicate Abstraction: Merge states satisfying same predicates to one abstract state
Predicate Abstraction: Merge states satisfying same predicates to one abstract state
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.

Spurious error path: introduced due to coarse abstraction
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.

Abstraction Refinement:
Add more predicates to make abstraction at a finer granularity.
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.
Continue searching in the new abstract state space. CounterExample-Guided Abstraction Refinement (CEGAR)
CounterExample-Guided Abstraction Refinement

Compute approximation of system w.r.t. set of predicates

Abstract model has a path to error?

No

Program Correct

Yes

Refine abstraction

Path is spurious?

Yes

No

Has Error

CEGAR
CounterExample-Guided Abstraction Refinement

Compute approximation of system w.r.t. set of predicates

Abstract model has a path to error?

No

Program Correct

Yes

Refine abstraction

Path is spurious?

Yes

Has Error

No
CounterExample-Guided Abstraction Refinement

Abstraction refinement: Craig interpolation
Craig interpolant for an inconsistent pair of formulae \((F, G)\) is a formula \(I\) s.t.

1. \(F \rightarrow I\),
2. \(I \land G\) is unsatisfiable,
3. \(I\) refers only to the common variables of \(F\) and \(G\).

Interpolant summarizes the reason two formulae are inconsistent in their shared language.
Craig Interpolation

Craig interpolant for an inconsistent pair of formulae \((F, G)\) is a formula \(I\) s.t.

1. \(F \rightarrow I\),
2. \(I \land G\) is unsatisfiable,
3. \(I\) refers only to the common variables of \(F\) and \(G\).

Interpolant summarizes the reason two formulae are inconsistent in their shared language

\[
(A \land B, \neg B)
\]

\[A, B \in B\]
Craig interpolant for an inconsistent pair of formulae \((F, G)\) is a formula \(I\) s.t.

1. \(F \rightarrow I\),
2. \(I \land G\) is unsatisfiable,
3. \(I\) refers only to the common variables of \(F\) and \(G\).

Interpolant summarizes the reason two formulae are inconsistent in their shared language.

\[
(A \land B \quad , \quad \neg B)
\]

\(A, B \in \mathbb{B}\)
Craig interpolant for an inconsistent pair of formulae $(F, G)$ is a formula $I$ s.t.

1. $F \rightarrow I$,  
2. $I \land G$ is unsatisfiable,  
3. $I$ refers only to the common variables of $F$ and $G$.

Interpolant summarizes the reason two formulae are inconsistent in their shared language.

$$ (A \land B, \neg B) $$

$$ I = B $$

Set of States

$A, B \in \mathbb{B}$
Craig Interpolation

Craig interpolant for an inconsistent pair of formulae \((F, G)\) is a formula \(I\) s.t.

1. \(F \rightarrow I\),
2. \(I \land G\) is unsatisfiable,
3. \(I\) refers only to the common variables of \(F\) and \(G\).

Interpolant summarizes the reason two formulae are inconsistent in their shared language.

\[
\begin{align*}
\text{Set of States} & : F \quad I \quad G \\
\text{Craig interpolant} & : I = B \\
\text{common variables} & : (A \land B, \neg B) \\
A, B \in \mathbb{B}
\end{align*}
\]
Craig interpolant for an inconsistent pair of formulae \((F, G)\) is a formula \(I\) s.t.

1. \(F \rightarrow I\),
2. \(I \land G\) is unsatisfiable,
3. \(I\) refers only to the common variables of \(F\) and \(G\).

Interpolant summarizes the reason two formulae are inconsistent in their shared language.

- Craig’s Theorem [1957]:
  - First-order logic has the interpolation property
  - If \(F \land G\) is unsatisfiable, then a Craig interpolant exists
- For certain theories (like linear arithmetic) interpolant can be derived from a refutation of \(F \land G\) in polynomial time
Craig Interpolation

Craig interpolant for an inconsistent pair of formulae \((F, G)\) is a formula \(I\) s.t.

1. \(F \rightarrow I\),
2. \(I \land G\) is unsatisfiable,
3. \(I\) refers only to the common variables of \(F\) and \(G\).

Interpolant summarizes the reason two formulae are inconsistent in their shared language.

Let \(\Gamma = \{F_1, F_2, \cdots, F_n\}\) be a set of formulae such that \(\bigwedge \Gamma \equiv false\)

An Inductive Interpolant Sequence for \(\Gamma\) is a set \(\{I_0, I_1, \cdots, I_n\}\) such that:

1. \(I_0 = true\) and \(I_n = false\)
2. For every \(0 \leq j < n\) : \(I_j \land F_{j+1} \rightarrow I_{j+1}\)
3. For every \(0 < j < n\) : \(\mathcal{L}(I_j) \subseteq \mathcal{L}(F_1, \cdots, F_j) \cap \mathcal{L}(F_{j+1}, \cdots, F_n)\)
CEGAR: Example

\[
\begin{align*}
x & = -1 \\
x & = y \\
x & \neq y \\
x' & = x - y \\
x' & = y - x \\
\end{align*}
\]

\[
\begin{align*}
x & \geq 0 \land y \geq 0 \\
x & \leq y \\
x & = y \\
x & = -1 \\
x' & = x - y \\
y' & = y - x \\
\end{align*}
\]
CEGAR: Example

\[ x \geq 0 \land y \geq 0 \]

\[ x = -1 \]

\[ x = y \]

\[ x \neq y \]

\[ x' = x - y \]

\[ y' = y - x \]

\[ x > y \]

\[ x \leq y \]

\[ q_0, \emptyset \]

\[ x \geq 0 \land y \geq 0 \]

\[ q_1, \emptyset \]

\[ x = y \]

\[ q_5, \emptyset \]

\[ x = -1 \]

\[ err \]
CEGAR: Example

\[ (x \geq 0 \land y \geq 0) \land (x = y) \land (x = -1) = false \]

Interpolant: \{x \geq 0, x \geq 0\}
CEGAR: Example

\( (x \geq 0 \land y \geq 0) \land (x = y) \land (x = -1) = false \)

- Interpolant: \( \{x \geq 0, x \geq 0\} \)
CEGAR: Example

\[
\begin{align*}
&x \geq 0 \land y \geq 0, \\
&x = y, \\
&x \neq y, \\
&x' = x - y, \\
&y' = y - x
\end{align*}
\]

Infeasible suffix

\[
\begin{align*}
x = -1 &
\end{align*}
\]
CEGAR: Example

- \( (x \leq y) \land (y_1 = y - x) \land (x = y_1) \land (x = -1) \) = false
- Interpolant: \( \{x \leq y, y_1 \geq 0, x \geq 0\} \)
Abstract Reachability Graph:
Unfolding the program in the abstract space