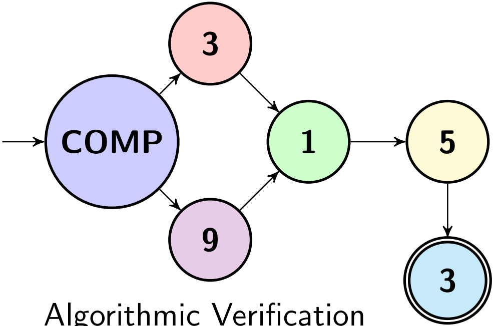


Algorithmic Verification

CEGAR and Predicate Abstraction

Dr. Liam O'Connor
CSE, UNSW (for now)
Term 1 2020

Model Checking with Abstractions

Abstractions typically have a smaller state space, so it is advantageous to try to model check with abstractions rather than a concrete model.

Model Checking with Abstractions

Abstractions typically have a smaller state space, so it is advantageous to try to model check with abstractions rather than a concrete model.

We need:

- To know that properties that hold for our abstractions hold for our model — true for all $\varphi \in \mathsf{ACTL}$.

Model Checking with Abstractions

Abstractions typically have a smaller state space, so it is advantageous to try to model check with abstractions rather than a concrete model.

We need:

- To know that properties that hold for our abstractions hold for our model — true for all $\varphi \in \mathrm{ACTL}$.
- To know that when our properties don't hold for our abstractions, they don't hold for our model — not true in general!

Model Checking with Abstractions

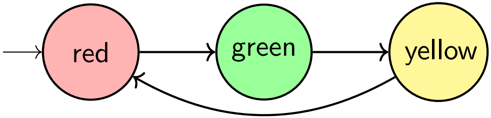
Abstractions typically have a smaller state space, so it is advantageous to try to model check with abstractions rather than a concrete model.

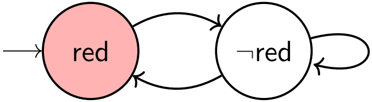
We need:

- To know that properties that hold for our abstractions hold for our model — true for all $\varphi \in \text{ACTL}$.
- To know that when our properties don't hold for our abstractions, they don't hold for our model — not true in general!

We need to pick the abstraction **based on** the properties we care about, and if necessary change our abstraction on the fly based on the results we see.

Model Checking with Abstractions

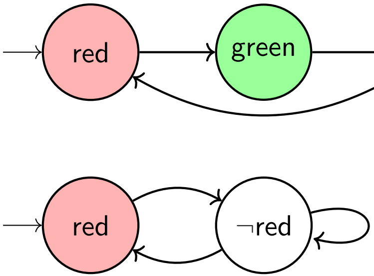




Consider the following ACTL formulae:

- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \neg\text{red})$

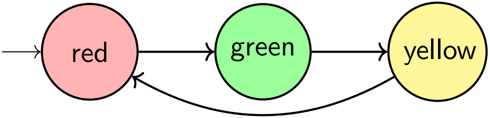
Model Checking with Abstractions

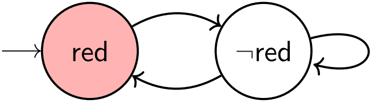


Consider the following ACTL formulae:

- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \neg\text{red})$
- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \mathbf{AX}\ \text{red})$

Model Checking with Abstractions

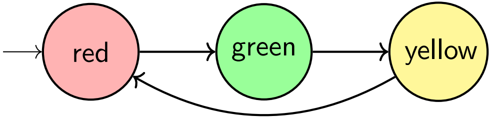


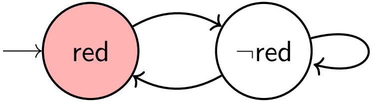


Consider the following ACTL formulae:

- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \neg\text{red})$
- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \mathbf{AX}\ \text{red})$
- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \mathbf{AX}\ \mathbf{AX}\ \text{red})$

Model Checking with Abstractions





Consider the following ACTL formulae:

- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \neg\text{red})$
- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \mathbf{AX}\ \text{red})$
- $\mathbf{AG}\ (\text{red} \Rightarrow \mathbf{AX}\ \mathbf{AX}\ \mathbf{AX}\ \text{red})$

We know that if $A \sqsubseteq C$ then $(A \models \varphi) \Rightarrow (C \models \varphi)$ for $\varphi \in \text{ACTL}$, but what about if $A \not\models \varphi$?

Counterexamples

Note

If $A \not\models \varphi$ for some $\varphi \in \mathrm{ACTL}$, then there exists a run that serves as a *counterexample* to the formula φ.

- If $A \not\models \varphi$, that tells us either that $C \not\models \varphi$ or that our abstraction is not precise enough — the counterexample will be *spurious*.

Counterexamples

Note

If $A \not\models \varphi$ for some $\varphi \in$ ACTL, then there exists a run that serves as a *counterexample* to the formula φ.

- If $A \not\models \varphi$, that tells us either that $C \not\models \varphi$ or that our abstraction is not precise enough — the counterexample will be *spurious*.
- **Our approach:** To check if our counterexample is spurious, convert it to a concrete run $\in$ C.

Abstract to Concrete Run

Let α be our abstraction mapping $Q_C \rightarrow Q_A$ and our run be $q_0 q_1 q_2 \ldots$. We apply the mapping in reverse, α^{-1}, and try to find a concrete run starting from our initial state I_C according to transition relation δ_C:

$$\begin{aligned} S_0 &= I_C \cap \alpha^{-1}(q_0) \\ S_1 &= \delta_C(S_0) \cap \alpha^{-1}(q_1) \\ S_2 &= \delta_C(S_1) \cap \alpha^{-1}(q_2) \quad \text{etc..} \end{aligned}$$

Abstract to Concrete Run

Let α be our abstraction mapping $Q_C \to Q_A$ and our run be $q_0 q_1 q_2 \ldots$. We apply the mapping in reverse, α^{-1}, and try to find a concrete run starting from our initial state I_C according to transition relation δ_C:

$$\begin{aligned} S_0 &= I_C \cap \alpha^{-1}(q_0) \\ S_1 &= \delta_C(S_0) \cap \alpha^{-1}(q_1) \\ S_2 &= \delta_C(S_1) \cap \alpha^{-1}(q_2) \quad \text{etc..} \end{aligned}$$

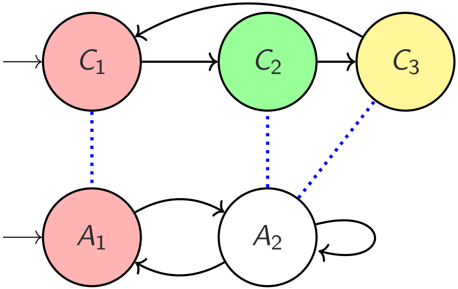
If there is such a run (i.e. no $S_i = \emptyset$), the run is not spurious.

Abstract to Concrete Run

Let α be our abstraction mapping $Q_C \to Q_A$ and our run be $q_0 q_1 q_2 \ldots$. We apply the mapping in reverse, α^{-1}, and try to find a concrete run starting from our initial state I_C according to transition relation δ_C:

$$\begin{aligned} S_0 &= I_C \cap \alpha^{-1}(q_0) \\ S_1 &= \delta_C(S_0) \cap \alpha^{-1}(q_1) \\ S_2 &= \delta_C(S_1) \cap \alpha^{-1}(q_2) \quad \text{etc..} \end{aligned}$$

If there is such a run (i.e. no $S_i = \emptyset$), the run is not spurious.



Example

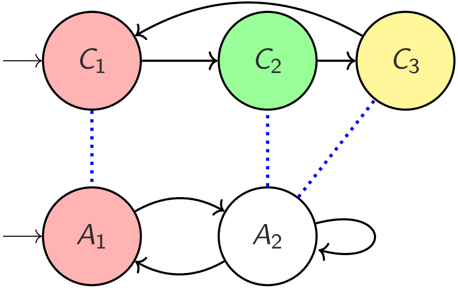
AG (red ⇒ **AX AX** red)

Abstract to Concrete Run

Let α be our abstraction mapping $Q_C \to Q_A$ and our run be $q_0 q_1 q_2 \ldots$. We apply the mapping in reverse, α^{-1}, and try to find a concrete run starting from our initial state I_C according to transition relation δ_C:

$$\begin{aligned} S_0 &= I_C \cap \alpha^{-1}(q_0) \\ S_1 &= \delta_C(S_0) \cap \alpha^{-1}(q_1) \\ S_2 &= \delta_C(S_1) \cap \alpha^{-1}(q_2) \quad \text{etc..} \end{aligned}$$

If there is such a run (i.e. no $S_i = \emptyset$), the run is not spurious.



Example

AG (red $\Rightarrow$ **AX AX** red)

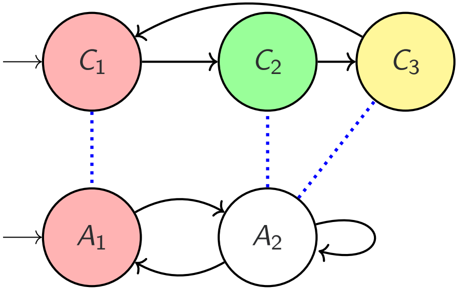
Counterexample: $A_1 A_2 A_2$

Abstract to Concrete Run

Let α be our abstraction mapping $Q_C \rightarrow Q_A$ and our run be $q_0 q_1 q_2 \ldots$. We apply the mapping in reverse, α^{-1}, and try to find a concrete run starting from our initial state I_C according to transition relation δ_C:

$$\begin{aligned} S_0 &= I_C \cap \alpha^{-1}(q_0) \\ S_1 &= \delta_C(S_0) \cap \alpha^{-1}(q_1) \\ S_2 &= \delta_C(S_1) \cap \alpha^{-1}(q_2) \quad \text{etc..} \end{aligned}$$

If there is such a run (i.e. no $S_i = \emptyset$), the run is not spurious.



Example

AG (red $\Rightarrow$ **AX AX** red)

Counterexample: $A_1 A_2 A_2$

$\alpha^{-1}(A_1 A_2 A_2)$

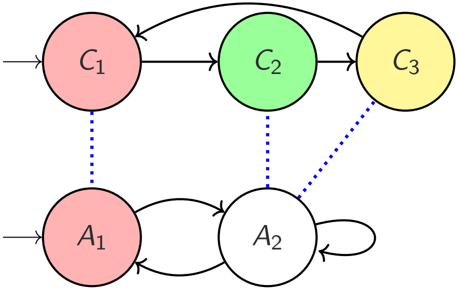
$= \{C_1\}\{C_2, C_3\}\{C_2, C_3\}$

Abstract to Concrete Run

Let α be our abstraction mapping $Q_C \to Q_A$ and our run be $q_0 q_1 q_2 \ldots$. We apply the mapping in reverse, α^{-1}, and try to find a concrete run starting from our initial state I_C according to transition relation δ_C:

$$\begin{aligned} S_0 &= I_C \cap \alpha^{-1}(q_0) \\ S_1 &= \delta_C(S_0) \cap \alpha^{-1}(q_1) \\ S_2 &= \delta_C(S_1) \cap \alpha^{-1}(q_2) \quad \text{etc..} \end{aligned}$$

If there is such a run (i.e. no $S_i = \emptyset$), the run is not spurious.



Example

AG (red $\Rightarrow$ **AX AX** red)

Counterexample: $A_1 A_2 A_2$

$\alpha^{-1}(A_1 A_2 A_2)$

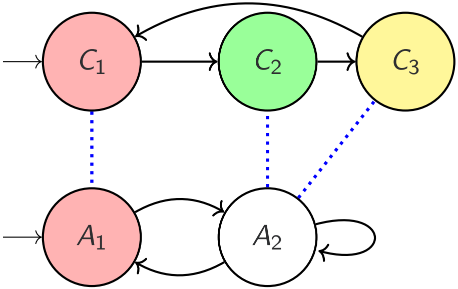
$= \{C_1\}\{C_2, C_3\}\{C_2, C_3\}$

There is a run

$C_1 \xrightarrow{\delta_C} C_2 \xrightarrow{\delta_C} C_3$

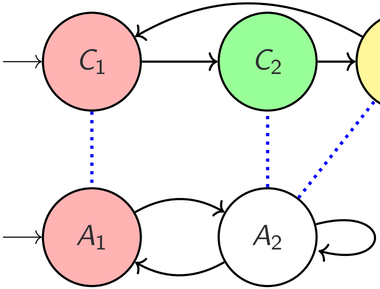
$\therefore$ **Not spurious**.

Spurious Counterexamples



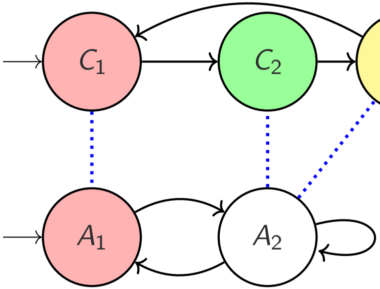
AG (red $\Rightarrow$ **AX AX AX** red)

Spurious Counterexamples



AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

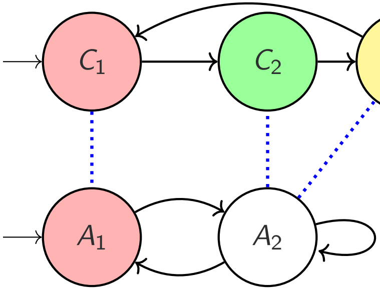
Spurious Counterexamples



AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$S_0 \quad = \quad I_C \cap \alpha^{-1}(A_1) \qquad =$

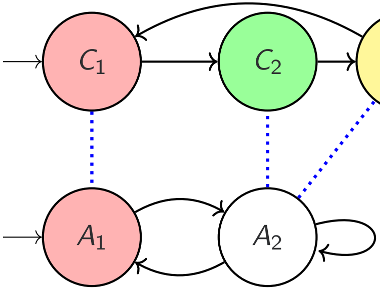
Spurious Counterexamples



AG (red $\Rightarrow$ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$S_0 \;=\; I_C \cap \alpha^{-1}(A_1) \;=\; \{C_1\} \cap \{C_1\} \;=$

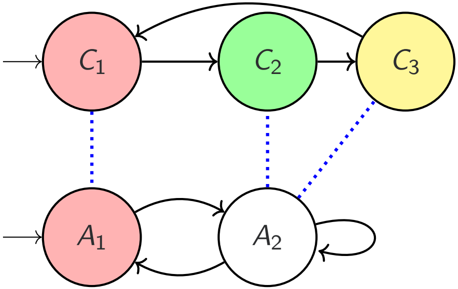
Spurious Counterexamples



AG (red $\Rightarrow$ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$
\begin{array}{lllll}
S_0 & = & I_C \cap \alpha^{-1}(A_1) & = & \{C_1\} \cap \{C_1\} & = & \{C_1\} \\
S_1 & = & \delta_C(S_0) \cap \alpha^{-1}(A_2) & & & &
\end{array}
$$

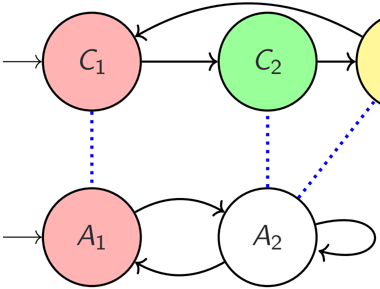
Spurious Counterexamples



AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$\begin{array}{lllllll} S_0 & = & I_C \cap \alpha^{-1}(A_1) & = & \{C_1\} \cap \{C_1\} & = & \{C_1\} \\ S_1 & = & \delta_C(S_0) \cap \alpha^{-1}(A_2) & = & \{C_2\} \cap \{C_2, C_3\} & = & \{C_2\} \\ S_2 & = & \delta_C(S_1) \cap \alpha^{-1}(A_2) & & & & \end{array}$$

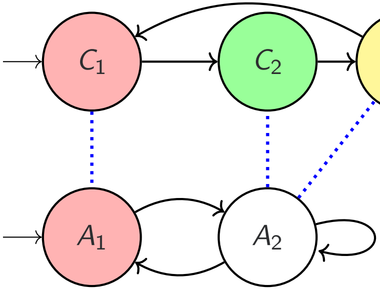
Spurious Counterexamples



AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$
\begin{array}{lllllll}
S_0 & = & I_C \cap \alpha^{-1}(A_1) & = & \{C_1\} \cap \{C_1\} & = & \{C_1\} \\
S_1 & = & \delta_C(S_0) \cap \alpha^{-1}(A_2) & = & \{C_2\} \cap \{C_2, C_3\} & = & \{C_2\} \\
S_2 & = & \delta_C(S_1) \cap \alpha^{-1}(A_2) & = & \{C_3\} \cap \{C_2, C_3\} & = & \{C_3\} \\
S_3 & = & \delta_C(S_2) \cap \alpha^{-1}(A_2) & & & &
\end{array}
$$

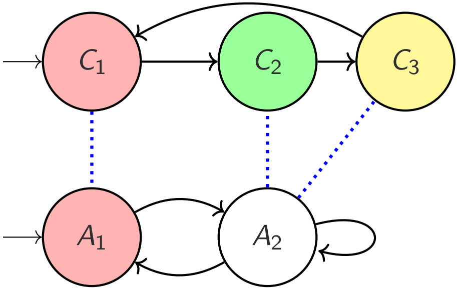
Spurious Counterexamples



AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$
\begin{array}{lclcl}
S_0 & = & I_C \cap \alpha^{-1}(A_1) & = & \{C_1\} \cap \{C_1\} & = & \{C_1\} \\
S_1 & = & \delta_C(S_0) \cap \alpha^{-1}(A_2) & = & \{C_2\} \cap \{C_2, C_3\} & = & \{C_2\} \\
S_2 & = & \delta_C(S_1) \cap \alpha^{-1}(A_2) & = & \{C_3\} \cap \{C_2, C_3\} & = & \{C_3\} \\
S_3 & = & \delta_C(S_2) \cap \alpha^{-1}(A_2) & = & \{C_1\} \cap \{C_2, C_3\} & = &
\end{array}
$$

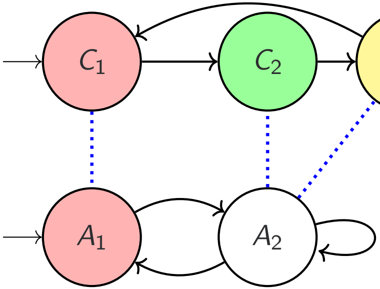
Spurious Counterexamples



AG (red $\Rightarrow$ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$\begin{array}{lllll} S_0 & = & I_C \cap \alpha^{-1}(A_1) & = & \{C_1\} \cap \{C_1\} & = & \{C_1\} \\ S_1 & = & \delta_C(S_0) \cap \alpha^{-1}(A_2) & = & \{C_2\} \cap \{C_2, C_3\} & = & \{C_2\} \\ S_2 & = & \delta_C(S_1) \cap \alpha^{-1}(A_2) & = & \{C_3\} \cap \{C_2, C_3\} & = & \{C_3\} \\ S_3 & = & \delta_C(S_2) \cap \alpha^{-1}(A_2) & = & \{C_1\} \cap \{C_2, C_3\} & = & \emptyset \end{array}$$

Spurious Counterexamples



AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$\begin{aligned}
S_0 &= I_C \cap \alpha^{-1}(A_1) &&= \{C_1\} \cap \{C_1\} &&= \{C_1\} \\
S_1 &= \delta_C(S_0) \cap \alpha^{-1}(A_2) &&= \{C_2\} \cap \{C_2, C_3\} &&= \{C_2\} \\
S_2 &= \delta_C(S_1) \cap \alpha^{-1}(A_2) &&= \{C_3\} \cap \{C_2, C_3\} &&= \{C_3\} \\
S_3 &= \delta_C(S_2) \cap \alpha^{-1}(A_2) &&= \{C_1\} \cap \{C_2, C_3\} &&= \emptyset
\end{aligned}$$

There is no concrete run — this counterexample is spurious. Our abstraction is too imprecise.

Abstraction Refinement

Definition

An abstraction mapping α generates an equivalence relation on states $\equiv_\alpha$ where $q \equiv_\alpha q' \Leftrightarrow \alpha(q) = \alpha(q')$.

Abstraction Refinement

Definition

An abstraction mapping α generates an equivalence relation on states $\equiv_\alpha$ where $q \equiv_\alpha q' \Leftrightarrow \alpha(q) = \alpha(q')$.

Consider two abstractions $\alpha : Q_C \rightarrow Q_A$ and $\alpha' : Q_C \rightarrow Q_B$.
We say that α' refines α iff $\equiv_{\alpha'} \subseteq \equiv_\alpha$.
Similarly, we say α' strictly refines α iff $\equiv_{\alpha'} \subsetneq \equiv_\alpha$

Informal Notion

We previously considered abstractions as grouping together concrete states into equivalence classes. We can refine abstractions by splitting those equivalence classes.

Abstraction Refinement

We have a spurious counterexample $q_1 q_2 q_3 \ldots$.
Which classes should we split up in our new abstraction?

Abstraction Refinement

We have a spurious counterexample $q_1 q_2 q_3 \ldots$.
Which classes should we split up in our new abstraction?

Counterexample Guidance

For each q_i in our counterexample, the class of concrete states it is abstracting is $\alpha^{-1}(q_i)$.

Abstraction Refinement

We have a spurious counterexample $q_1 q_2 q_3 \dots$.
Which classes should we split up in our new abstraction?

Counterexample Guidance

For each q_i in our counterexample, the class of concrete states it is abstracting is $\alpha^{-1}(q_i)$.
We will split this class into two sets:

1. Those that follow directly from the previous state: $\alpha^{-1}(q_i) \cap \delta_C(S_{i-1})$

Abstraction Refinement

We have a spurious counterexample $q_1 q_2 q_3 \ldots$.
Which classes should we split up in our new abstraction?

Counterexample Guidance

For each q_i in our counterexample, the class of concrete states it is abstracting is $\alpha^{-1}(q_i)$.
We will split this class into two sets:

1. Those that follow directly from the previous state: $\alpha^{-1}(q_i) \cap \delta_C(S_{i-1})$
2. Those that don't: $\alpha^{-1}(q_i) \setminus \delta_C(S_{i-1})$

Abstraction Refinement

We have a spurious counterexample $q_1 q_2 q_3 \ldots$.
Which classes should we split up in our new abstraction?

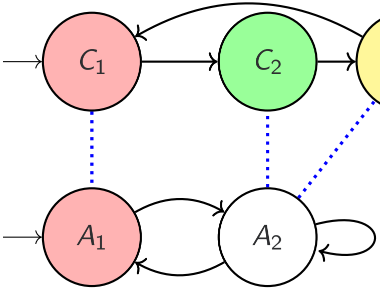
Counterexample Guidance

For each q_i in our counterexample, the class of concrete states it is abstracting is $\alpha^{-1}(q_i)$.
We will split this class into two sets:

1. Those that follow directly from the previous state: $\alpha^{-1}(q_i) \cap \delta_C(S_{i-1})$
2. Those that don't: $\alpha^{-1}(q_i) \setminus \delta_C(S_{i-1})$

The resulting classes will form the new, refined abstraction of our model. If both of these sets are non-empty, we split the state q_i into two states, one for each set.

Example



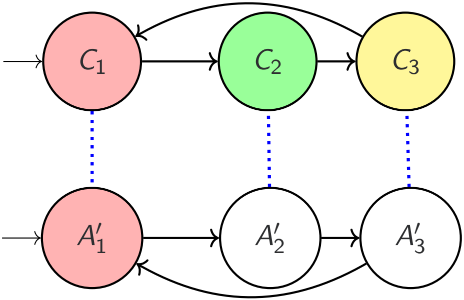
AG (red ⇒ **AX AX AX** red) **Counterexample:** $A_1 A_2 A_2 A_2$

$$
\begin{array}{lllllll}
S_0 & = & I_C \cap \alpha^{-1}(A_1) & = & \{C_1\} \cap \{C_1\} & = & \{C_1\} \\
S_1 & = & \delta_C(S_0) \cap \alpha^{-1}(A_2) & = & \{C_2\} \cap \{C_2, C_3\} & = & \{C_2\} \\
S_2 & = & \delta_C(S_1) \cap \alpha^{-1}(A_2) & = & \{C_3\} \cap \{C_2, C_3\} & = & \{C_3\} \\
S_3 & = & \delta_C(S_2) \cap \alpha^{-1}(A_2) & = & \{C_1\} \cap \{C_2, C_3\} & = & \emptyset
\end{array}
$$

$\alpha^{-1}(A_2) = \{C_2, C_3\}$. We have to split this into those that follow from S_0 ($\{C_2\}$) and those that don't ($\{C_3\}$).

After Splitting

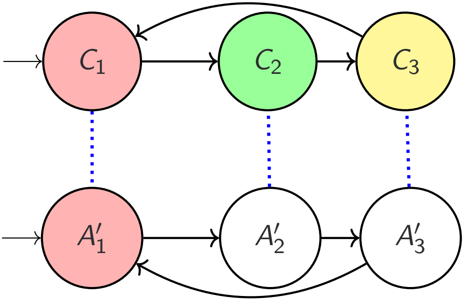
We split A_2 into A'_2 and A'_3



We now have an abstraction that does not exhibit our spurious counterexample, but the state space has increased.

After Splitting

We split A_2 into A_2' and A_3'

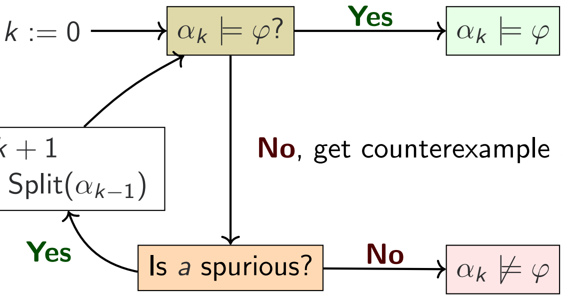


We now have an abstraction that does not exhibit our spurious counterexample, but the state space has increased.

In fact, it's impossible to refine this further, why?

CEGAR

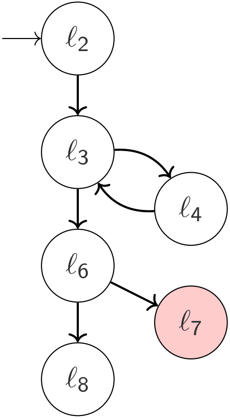
This technique gives us an approach called Counterexample Guided Abstraction Refinement (CEGAR). We have a starting abstraction α_0 and an ACTL formula φ:



C Programs

Objective: Prove that our assertion is never violated.

```c
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```



Need to check reachability, but can we simplify the state space first?

Predicate Abstraction

Predicate Abstraction

A *predicate abstraction* of a program is a version of the program with the same control flow graph, where all variables are replaced with boolean overapproximations.
Booleans can be `true`, `false`, or $*$ (nondeterministically true or false).

Basic PA

To start with, let's try using `i < n` as our only predicate:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

we want our boolean program to be an abstraction.

Requirement

If a location is not reachable in the abstraction, it is not reachable in the concrete program.

Basic PA

To start with, let's try using `i < n` as our only predicate:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

```
int main() {
    int b = false;
    while (b) {
        b = b?*:false;
    }
    if (b)
        assert(false);
}
```

Basic PA

To start with, let's try using `i < n` as our only predicate:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

```
int main() {
    int b = false;
    while (b) {
        b = b?*:false;
    }
    if (b)
        assert(false);
}
```

we want our boolean program to be an abstraction.

Requirement

If a location is not reachable in the abstraction, it is not reachable in the concrete program.

Harder PA

Now let's try using `i < 2` and `n >= 3` as our only predicates:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

What do we use for the ?? It must overapproximate `i < n`.

Harder PA

Now let's try using `i < 2` and `n >= 3` as our only predicates:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

```
int main() {
    int b1 = true, b2 = false;
    while (??) {
        b1 = b1?*:false;
    }
    if (??)
        assert(false);
}
```

Harder PA

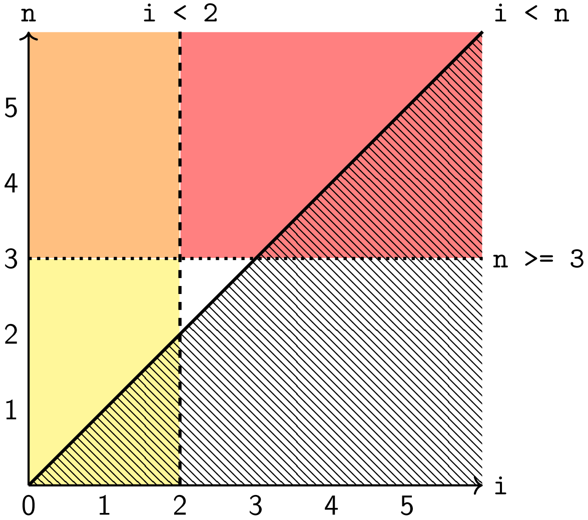
Now let's try using `i < 2` and `n >= 3` as our only predicates:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

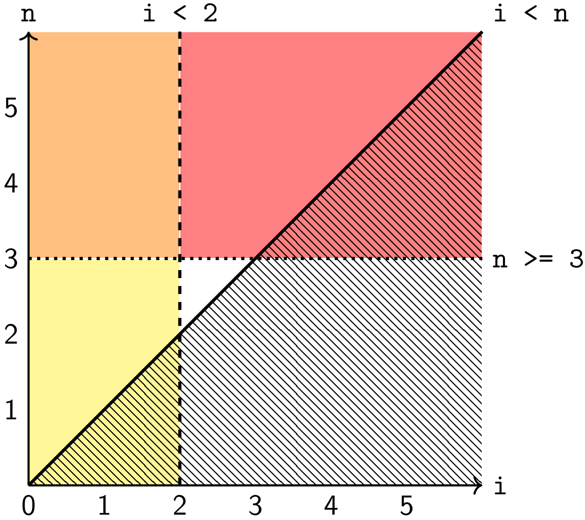
```
int main() {
    int b1 = true, b2 = false;
    while (??) {
        b1 = b1?*:false;
    }
    if (??)
        assert(false);
}
```

What do we use for the ?? It must overapproximate `i < n`.

Abstract Condition



Abstract Condition



The only overapproximation is $\neg(i < 2 \wedge n \geq 3)$ i.e. `!(b1 && b2)`

Harder PA

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

```
int main() {
    int b1 = true, b2 = false;
    while (!(b1 && b2)){
        b1 = b1?*:false;
    }
    if (!(b1 && b2))
        assert(false);
}
```

No Predicates

The abstraction with no predicates has all states reachable:

```
int main() {
    int i = 0, n = 0;
    while (i < n) {
        i++;
    }
    if (i < n)
        assert(false);
}
```

```
int main() {
    ;;
    while (*){
        ;;
    }
    if (*)
        assert(false);
}
```

How do we find out what predicates to add?

No Predicates

The abstraction with no predicates has all states reachable:

```c
int main() {
   int i = 0, n = 0;
   while (i < n) {
      i++;
   }
   if (i < n)
      assert(false);
}
```

```c
int main() {
   ;;
   while (*){
      ;;
   }
   if (*)
      assert(false);
}
```

How do we find out what predicates to add? **Use CEGAR!**

No Predicates

The abstraction with no predicates has all states reachable:

```
1 int main() {
2    int i = 0, n = 0;
3    while (i < n) {
4       i++;
5    }
6    if (i < n)
7       assert(false);
8 }
```

```
1 int main() {
2    ;;
3    while (*){
4       ;;
5    }
6    if (*)
7       assert(false);
8 }
```

How do we find out what predicates to add? **Use CEGAR!**

Example (Abstract Counterexample)

Lines $3 \rightarrow 6 \rightarrow 7$.

No Predicates

The abstraction with no predicates has all states reachable:

```
1 int main() {
2    int i = 0, n = 0;
3    while (i < n) {
4       i++;
5    }
6    if (i < n)
7       assert(false);
8 }
```

```
1 int main() {
2    ;;
3    while (*){
4       ;;
5    }
6    if (*)
7       assert(false);
8 }
```

How do we find out what predicates to add? **Use CEGAR!**

Example (Abstract Counterexample)

Lines $3 \to 6 \to 7$.Looking at the concrete program, this path would require `i >= n` (to move from line 3 to 6) and `i < n` (to move from line 6 to 7).

No Predicates

The abstraction with no predicates has all states reachable:

```
1 int main() {
2     int i = 0, n = 0;
3     while (i < n) {
4         i++;
5     }
6     if (i < n)
7         assert(false);
8 }
```

```
1 int main() {
2     ;;
3     while (*){
4         ;;
5     }
6     if (*)
7         assert(false);
8 }
```

How do we find out what predicates to add? **Use CEGAR!**

Example (Abstract Counterexample)

Lines 3 $\rightarrow$ 6 $\rightarrow$ 7.Looking at the concrete program, this path would require `i >= n` (to move from line 3 to 6) and `i < n` (to move from line 6 to 7).

Both can't be true simultaneously. This path is spurious.

Interpolants

Craig's Interpolation Theorem

If we have two predicates $P(x)$ and $Q(y)$ such are contradictory (i.e. $\neg(P(x) \wedge Q(y))$), then there exists a predicate $I(x \cap y)$ which:

- is implied by $P(x)$, i.e. $P(x) \Rightarrow I(x \cap y)$, and
- contradicts $Q(y)$ i.e. $\neg(I(x \cap y) \wedge Q(y))$.

Crucially, the interpolant $I(x \cap y)$ only ranges over variables common to both predicates.

Interpolants

Craig's Interpolation Theorem

If we have two predicates $P(x)$ and $Q(y)$ such are contradictory (i.e. $\neg(P(x) \wedge Q(y))$), then there exists a predicate $I(x \cap y)$ which:

- is implied by $P(x)$, i.e. $P(x) \Rightarrow I(x \cap y)$, and
- contradicts $Q(y)$ i.e. $\neg(I(x \cap y) \wedge Q(y))$.

Crucially, the interpolant $I(x \cap y)$ only ranges over variables common to both predicates.

Example

- $(i = 1)$ and $i \leq 0$:

Interpolants

Craig's Interpolation Theorem

If we have two predicates $P(x)$ and $Q(y)$ such are contradictory (i.e. $\neg(P(x) \wedge Q(y))$), then there exists a predicate $I(x \cap y)$ which:

- is implied by $P(x)$, i.e. $P(x) \Rightarrow I(x \cap y)$, and
- contradicts $Q(y)$ i.e. $\neg(I(x \cap y) \wedge Q(y))$.

Crucially, the interpolant $I(x \cap y)$ only ranges over variables common to both predicates.

Example

- $(i = 1)$ and $i \leq 0$:$i > 0$
- $(i \leq 2 \wedge k = i + 1)$ and $k > 5$:

Interpolants

Craig's Interpolation Theorem

If we have two predicates $P(x)$ and $Q(y)$ such are contradictory (i.e. $\neg(P(x) \wedge Q(y))$), then there exists a predicate $I(x \cap y)$ which:

- is implied by $P(x)$, i.e. $P(x) \Rightarrow I(x \cap y)$, and
- contradicts $Q(y)$ i.e. $\neg(I(x \cap y) \wedge Q(y))$.

Crucially, the interpolant $I(x \cap y)$ only ranges over variables common to both predicates.

Example

- $(i = 1)$ and $i \leq 0$:$i > 0$
- $(i \leq 2 \wedge k = i + 1)$ and $k > 5$: $k \leq 4$
- $(i \geq n)$ and $i < n$:

Interpolants

Craig's Interpolation Theorem

If we have two predicates $P(x)$ and $Q(y)$ such are contradictory (i.e. $\neg(P(x) \wedge Q(y))$), then there exists a predicate $I(x \cap y)$ which:

- is implied by $P(x)$, i.e. $P(x) \Rightarrow I(x \cap y)$, and
- contradicts $Q(y)$ i.e. $\neg(I(x \cap y) \wedge Q(y))$.

Crucially, the interpolant $I(x \cap y)$ only ranges over variables common to both predicates.

Example

- $(i = 1)$ and $i \leq 0$:$i > 0$
- $(i \leq 2 \wedge k = i + 1)$ and $k > 5$: $k \leq 4$
- $(i \geq n)$ and $i < n$:$i \geq n$

Path Interpolant

Sequence of program locations

$$\ell_1 \ell_2 \ell_3 \ldots \ell_k$$

Path Interpolant

Sequence of program locations

$$\ell_1\ell_2\ell_3\ldots\ell_k$$

$$\downarrow$$

Sequence of predicates

$$\pi_1\pi_2\pi_3\ldots\pi_k$$

Path Interpolant

Sequence of program locations

$$\ell_1 \ell_2 \ell_3 \ldots \ell_k$$

$$\downarrow$$

Sequence of predicates

$$\pi_1 \pi_2 \pi_3 \ldots \pi_k$$

$$\downarrow$$

$$\pi_1 \wedge \pi_2 \wedge \pi_3 \ldots \pi_k$$

Path Interpolant

Sequence of program locations

$\ell_1 \ell_2 \ell_3 \ldots \ell_k$

↓

Sequence of predicates

$\pi_1 \pi_2 \pi_3 \ldots \pi_k$

↓

$\pi_1 \wedge \pi_2 \wedge \pi_3 \ldots \pi_k$

SAT

Non-spurious case

Path Interpolant

Sequence of program locations

$\ell_1 \ell_2 \ell_3 \ldots \ell_k$

↓

Sequence of predicates

$\pi_1 \pi_2 \pi_3 \ldots \pi_k$

↓

$\pi_1 \wedge \pi_2 \wedge \pi_3 \ldots \pi_k$

SAT → Non-spurious case

UNSAT → Spurious case

Path Interpolant

Sequence of program locations

$\ell_1 \ell_2 \ell_3 \dots \ell_k$

↓

Sequence of predicates

$\pi_1 \pi_2 \pi_3 \dots \pi_k$

↓

$\pi_1 \wedge \pi_2 \wedge \pi_3 \dots \pi_k$

SAT → Non-spurious case

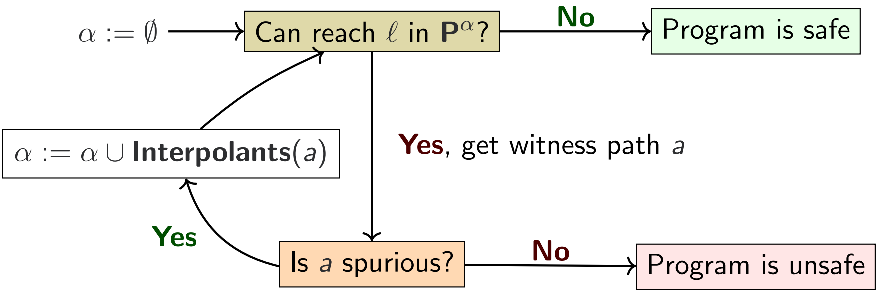
UNSAT → Spurious case

There exists interpolants

$I_1 I_2 I_3 \dots I_{k-1}$

CEGAR for C Programs

Let **P** be our program, α be our predicate set, and $\mathbf{P}^\alpha$ be the predicate abstraction of **P** using α. The location $\ell \in \mathbf{P}$ is our bad state we want to avoid (assertion failure).



Termination

On finite automata

- Finite number of states

Termination

On finite automata

- Finite number of states
- Each CEGAR loop increases the number of states in the abstraction, but the number can't exceed the number of concrete states.

Termination

On finite automata

- Finite number of states
- Each CEGAR loop increases the number of states in the abstraction, but the number can't exceed the number of concrete states.

On C programs

Termination

On finite automata

- Finite number of states
- Each CEGAR loop increases the number of states in the abstraction, but the number can't exceed the number of concrete states.

On C programs

- (Effectively) infinite amount of states

Termination

On finite automata

- Finite number of states
- Each CEGAR loop increases the number of states in the abstraction, but the number can't exceed the number of concrete states.

On C programs

- (Effectively) infinite amount of states
- ∴ No guarantee of termination
- When it terminates it is both sound (in that it always finds errors if they exist) and complete (it will not provide spurious errors).

Bibliography

CEGAR is used in SLAM/SDV (Microsoft), BLAST (Berkeley) and CBMC (Oxford).

- E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In Computer Aided Verification, pages 154-169, 2000
- Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Gregoire Sutre, Software Verification with BLAST. In SPIN Workshop 2003, LNCS 2648, pages 235-239.