IA169 Model Checking

Abstraction and CEGAR

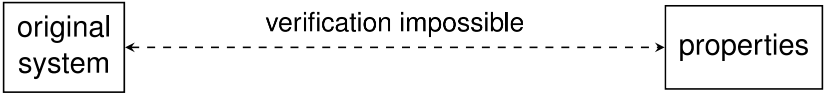
Jan Strejček

Faculty of Informatics
Masaryk University

Motivation

Abstraction is one of the most important techniques for reducing the state explosion problem.

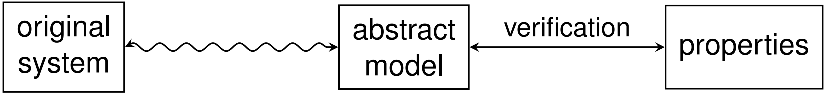
[CGKPV18]



Motivation

Abstraction is one of the most important techniques for reducing the state explosion problem.

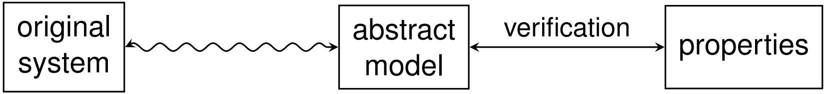
[CGKPV18]



Motivation

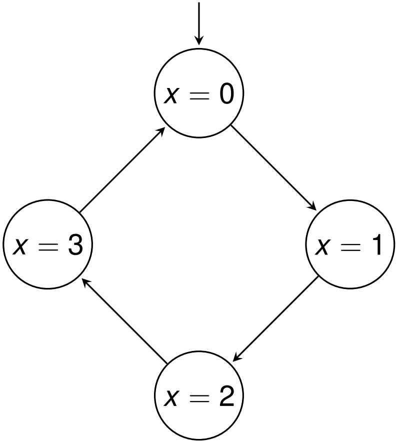
Abstraction is one of the most important techniques for reducing the state explosion problem.

[CGKPV18]

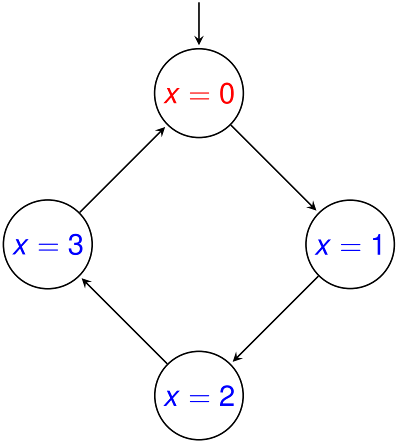


- large finite systems $\longrightarrow$ smaller finite systems
- infinite-state systems $\longrightarrow$ finite systems

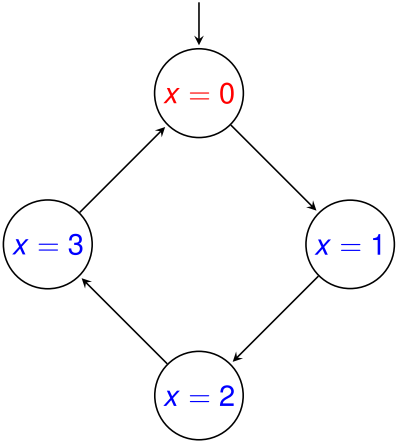
Intuition

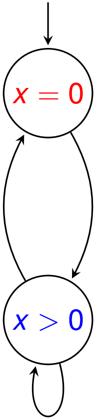


Intuition

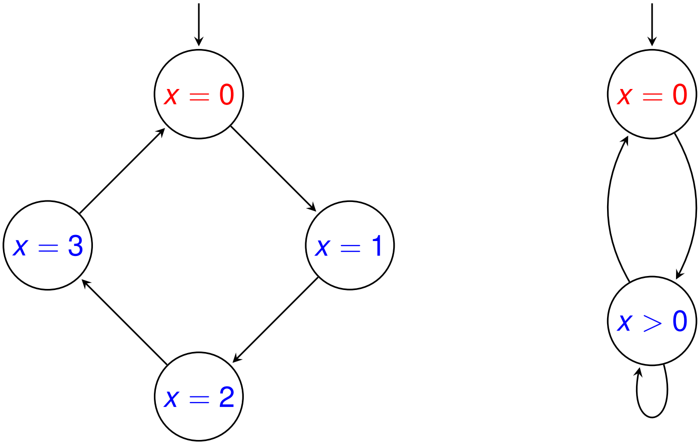


Intuition





Intuition



- equivalent with respect to $\mathsf{F}(x > 0)$
- nonequivalent with respect to $\mathsf{GF}(x = 0)$

Agenda and sources

agenda

- simulation
- exact abstractions
- non-exact abstractions, in particular predicate abstraction
- abstraction in practice
- CEGAR: counterexample-guided abstraction refinement

sources

- Chapter 13 of E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith: *Model Checking*, Second Edition, MIT, 2018.
- R. Pelánek: *Reduction and Abstraction Techniques for Model Checking*, PhD thesis, FI MU, 2006.
- E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith: *Counterexample-guided Abstraction Refinement for Symbolic Model Checking*, J. ACM 50(5), 2003.

Simulation

Simulation

Definition (simulation)

Given two Kripke structures $M = (S, \to, S_0, L)$ and $M' = (S', \to', S'_0, L')$, we say that M' simulates M, written $M \leq M'$, if there exists a relation $R \subseteq S \times S'$ such that:

- $\forall s_0 \in S_0 \,.\, \exists s'_0 \in S'_0 \,.\, (s_0, s'_0) \in R$
- $(s, s') \in R \implies L(s) = L'(s')$
- $(s, s') \in R \wedge s \to p \implies \exists p' \in S' \,.\, s' \to' p' \wedge (p, p') \in R$

Simulation

Definition (simulation)

Given two Kripke structures $M = (S, \rightarrow, S_0, L)$ and $M' = (S', \rightarrow', S_0', L')$, we say that M' simulates M, written $M \leq M'$, if there exists a relation $R \subseteq S \times S'$ such that:

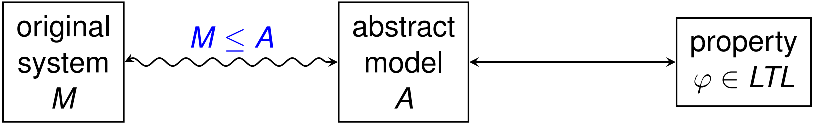
- $\forall s_0 \in S_0 \,.\, \exists s_0' \in S_0' \,.\, (s_0, s_0') \in R$
- $(s, s') \in R \implies L(s) = L'(s')$
- $(s, s') \in R \wedge s \rightarrow p \implies \exists p' \in S' \,.\, s' \rightarrow' p' \wedge (p, p') \in R$

Lemma

If $M \leq M'$, then for every path $\sigma = s_1 s_2 \ldots$ of M starting in an initial state there is a run $\sigma' = s_1' s_2' \ldots$ of M' starting in an initial state and satisfying

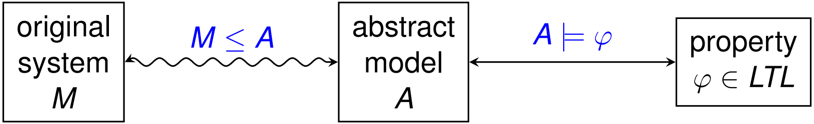
$$L(s_1)L(s_2)\ldots = L'(s_1')L'(s_2')\ldots.$$

Relations between original and abstract systems



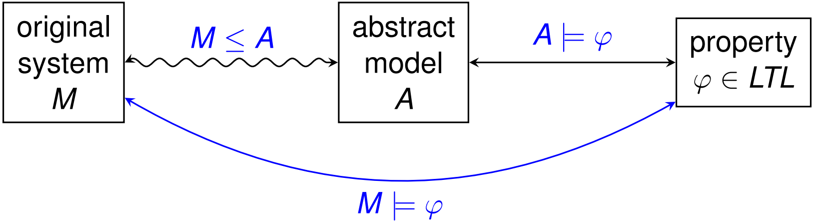
$M \leq A \implies$ all behaviours of M are also in A
(but not vice versa)

Relations between original and abstract systems



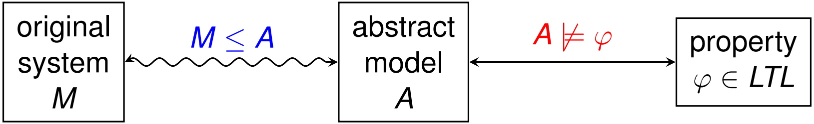
$M \leq A \implies$ all behaviours of M are also in A
(but not vice versa)

Relations between original and abstract systems



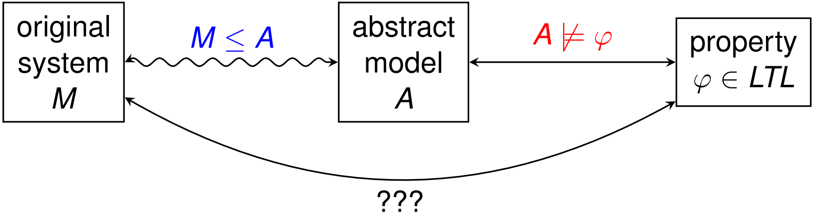
$M \leq A \implies$ all behaviours of M are also in A (but not vice versa)

Relations between original and abstract systems



$M \leq A \implies$ all behaviours of M are also in A
(but not vice versa)

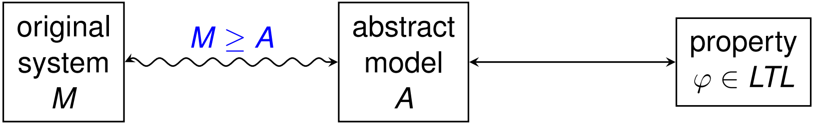
Relations between original and abstract systems



If A has a behaviour violating φ (i.e. $A \not\models \varphi$), then either

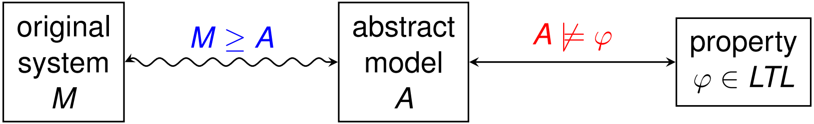
1. M has this behaviour as well (i.e. $M \not\models \varphi$), or
2. M does not have this behaviour, which is then called false positive or spurious counterexample ($M \models \varphi$ or $M \not\models \varphi$ due to another behaviour violating φ).

Relations between original and abstract systems



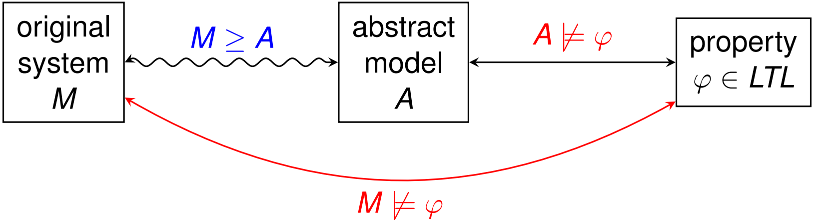
$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

Relations between original and abstract systems



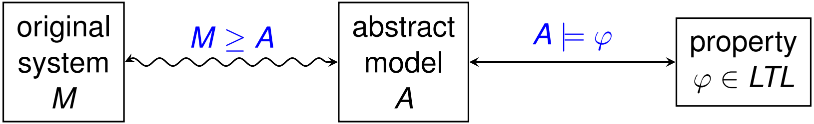
$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

Relations between original and abstract systems



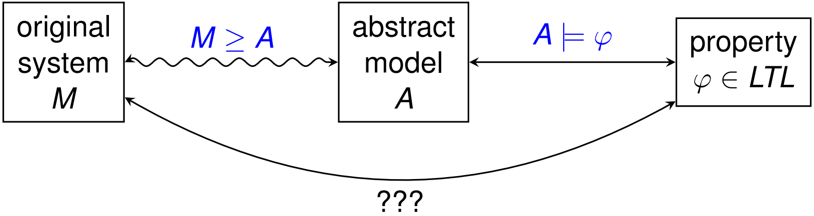
$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

Relations between original and abstract systems



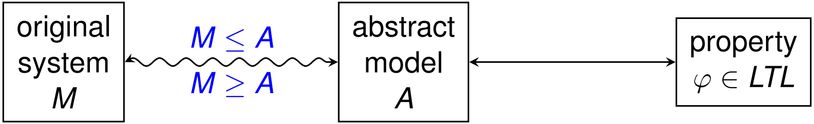
$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

Relations between original and abstract systems



$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

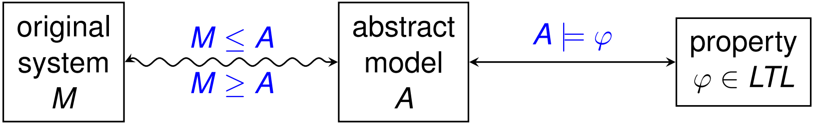
Relations between original and abstract systems



$M \leq A \leq M \implies$ A and M have tha same behaviours
A is an exact abstraction of M

note: A and M are bisimilar $\implies$ $M \leq A \leq M$
$\not\Longleftarrow$

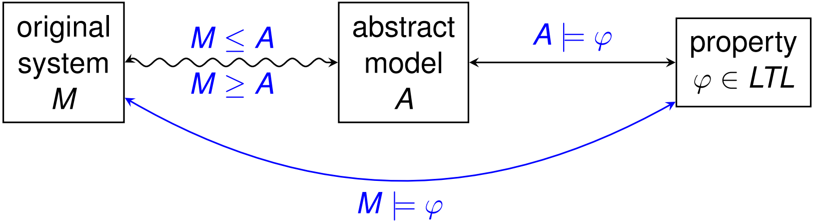
Relations between original and abstract systems



$M \leq A \leq M \implies$ A and M have tha same behaviours
A is an exact abstraction of M

note: A and M are bisimilar $\implies$ $M \leq A \leq M$
$\not\Longleftarrow$

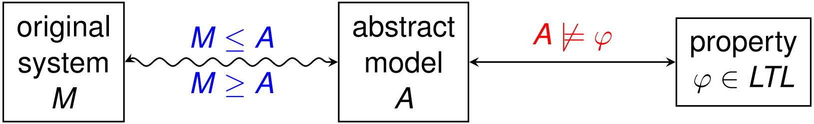
Relations between original and abstract systems



$M \leq A \leq M \implies$ A and M have tha same behaviours
A is an exact abstraction of M

note: A and M are bisimilar $\implies$ $M \leq A \leq M$
$\not\Longleftarrow$

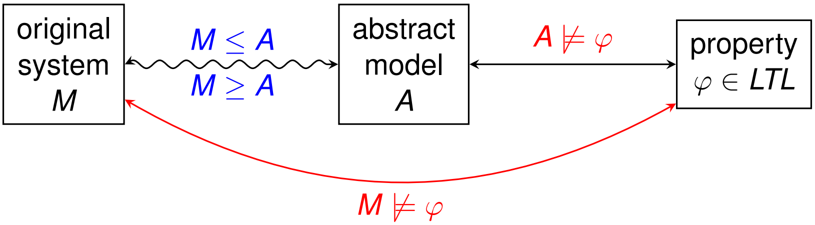
Relations between original and abstract systems



$M \leq A \leq M \implies$ A and M have tha same behaviours
A is an exact abstraction of M

note: A and M are bisimilar $\implies$ $M \leq A \leq M$
$\not\Longleftarrow$

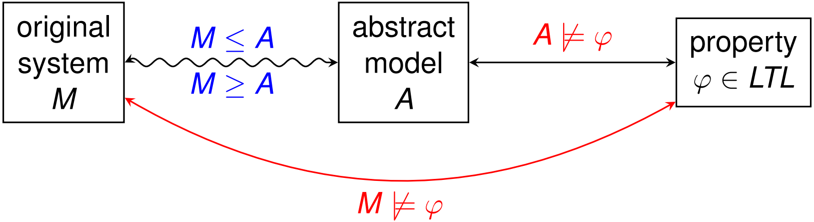
Relations between original and abstract systems



$M \leq A \leq M \implies$ A and M have tha same behaviours
A is an exact abstraction of M

note: A and M are bisimilar $\overset{\implies}{\not\Longleftarrow}$ $M \leq A \leq M$

Relations between original and abstract systems



All these relations hold even for $\varphi \in$ CTL*.

Exact abstractions

Cone of influence (aka dead variables)

Idea

We eliminate the state variables that do not influence the variables in the specification.

Cone of influence (aka dead variables)

- assume that our system is a program
- let V be the set of variables appearing in specification
- cone of influence C of V is the minimal set of variables such that
  - $V \subseteq C$
  - if v occurs in a test affecting the control flow, then $v \in C$
  - if there is an assignment $v := e$ for some $v \in C$, then all variables occurring in the expression e are also in C
- C can be computed by the source code analysis
- variables that are not in C can be eliminated from the code together with all commands they participate in

Cone of influence: example

```
S: v = getinput();
   x = getinput();
   y = 1;
   z = 1;
   while (v > 0) {
     z = z * x;
     x = x - 1;
     y = y * v;
     v = v - 1;
   }
   z = z * y;
E:
```

specification: $\mathrm{F}(pc = \mathtt{E})$

Cone of influence: example

```
S: v = getinput();
   x = getinput();
   y = 1;
   z = 1;
   while (v > 0) {
     z = z * x;
     x = x - 1;
     y = y * v;
     v = v - 1;
   }
   z = z * y;
E:
```

specification: $\mathsf{F}(pc = \mathtt{E})$

$V = \emptyset$, $C = \{\mathtt{v}\}$

Cone of influence: example

```
S: v = getinput();
   x = getinput();
   y = 1;
   z = 1;
   while (v > 0) {
     z = z * x;
     x = x - 1;
     y = y * v;
     v = v - 1;
   }
   z = z * y;
E:
```

```
S: v = getinput();
   skip;
   skip;
   skip;
   while (v > 0) {
     skip;
     skip;
     skip;
     v = v - 1;
   }
   skip;
E:
```

specification: $\mathsf{F}(pc = \mathtt{E})$

$V = \emptyset$, $C = \{\mathtt{v}\}$

Other exact abstractions

symmetry reduction

- in systems with more identical parallel components, their order is not important

equivalent values

- if the set of behaviours starting in a state s is the same for values a, b of a variable v, then the two values can be replaced by one
- applicable to larger sets of values as well
- used in timed automata for timer values

Non-exact abstractions, in particular predicate abstraction

Concept

we face two problems

1. to find a suitable set of abstract states (called abstract domain) and a mapping between the original states and the abstract ones
2. to compute a transition relation on abstract states

Finding abstract states

abstract states are usually defined in one of the following ways

1. for each variable x, we replace the original variable domain D_x by an abstract variable domain A_x and we define a total function $h_x : D_x \to A_x$

   a state $s = (v_1, \dots, v_m) \in D_{x_1} \times \dots \times D_{x_m}$ given by values of all variables corresponds to an abstract state

   $$h(s) = (h_{x_1}(v_1), \dots, h_{x_m}(v_m)) \in A_{x_1} \times \dots \times A_{x_m}$$

2. predicate abstraction - we choose a finite set $\Phi = \{\phi_1, \dots, \phi_n\}$ of predicates over the set of variables;
   we have several choices of an abstract domain

The first approach can be seen as a special case the latter one.

Popular abstract domains for integers

sign abstraction

- $A_x = \{a_+, a_-, a_0\}$
- $h_x(v) = \begin{cases} a_- & \text{if } v < 0 \\ a_0 & \text{if } v = 0 \\ a_+ & \text{if } v > 0 \end{cases}$

parity abstraction

- $A_x = \{a_e, a_o\}$
- $h_x(v) = \begin{cases} a_e & \text{if } v \text{ is even} \\ a_o & \text{if } v \text{ is odd} \end{cases}$
- good for verification of properties related to the last bit of binary representation

Popular abstract domains for integers

congruence modulo an integer

- $A_x = \{0, 1, \ldots, m-1\}$ for some $m > 1$
- $h_x(v) = v \bmod m$
- nice properties

$$
\begin{aligned}
((x \bmod m) + (y \bmod m)) \bmod m &= (x + y) \bmod m \\
((x \bmod m) - (y \bmod m)) \bmod m &= (x - y) \bmod m \\
((x \bmod m) \cdot (y \bmod m)) \bmod m &= (x \cdot y) \bmod m
\end{aligned}
$$

representation by logarithm

- $h_x(v) = \lceil \log_2(v+1) \rceil$
- the number of bits needed to represent v
- good for verification of properties related to overflow problems

Popular abstract domains for integers

single bit abstraction

- $A_x = \{0, 1\}$
- $h_x(v) =$the i-th bit of v for a fixed i

single value abstraction

- $A_x = \{0, 1\}$
- $h_x(v) = \begin{cases} 1 & \text{if } v = c \\ 0 & \text{otherwise} \end{cases}$

...and others

Predicate abstraction

Let $\Phi = \{\phi_1, \dots, \phi_n\}$ be a set of predicates over the set of variables.

abstract domain $\{0,1\}^n$

- a state $s = (v_1, \dots, v_m)$ corresponds to an abstract state given by a vector of truth values of $\{\phi_1, \dots, \phi_n\}$, i.e.,

$$h(s) = (\phi_1(v_1, \dots, v_m), \dots, \phi_n(v_1, \dots, v_m)) \in \{0,1\}^n$$

- example: $\phi_1 = (x_1 > 3) \quad \phi_2 = (x_1 < x_2) \quad \phi_3 = (x_2 > 10)$
  $s = (5, 7)$
  $h(s) = (1, 1, 0)$

assume that

- we have an original Kripke structure $M = (S, \rightarrow, S_0, L)$
- we have an abstract domain A and a mapping $h : S \rightarrow A$

we define an abstract model as a Kripke structure $(A, \rightarrow', A_0, L_A)$, where

- $A_0 = \{h(s_0) \mid s_0 \in S_0\}$
- $L_A : A \rightarrow 2^{AP}$ has to be correctly defined, i.e.,
  - for abstraction based on variable domains, validity of atomic propositions is determined by abstract states in $A_{x_1} \times \ldots \times A_{x_m}$
  - for predicate abstraction, validity of atomic propositions is determined by abstraction predicates $\{\phi_1, \ldots, \phi_n\}$ (AP is typically a subset of it)

  and L_A has to agree with L, i.e., $L(s) = L_A(h(s))$
- $\rightarrow'$ is defined in one of the following ways

May abstraction

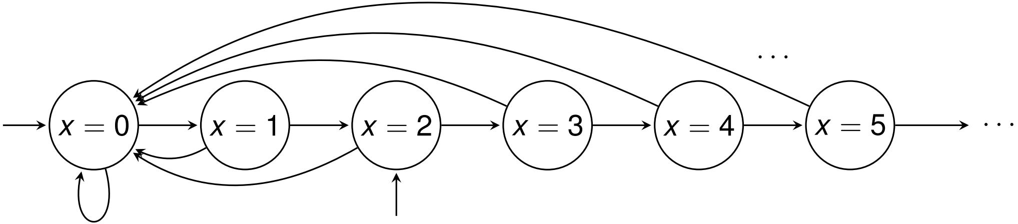
may abstraction produces $M_{may} = (A, \rightarrow_{may}, A_0, L_A)$

- $a_1 \rightarrow_{may} a_2$ iff there exist $s_1, s_2 \in S$ such that $h(s_1) = a_1$, $h(s_2) = a_2$, $s_1 \rightarrow s_2$

May abstraction

may abstraction produces $M_{may} = (A, \rightarrow_{may}, A_0, L_A)$

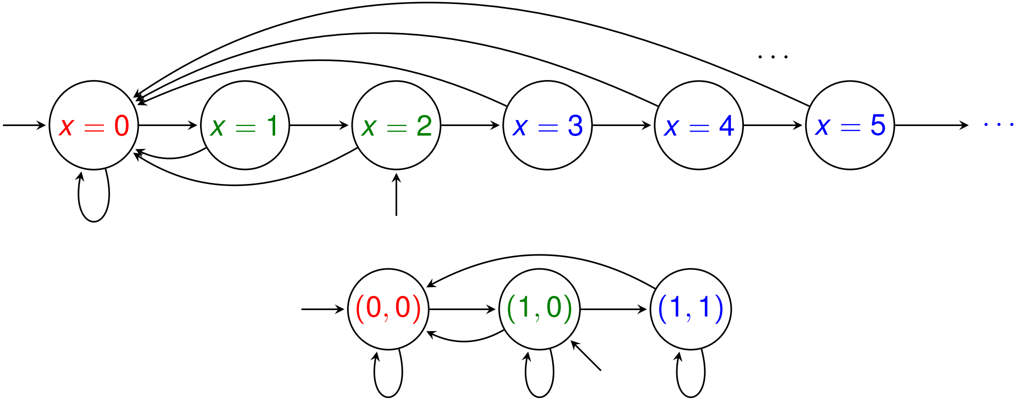
- $a_1 \rightarrow_{may} a_2$ iff there exist $s_1, s_2 \in S$ such that $h(s_1) = a_1$, $h(s_2) = a_2$, $s_1 \rightarrow s_2$
- example: construct M_{may} for the following system using predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and abstract domain $\{0, 1\}^2$



May abstraction

may abstraction produces $M_{may} = (A, \rightarrow_{may}, A_0, L_A)$

- $a_1 \rightarrow_{may} a_2$ iff there exist $s_1, s_2 \in S$ such that $h(s_1) = a_1$, $h(s_2) = a_2$, $s_1 \rightarrow s_2$
- example: construct M_{may} for the following system using predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and abstract domain $\{0, 1\}^2$



Must abstraction

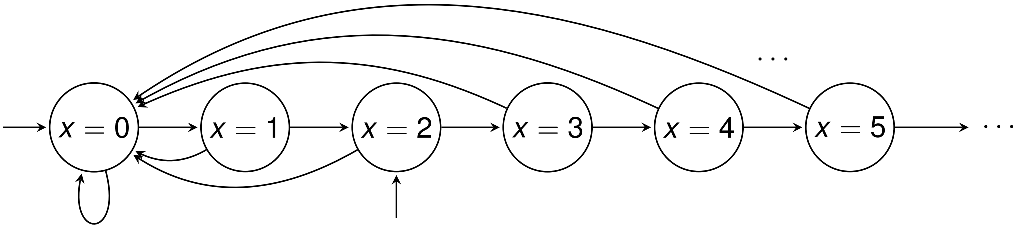
must abstraction produces $M_{must} = (A, \rightarrow_{must}, A_0, L_A)$

- $a_1 \rightarrow_{must} a_2$ iff for each $s_1 \in S$ satisfying $h(s_1) = a_1$ there exists $s_2 \in S$ such that $h(s_2) = a_2$ and $s_1 \rightarrow s_2$

Must abstraction

must abstraction produces $M_{must} = (A, \rightarrow_{must}, A_0, L_A)$

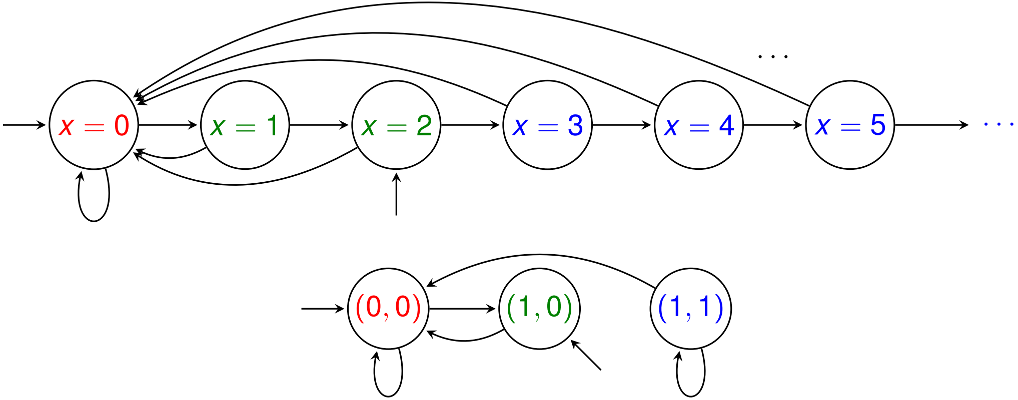
- $a_1 \rightarrow_{must} a_2$ iff for each $s_1 \in S$ satisfying $h(s_1) = a_1$ there exists $s_2 \in S$ such that $h(s_2) = a_2$ and $s_1 \rightarrow s_2$
- example: construct M_{must} for the following system using predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and abstract domain $\{0, 1\}^2$



Must abstraction

must abstraction produces $M_{must} = (A, \rightarrow_{must}, A_0, L_A)$

- $a_1 \rightarrow_{must} a_2$ iff for each $s_1 \in S$ satisfying $h(s_1) = a_1$ there exists $s_2 \in S$ such that $h(s_2) = a_2$ and $s_1 \rightarrow s_2$
- example: construct M_{must} for the following system using predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and abstract domain $\{0, 1\}^2$



Relations between M, M_{must}, and M_{may}

Lemma

For every Kripke structure M, abstract domain A with a mapping function h it holds

$$M_{must} \;\leq\; M \;\leq\; M_{may}.$$

Relations between M, M_{must}, and M_{may}

Lemma

For every Kripke structure M, abstract domain A with a mapping function h it holds

$$M_{must} \;\leq\; M \;\leq\; M_{may}.$$

- computing M_{must} or M_{may} requires constructing M first (recall that M can be very large or even infinite)
- we rather compute an under-approximation M'_{must} of M_{must} or an over-approximation M'_{may} of M_{may} directly from the implicit representation of M
- it holds that $M'_{must} \leq M_{must} \leq M \leq M_{may} \leq M'_{may}$

Abstraction in practice

Predicate abstraction: abstracting sets of states

Abstract domain $\{0,1\}^n$ can lead to too many transitions $\Longrightarrow$ it is sometimes better to assign a single abstract state to a set of original states.

abstract domain $2^{\{0,1\}^n}$

- let $\vec{b} = \langle b_1, \dots, b_n \rangle$ be a vector of $b_i \in \{0,1\}$
- we set $[\vec{b}, \Phi] = b_1 \cdot \phi_1 \wedge \dots \wedge b_n \cdot \phi_n$, where $0 \cdot \phi_i = \neg \phi_i$ and $1 \cdot \phi_i = \phi_i$
- let X denotes the set of original states
- $h(X) = \{\vec{b} \in \{0,1\}^n \mid \exists s \in X : s \models [\vec{b}, \Phi]\}$
- example: $\phi_1 = (x_1 > 3) \quad \phi_2 = (x_1 < x_2) \quad \phi_3 = (x_2 > 10)$
  $X = \{(5,7), (4,5), (2,9)\}$
  $h(X) = \{(1,1,0), (0,1,0)\}$
- nice theoretical properties
- not used in practice (this abstract domain grows too fast)

Predicate abstraction: abstracting sets of states

abstract domain $\{0, 1, *\}^n$ (predicate-cartesian abstraction)

- let $\vec{b} = \langle b_1, \dots, b_n \rangle$ be a vector of $b_i \in \{0, 1, *\}$
- we set $[\vec{b}, \Phi] = b_1 \cdot \phi_1 \wedge \dots \wedge b_n \cdot \phi_n$, where $0 \cdot \phi_i = \neg\phi_i$, $1 \cdot \phi_i = \phi_i$, $* \cdot \phi_i = \top$
- $h(X) = \min\{\vec{b} \in \{0, 1, *\}^n \mid \forall s \in X : s \models [\vec{b}, \Phi]\}$, where min means “the most specific”
- example: $\phi_1 = (x_1 > 3) \quad \phi_2 = (x_1 < x_2) \quad \phi_3 = (x_2 > 10)$
  $X = \{(5, 7), (4, 5), (2, 9)\}$
  $h(X) = (*, 1, 0)$
- this one is sometimes used in practice

Guarded command language

syntax

- let V be a finite set of integer variables
- Act is a set of action names
- model is a pair $M = (V, E)$, where $E = \{t_1, \ldots, t_m\}$ is a finite set of transitions of the form $t_i = (a_i, g_i, u_i)$, where
  - $a_i \in Act$
  - g_i is a first-order formula called guard and built with V, integers, standard binary operations $(+, -, \cdot, \ldots)$ and relations $(=, <, >, \ldots)$
  - u_i is a finite sequence of assignments $x := e$, where $x \in V$ and e is an expression built with V, integers, and standard binary operations $(+, -, \cdot, \ldots)$

Guarded command language

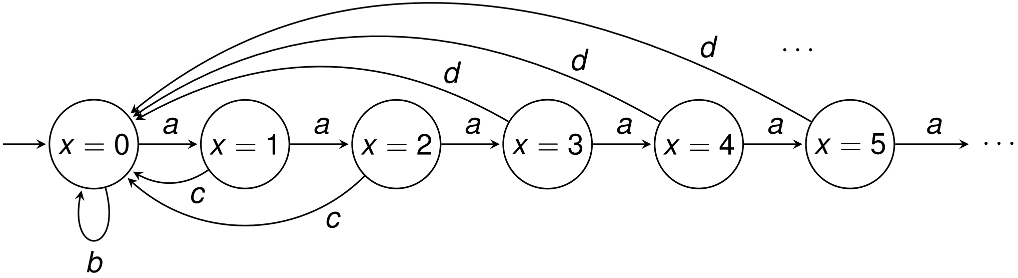
syntax

- let V be a finite set of integer variables
- Act is a set of action names
- model is a pair $M = (V, E)$, where $E = \{t_1, \ldots, t_m\}$ is a finite set of transitions of the form $t_i = (a_i, g_i, u_i)$, where
  - $a_i \in Act$
  - g_i is a first-order formula called guard and built with V, integers, standard binary operations $(+, -, \cdot, \ldots)$ and relations $(=, <, >, \ldots)$
  - u_i is a finite sequence of assignments $x := e$, where $x \in V$ and e is an expression built with V, integers, and standard binary operations $(+, -, \cdot, \ldots)$

semantics

- M defines a labelled transition system where
  - states are valuations of variables $S = 2^{V \to \mathbb{Z}}$
  - initial state is the zero valuation $s_0(v) = 0$ for all $v \in V$
  - $s \xrightarrow{a_i} s'$ whenever $s \models g_i$ and s' arises from s by applying the assignments in u_i
- M can also describe a Kripke structure if we add a labelling function

Example



implicit description in guarded command language by model (V, E), where

$$
\begin{aligned}
V &= \{x\} \\
E &= \{(a,\ \top, && x := x + 1), \\
&\quad\ (b,\ \neg(x > 0), && x := 0), \\
&\quad\ (c,\ (x > 0) \wedge (x \leq 2), && x := 0), \\
&\quad\ (d,\ (x > 2), && x := 0)\}
\end{aligned}
$$

Abstraction in practice

- we use predicate abstraction with domain $\{0, 1, *\}^n$
- given a formula φ with free variables $\vec{x}$ from V, we set

$$pre(a_i, \varphi) = (g_i \implies \varphi[\vec{x}/u_i(\vec{x})])$$

  where $\varphi[\vec{x}/u_i(\vec{x})]$ denotes the formula φ with each free variable x replaced by $u_i(x)$, which is the expression representing the value of x after the assignments in u_i
- intuitively, $pre(a_i, \varphi)$ transforms the condition φ to the situation before taking the transition (a_i, g_i, u_i)
- we use a sound (potentially not complete) decision procedure *is_valid*, i.e.,

$$is\_valid(\varphi) = \top \implies \varphi \text{ is a tautology}$$

Abstraction in practice

for every abstract state $\vec{b} \in \{0, 1, *\}^n$ and for every transition $t_i = (a_i, g_i, u_i)$, we compute an over-approximation of a *may*-successor of $\vec{b}$ under t_i as

- if $is\_valid([\vec{b}, \Phi] \implies \neg g_i)$ then there is no successor
- otherwise, the successor $\vec{b'}$ is given by

$$b'_j = \begin{cases} 1 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \phi_j)) \\ 0 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \neg\phi_j)) \\ * & \text{otherwise} \end{cases}$$

Abstraction in practice

for every abstract state $\vec{b} \in \{0, 1, *\}^n$ and for every transition $t_i = (a_i, g_i, u_i)$, we compute an over-approximation of a *may*-successor of $\vec{b}$ under t_i as

- if $is\_valid([\vec{b}, \Phi] \implies \neg g_i)$ then there is no successor
- otherwise, the successor $\vec{b'}$ is given by

$$b'_j = \begin{cases} 1 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \phi_j)) \\ 0 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \neg\phi_j)) \\ * & \text{otherwise} \end{cases}$$

- example: consider the abstract state $\vec{b} = (1, 0)$ where $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and compute the successor corresponding to $(a, \top, x := x + 1)$

$$(1, 0) \xrightarrow{a}_{may'} (\quad , \quad)$$

Abstraction in practice

for every abstract state $\vec{b} \in \{0, 1, *\}^n$ and for every transition $t_i = (a_i, g_i, u_i)$, we compute an over-approximation of a *may*-successor of $\vec{b}$ under t_i as

- if $is\_valid([\vec{b}, \Phi] \implies \neg g_i)$ then there is no successor
- otherwise, the successor $\vec{b'}$ is given by

$$b'_j = \begin{cases} 1 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \phi_j)) \\ 0 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \neg\phi_j)) \\ * & \text{otherwise} \end{cases}$$

- example: consider the abstract state $\vec{b} = (1, 0)$ where $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and compute the successor corresponding to $(a, \top, x := x + 1)$

$$(1, 0) \xrightarrow{a}_{may'} (1, \quad)$$

  - $(x > 0) \wedge (x \leq 2) \implies (\top \implies (x + 1 > 0))$ is true

Abstraction in practice

for every abstract state $\vec{b} \in \{0, 1, *\}^n$ and for every transition $t_i = (a_i, g_i, u_i)$, we compute an over-approximation of a *may*-successor of $\vec{b}$ under t_i as

- if $is\_valid([\vec{b}, \Phi] \implies \neg g_i)$ then there is no successor
- otherwise, the successor $\vec{b'}$ is given by

$$b'_j = \begin{cases} 1 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \phi_j)) \\ 0 & \text{if } is\_valid([\vec{b}, \Phi] \implies pre(a_i, \neg\phi_j)) \\ * & \text{otherwise} \end{cases}$$

- example: consider the abstract state $\vec{b} = (1, 0)$ where $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$ and compute the successor corresponding to $(a, \top, x := x + 1)$

$$(1, 0) \xrightarrow{a}_{may'} (1, *)$$

  - $(x > 0) \wedge (x \leq 2) \implies (\top \implies (x + 1 > 0))$ is true
  - $(x > 0) \wedge (x \leq 2) \implies (\top \implies (x + 1 > 2))$ is not true
  - $(x > 0) \wedge (x \leq 2) \implies (\top \implies (x + 1 \leq 2))$ is not true

Abstraction in practice

- for every transition, we compute successors of all abstract states
- based on the successors, we transform the original implicit representation of a system into a Boolean program
- it is very similar to a model in guarded command language, but instead of integers it uses only Boolean variables $\vec{b}$ representing the validity of abstraction predicates Φ
- Boolean program is an implicit representation of an over-approximation of M_{may}
- Boolean program can be used as an input for a suitable model checker (of finite-state systems)

Example

- consider the model (V, E), where

$$\begin{aligned}
V &= \{x\} \\
E &= \{(a,\ \top,\ & x := x + 1), \\
&\quad (b,\ \neg(x > 0),\ & x := 0), \\
&\quad (c,\ (x > 0) \land (x \leq 2),\ & x := 0), \\
&\quad (d,\ (x > 2),\ & x := 0)\}
\end{aligned}$$

- using the predicates $\phi_1 = (x > 0)$, $\phi_2 = (x > 2)$, we get the following Boolean program defining an over-approximation of M_{may}

$$\begin{aligned}
V &= \{b_1, b_2\}, \text{ where } b_1, b_2 \text{ represents the validity of } \phi_1, \phi_2 \\
E &= \{(a,\ \top,\ & b_1 := \mathit{if}\ b_1\ \mathit{then}\ 1\ \mathit{else}\ *; \\
&& b_2 := \mathit{if}\ b_2\ \mathit{then}\ 1\ \mathit{else\ if}\ b_1\ \mathit{then}\ *\ \mathit{else}\ 0), \\
&\quad (b,\ \neg b_1,\ & b_1 := 0;\ b_2 := 0), \\
&\quad (c,\ b_1 \land \neg b_2,\ & b_1 := 0;\ b_2 := 0), \\
&\quad (d,\ b_2,\ & b_1 := 0;\ b_2 := 0)\}
\end{aligned}$$

Example of a real NQC code and its absraction

```
task light_sensor_control() {
  int x = 0;
  while (true) {
    if (LIGHT > LIGHT_THRESHOLD) {
      PlaySound(SOUND_CLICK);
      Wait(30);
      x = x + 1;
    } else {
      if (x > 2) {
        PlaySound(SOUND_UP);
        ClearTimer(0);
        brick = LONG;
      } else if (x > 0) {
        PlaySound(SOUND_DOUBLE_BEEP);
        ClearTimer(0);
        brick = SHORT;
      }
      x = 0;
    }
  }
}
```

Example of a real NQC code and its absraction

```
task light_sensor_control() {
  int x = 0;
  while (true) {
    if (LIGHT > LIGHT_THRESHOLD) {
      PlaySound(SOUND_CLICK);
      Wait(30);
      x = x + 1;
    } else {
      if (x > 2) {
        PlaySound(SOUND_UP);
        ClearTimer(0);
        brick = LONG;
      } else if (x > 0) {
        PlaySound(SOUND_DOUBLE_BEEP);
        ClearTimer(0);
        brick = SHORT;
      }
      x = 0;
    }
  }
}
```

```
task A_light_sensor_control() {
  bool b = false;
  while (true) {
    if (*) {


      b = b ? true : * ;
    } else {
      if (b) {


        brick = LONG;
      } else if (b ? true : *) {


        brick = SHORT;
      }
      b = false;
    }
  }
}
```

CEGAR: counterexample-guided abstraction refinement

Motivation

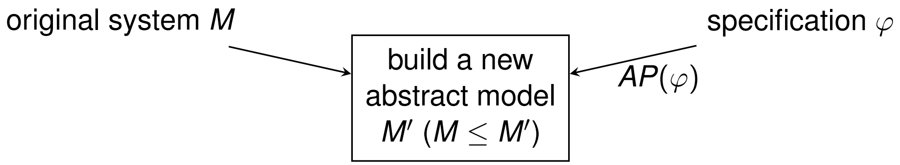
- it is hard to find a small and valuable abstraction
- abstraction predicates were originally provided by a user
- CEGAR tries to find a suitable abstraction automatically
- implemented in SLAM, BLAST, Static Driver Verifier (SDV), and many others
- incomplete method, but very successfull in practice

Principle of CEGAR

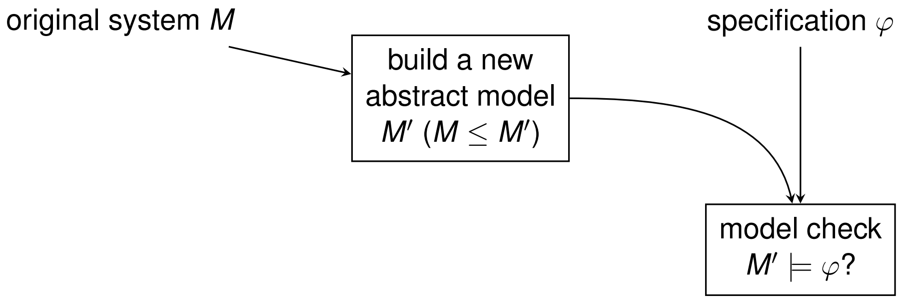
original system M

specification φ

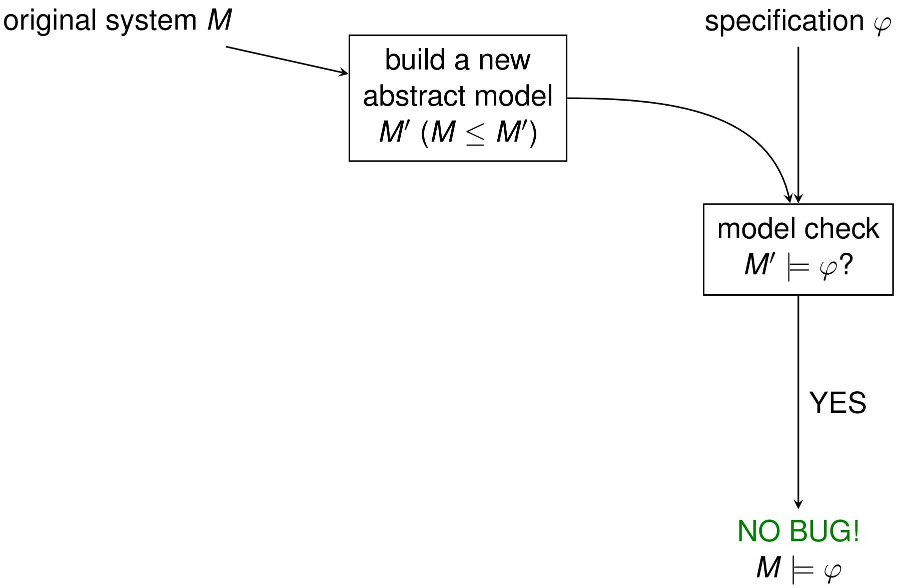
Principle of CEGAR



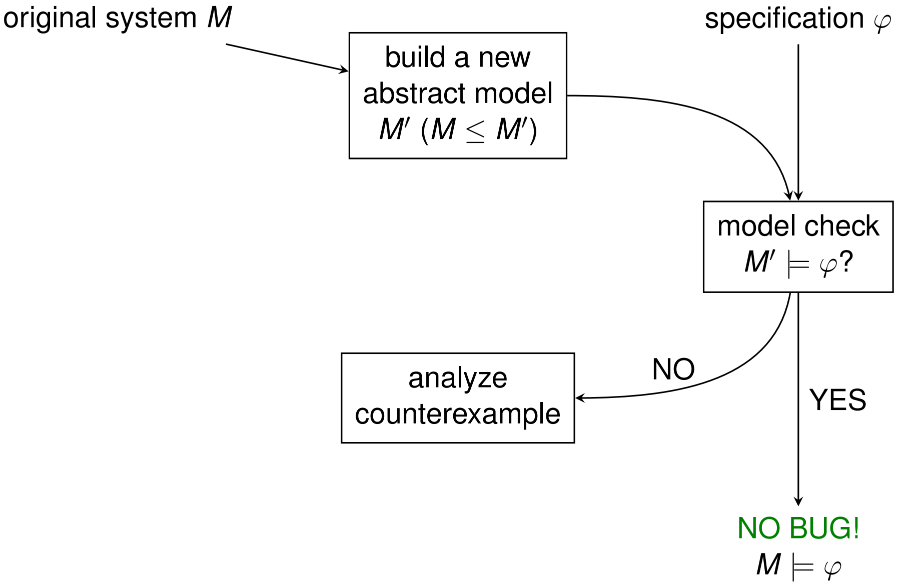
Principle of CEGAR



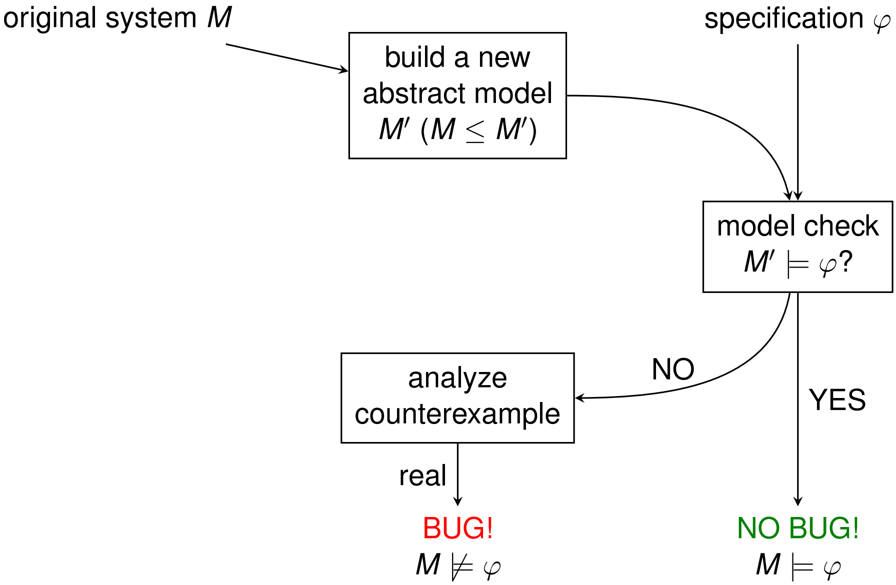
Principle of CEGAR



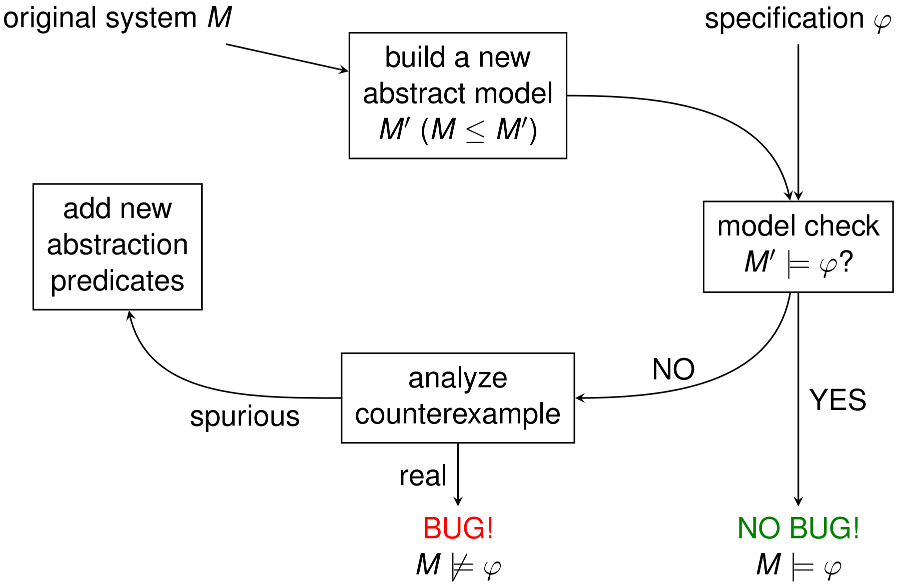
Principle of CEGAR

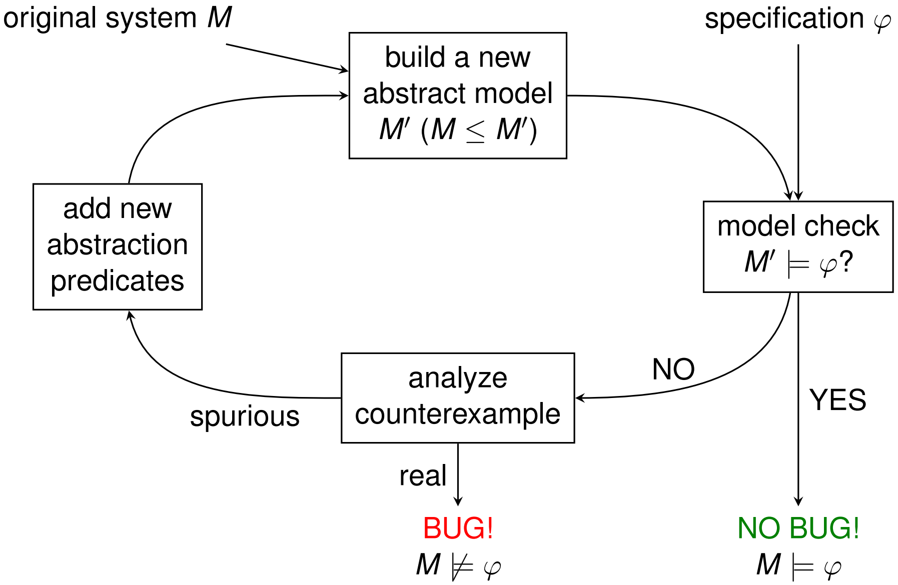


Principle of CEGAR



Principle of CEGAR



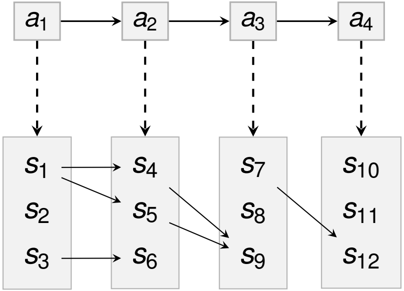

original system M
specification φ
build a new abstract model M' ($M \leq M'$)
model check $M' \models \varphi$?
add new abstraction predicates
analyze counterexample
spurious
NO
YES
real
BUG!
$M \not\models \varphi$
NO BUG!
$M \models \varphi$

Notes

- added abstraction predicates ensure that the new abstract model M' does not have the behaviour corresponding to the spurious counterexample of the previous M'
- the analysis of an abstract counterexample and finding new abstract predicates are nontrivial tasks
- the method is sound but incomplete: the algorithm can run in the cycle forever or fail to find new abstraction predicates

Counterexample analysis

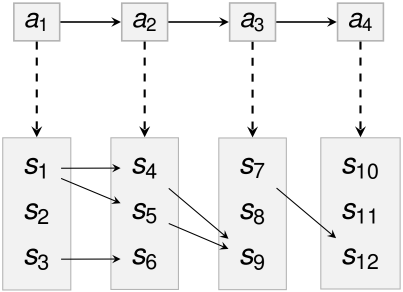
- an abstract path is a finite or infinite path in an abstract model
- an abstract path $a_1 a_2 \ldots$ is real if there exists a path $s_1 s_2 \ldots$ in the original system M of the same length such that s_1 is initial and $s_i \in h^{-1}(a_i)$ for all i
- an abstract path that is not real is called spurious



Analysis of finite counterexamples (e.g. for reachability)

input : a nonempty abstract path $a_1 \dots a_n$,
an original system $M = (S, \rightarrow, S_0, L)$, an abstraction function h
output: "real" if the path is real; j, R' otherwise, where j is the length of the maximal real prefix of the path and R' is the set of the last states of the paths in M corresponding to the prefix

$R \leftarrow h^{-1}(a_1) \cap S_0$
if $R = \emptyset$ **then return** $0, \emptyset$ `// spurious`
$j \leftarrow 1$
while $R \neq \emptyset \wedge j < n$ **do**
  $j \leftarrow j + 1$
  $R' \leftarrow R$
  $R \leftarrow \{s \mid \exists s' \in R \,.\, s' \rightarrow s\} \cap h^{-1}(a_j)$
if $R \neq \emptyset$ **then return** real
return $j - 1, R'$ `// spurious`

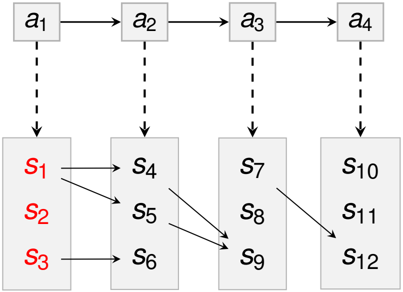


Analysis of finite counterexamples (e.g. for reachability)

input : a nonempty abstract path $a_1 \ldots a_n$,
an original system $M = (S, \to, S_0, L)$, an abstraction function h

output: "real" if the path is real; j, R' otherwise, where j is the length of the maximal real prefix of the path and R' is the set of the last states of the paths in M corresponding to the prefix

$R \leftarrow h^{-1}(a_1) \cap S_0$
if $R = \emptyset$ **then return** $0, \emptyset$ `// spurious`
$j \leftarrow 1$
while $R \neq \emptyset \,\wedge\, j < n$ **do**
  $j \leftarrow j + 1$
  $R' \leftarrow R$
  $R \leftarrow \{s \mid \exists s' \in R \,.\, s' \to s\} \cap h^{-1}(a_j)$
if $R \neq \emptyset$ **then return** real
return $j - 1, R'$ `// spurious`

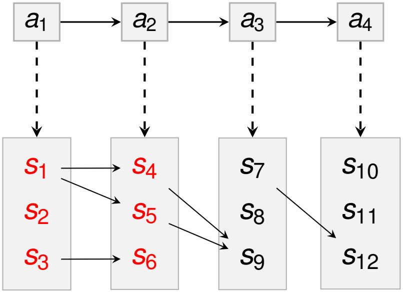


Analysis of finite counterexamples (e.g. for reachability)

input : a nonempty abstract path $a_1 \ldots a_n$,
an original system $M = (S, \rightarrow, S_0, L)$, an abstraction function h

output: "real" if the path is real; j, R' otherwise, where j is the length of the maximal real prefix of the path and R' is the set of the last states of the paths in M corresponding to the prefix

$R \leftarrow h^{-1}(a_1) \cap S_0$
if $R = \emptyset$ **then return** $0, \emptyset$ `// spurious`
$j \leftarrow 1$
while $R \neq \emptyset \wedge j < n$ **do**
  $j \leftarrow j + 1$
  $R' \leftarrow R$
  $R \leftarrow \{s \mid \exists s' \in R \,.\, s' \rightarrow s\} \cap h^{-1}(a_j)$
if $R \neq \emptyset$ **then return** real
return $j - 1, R'$ `// spurious`

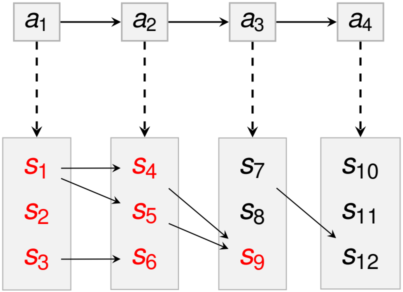


Analysis of finite counterexamples (e.g. for reachability)

input : a nonempty abstract path $a_1 \ldots a_n$,
an original system $M = (S, \rightarrow, S_0, L)$, an abstraction function h

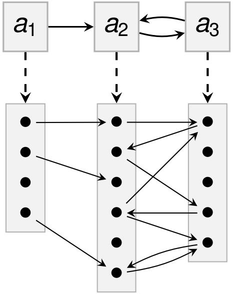
output: "real" if the path is real; j, R' otherwise, where j is the length of the maximal real prefix of the path and R' is the set of the last states of the paths in M corresponding to the prefix

$R \leftarrow h^{-1}(a_1) \cap S_0$
if $R = \emptyset$ **then return** $0, \emptyset$ `// spurious`
$j \leftarrow 1$
while $R \neq \emptyset \ \wedge \ j < n$ **do**
  $j \leftarrow j + 1$
  $R' \leftarrow R$
  $R \leftarrow \{s \mid \exists s' \in R \,.\, s' \rightarrow s\} \cap h^{-1}(a_j)$
if $R \neq \emptyset$ **then return** real
return $j - 1, R'$ `// spurious`

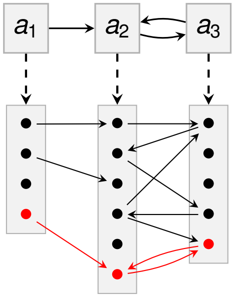


produced output: $3, \{s_9\}$

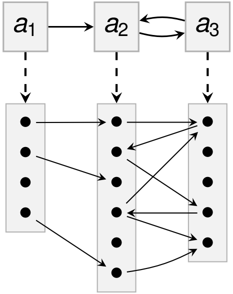
Analysis of lasso-shaped counterexamples



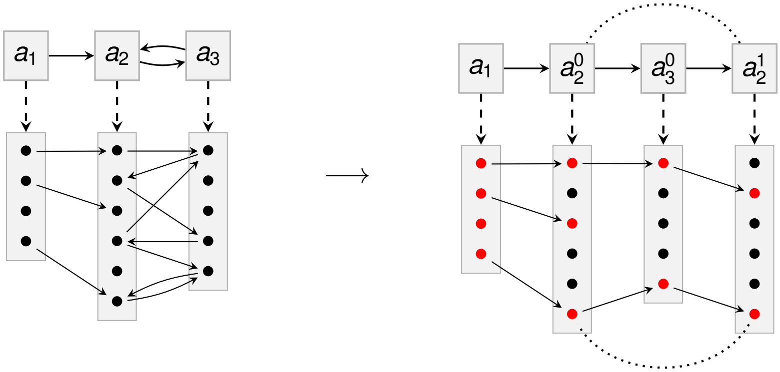
Analysis of lasso-shaped counterexamples



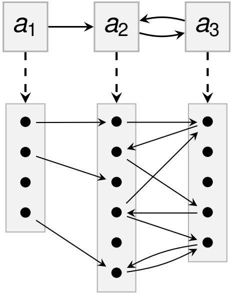
Analysis of lasso-shaped counterexamples



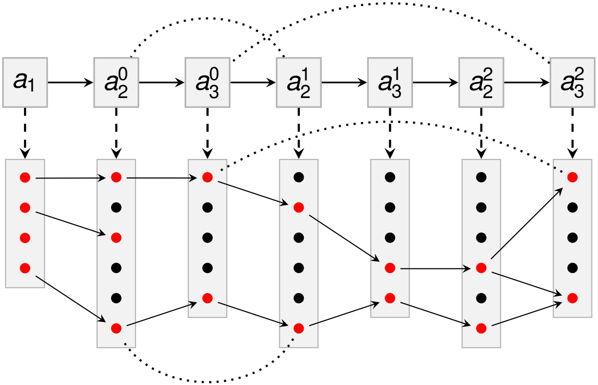
Analysis of lasso-shaped counterexamples



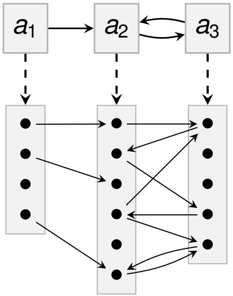
Analysis of lasso-shaped counterexamples

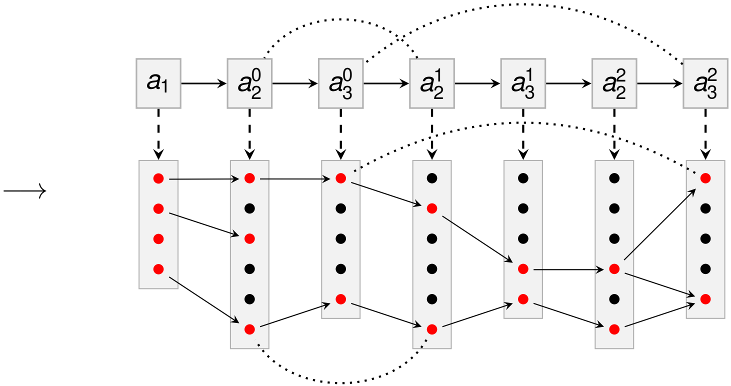


$\longrightarrow$



Analysis of lasso-shaped counterexamples





- an abstract loop may correspond to loops of different size and starting at different stages of the unwinding
- the unwinding eventually becomes periodic, the size of the period is the least common multiple of the size of individual loops

Analysis of lasso-shaped counterexamples

Analysis of a lasso-shaped counterexample can be reduced to analysis of a finite path counterexample.

Theorem

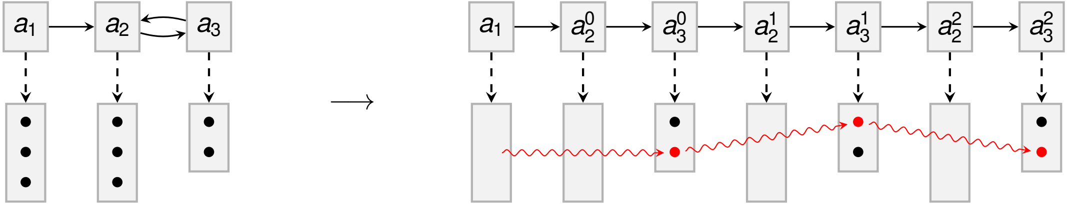
An abstract lasso-shaped path $a_1 \ldots a_i(a_{i+1} \ldots a_n)^\omega$ *is real iff the abstract path* $a_1 \ldots a_i(a_{i+1} \ldots a_n)^{m+1}$ *is real, where* $m = \min_{i+1 \leq j \leq n} |h^{-1}(a_j)|$.

Analysis of lasso-shaped counterexamples

Analysis of a lasso-shaped counterexample can be reduced to analysis of a finite path counterexample.

Theorem

An abstract lasso-shaped path $a_1 \dots a_i(a_{i+1} \dots a_n)^\omega$ is real iff the abstract path $a_1 \dots a_i(a_{i+1} \dots a_n)^{m+1}$ is real, where $m = \min_{i+1 \leq j \leq n} |h^{-1}(a_j)|$.

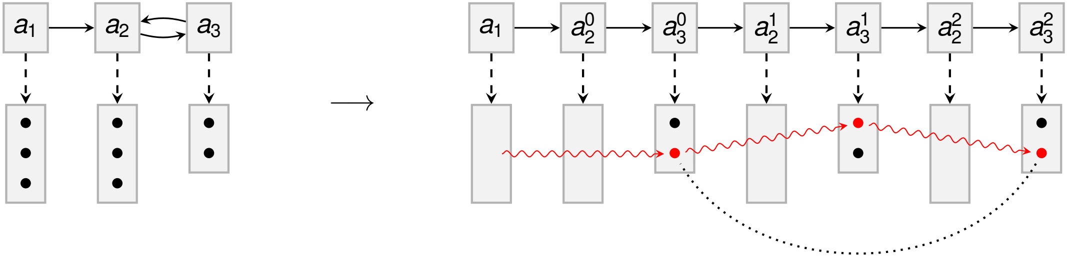


Analysis of lasso-shaped counterexamples

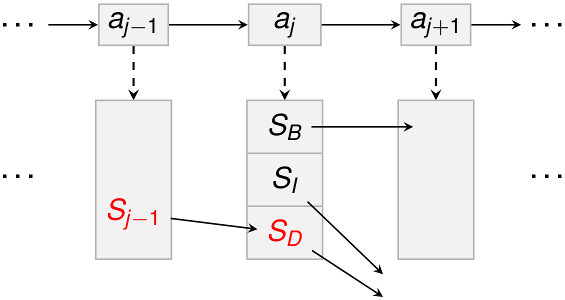
Analysis of a lasso-shaped counterexample can be reduced to analysis of a finite path counterexample.

Theorem

An abstract lasso-shaped path $a_1 \dots a_i(a_{i+1} \dots a_n)^\omega$ *is real iff the abstract path* $a_1 \dots a_i(a_{i+1} \dots a_n)^{m+1}$ *is real, where* $m = \min_{i+1 \leq j \leq n} |h^{-1}(a_j)|$.



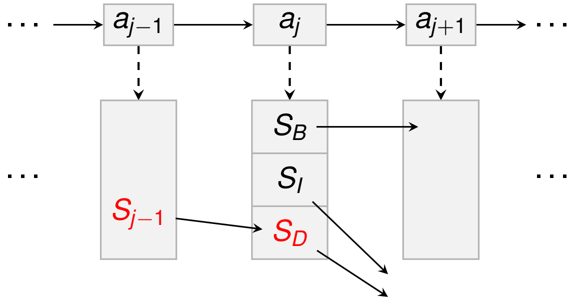
Abstraction refinement



$$
\begin{aligned}
S_B &= h^{-1}(a_j) \cap \{s \mid \exists s' \in h^{-1}(a_{j+1}) \,.\, s \to s'\} \\
S_I &= h^{-1}(a_j) \smallsetminus (S_B \cup S_D) \\
S_D &= S_j
\end{aligned}
$$

bad states
irrelevant states
dead-end states

Abstraction refinement



$$
\begin{aligned}
S_B &= h^{-1}(a_j) \cap \{s \mid \exists s' \in h^{-1}(a_{j+1}) \,.\, s \to s'\} && \text{bad states} \\
S_I &= h^{-1}(a_j) \setminus (S_B \cup S_D) && \text{irrelevant states} \\
S_D &= S_j && \text{dead-end states}
\end{aligned}
$$

- to eliminate the spurious counterexample, we need to refine the abstraction such that no abstract state contains states from both S_B and S_D
- typically, we add an abstraction predicate that is an interpolant of S_B and S_D

Abstraction refinement

Consider abstract state $(3 \leq x \leq 5) \wedge (7 \leq y \leq 9)$ and S_B, S_I, S_D:

	3	4	5
7	B	I	I
8	D	I	B
9	I	D	D

Abstraction refinement

Consider abstract state $(3 \leq x \leq 5) \wedge (7 \leq y \leq 9)$ and S_B, S_I, S_D:

	3	4	5
7	B	I	I
8	D	I	B
9	I	D	D

$\longrightarrow$

	3 4	5
7	B + I	I
8	D + I	B
9	D + I	D

or

<table>
<tr><th></th><th>3</th><th>4 5</th></tr>
<tr><td>7
9</td><td>B + I</td><td>D + I</td></tr>
<tr><td>8</td><td>D</td><td>B + I</td></tr>
</table>

?

- there could be more possible abstraction refinements
- we want the coarsest refinement (i.e., with the least number of abstract states)

Abstraction refinement

Consider abstract state $(3 \leq x \leq 5) \wedge (7 \leq y \leq 9)$ and S_B, S_I, S_D:

	3	4	5
7	B	I	I
8	D	I	B
9	I	D	D

$\longrightarrow$

	3 4	5
7	B + I	I
8	D + I	B
9	D + I	D

or

	3	4 5
7, 9	B + I	D + I
8	D	B + I

?

- there could be more possible abstraction refinements
- we want the coarsest refinement (i.e., with the least number of abstract states)

Theorem

The problem of finding the coarsest refinement is NP-hard.

- there are heuristics that select suitable refinements