IA159 Formal Verification Methods

Abstraction

Jan Strejček

Department of Computer Science
Faculty of Informatics
Masaryk University

Focus and sources

Focus

- principle of abstraction
- exact abstractions and non-exact abstractions
- predicate abstraction
- CEGAR: counterexample-guided abstraction refinement

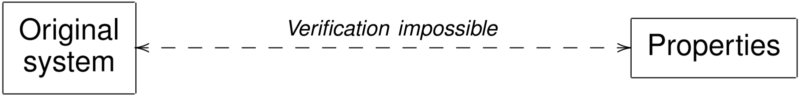
Sources

- Chapter 13 of *E. M. Clarke, O. Grumberg, and D. A. Peled: Model Checking, MIT, 1999.*
- R. Pelánek: *Reduction and Abstraction Techniques for Model Checking*, PhD thesis, FI MU, 2002.
- E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith: *Counterexample-guided Abstraction Refinement*, CAV 2000, LNCS 1855, Springer, 2000.

Motivation

Abstraction is probably the most important technique for reducing the state explosion problem.

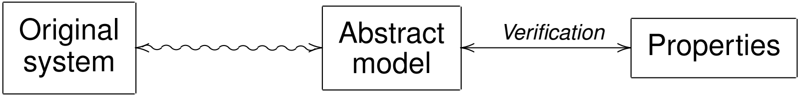
[CGP99]



Motivation

Abstraction is probably the most important technique for reducing the state explosion problem.

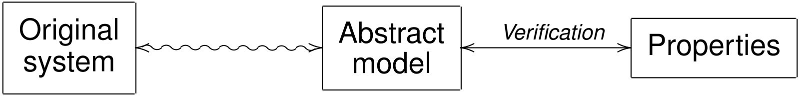
[CGP99]



Motivation

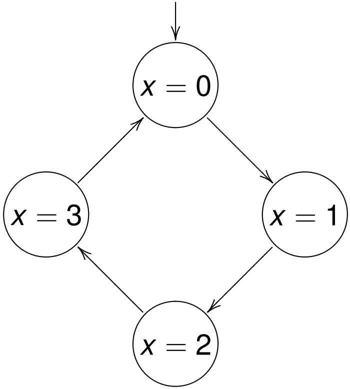
Abstraction is probably the most important technique for reducing the state explosion problem.

[CGP99]

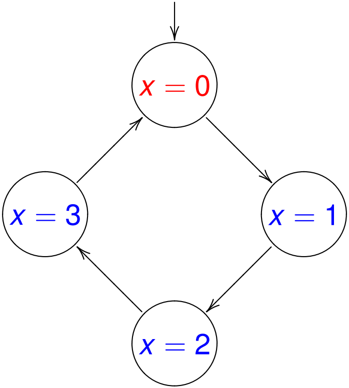


- large finite systems $\longrightarrow$ smaller finite systems
- infinite-state systems $\longrightarrow$ finite systems

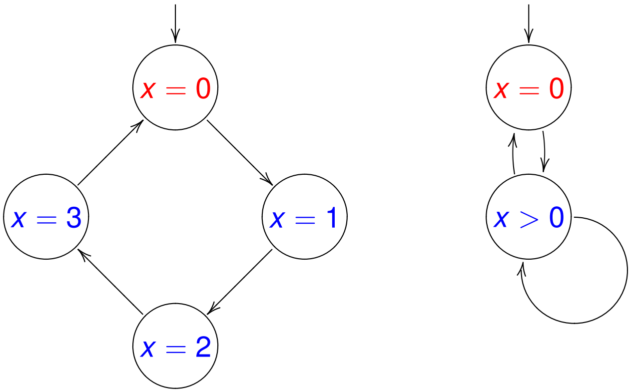
Intuition



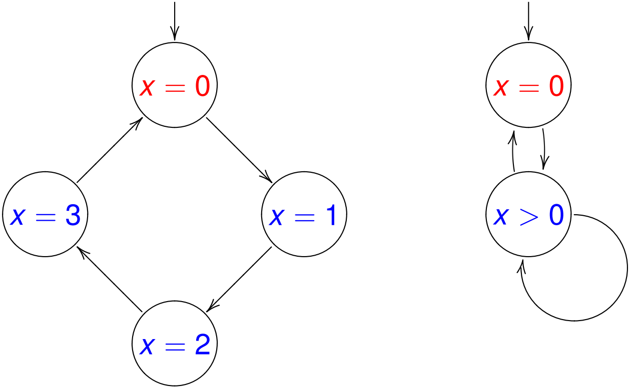
Intuition



Intuition



Intuition



- equivalent with respect to $F(x > 0)$
- nonequivalent with respect to $GF(x = 0)$

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:

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

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:

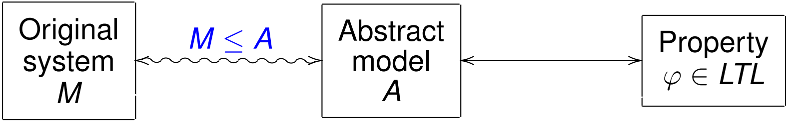
- $\forall s_0 \in S_0 \,.\, \exists s'_0 \in S'_0 \,:\, (s_0, s'_0) \in R$
- $(s, s') \in R \implies L(s) = L'(s')$
- $(s, s') \in R \wedge s \rightarrow p \implies \exists p' \in S' \,:\, s' \rightarrow' p' \wedge (p, p') \in R$

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

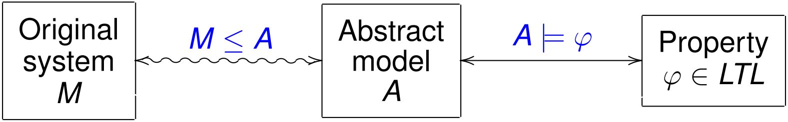
$$L(s_1)L(s_2)\ldots = L'(s'_1)L'(s'_2)\ldots .$$

Relations between original and abstract systems



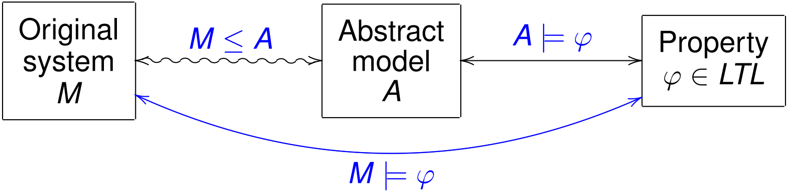
$M \leq A \implies$ all behaviours of M are also in A
(but not vice versa)

Relations between original and abstract systems



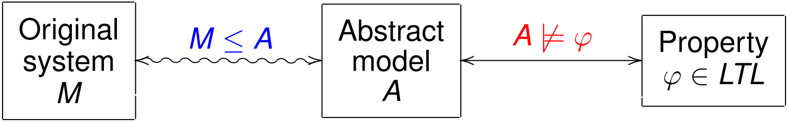
$M \leq A \implies$ all behaviours of M are also in A
(but not vice versa)

Relations between original and abstract systems



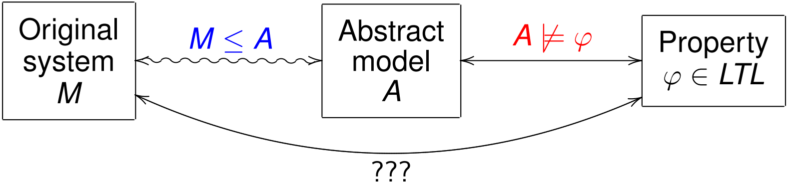
$M \leq A \implies$ all behaviours of M are also in A
(but not vice versa)

Relations between original and abstract systems



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

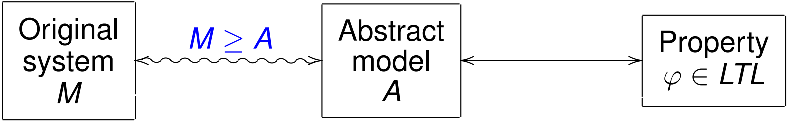
Relations between original and abstract systems



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

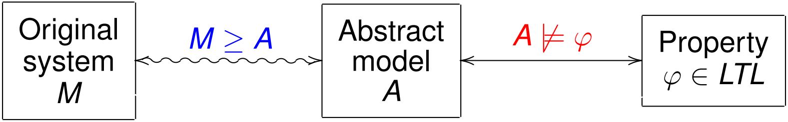
1. M has this behaviour as well (i.e. $M \not\models \varphi$), or
2. M does not have this behaviour, which is then called false positive or spurious counterexample ($M \models \varphi$ or $M \not\models \varphi$ due to another behaviour violating φ).

Relations between original and abstract systems

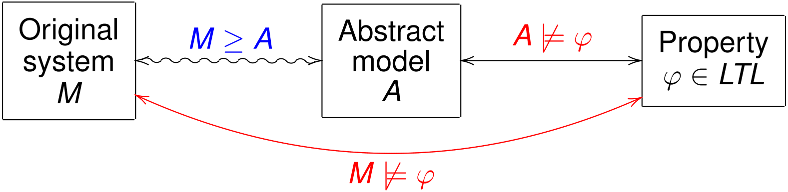


$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

Relations between original and abstract systems

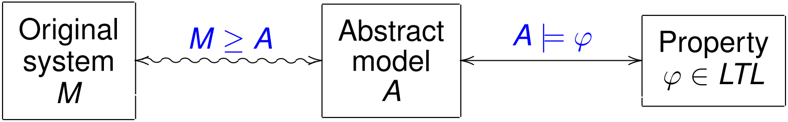


$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)



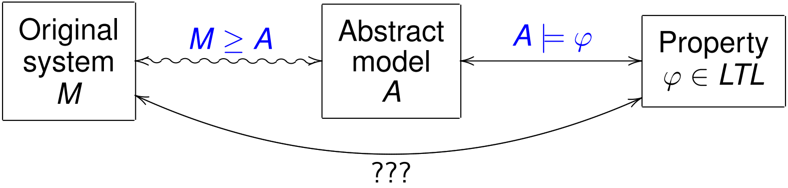
$M \geq A \implies$ all behaviours of A are also in M (but not vice versa)

Relations between original and abstract systems



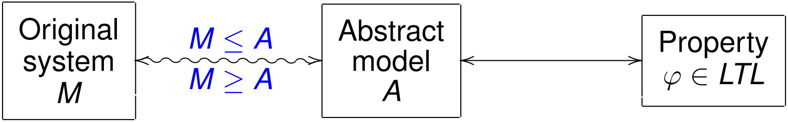
$M \geq A \implies$ all behaviours of A are also in M
(but not vice versa)

Relations between original and abstract systems



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

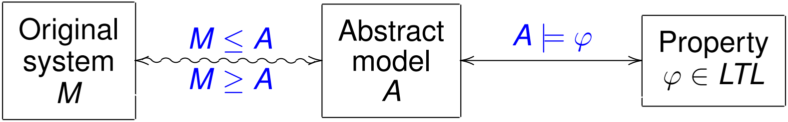
Relations between original and abstract systems



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

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

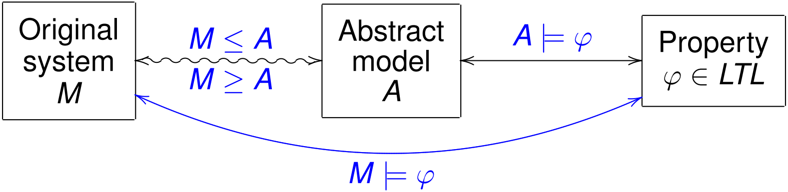
Relations between original and abstract systems



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

Note: A and M are bisimilar $\implies M \leq A \leq M$
$\not\Leftarrow$

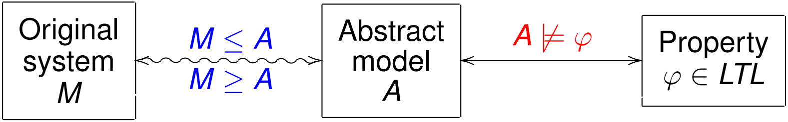
Relations between original and abstract systems



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

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

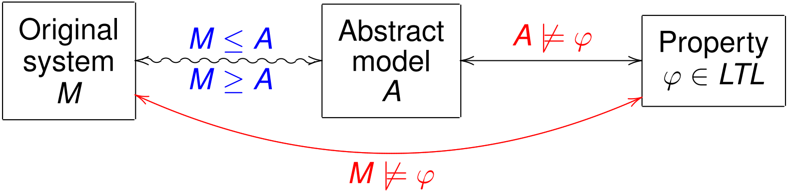
Relations between original and abstract systems



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

Note: A and M are bisimilar $\implies M \leq A \leq M$
$\not\Leftarrow$

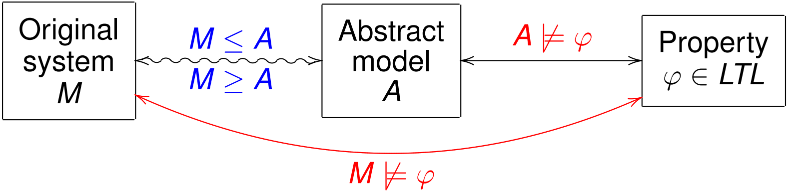
Relations between original and abstract systems



$M \leq A \leq M \implies$ 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



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

Exact abstractions

Cone of influence (aka dead variables)

Idea

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

Cone of influence (aka dead variables)

- 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 do
            z := z * x;
            x := x − 1;
            y := y * v;
            v := v − 1;
    z := z * y;
E:
```

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

Cone of influence: example

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

Specification: F($pc = E$)

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

Cone of influence: example

```
S:  v := getinput();
    x := getinput();
    y := 1;
    z := 1;
    while v > 0 do
            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 do
            skip;
            skip;
            skip;
            v := v − 1;
    skip;
E:
```

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

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

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

Concept

We face two problems

1. to find a suitable abstract domain (i.e. a set of abstract states) 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 domain A_x and we define a total function $h_x : D_x \to A_x$

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

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

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

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

- $h_x(v) = v \pmod{m}$ for some m
- nice properties:

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

Representation by logarithm

- $h_x(v) = \lceil \log_2(v + 1) \rceil$
- the number of bits needed for representation of 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 $\Phi = \{\phi_1, \ldots, \phi_n\}$ be a set of predicates over the set of variables.

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

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

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

- example: $\phi_1 = (x_1 > 3) \quad \phi_2 = (x_1 < x_2) \quad \phi_3 = (x_2 > 10)$
  $s = (5, 7)$
  $h(s) = (1, 1, 0)$
- not used in practice (too many transitions) $\implies$ it is better to assign a single abstract state to a set of original states

Predicate abstraction: abstracting sets of states

- let $\vec{b} = \langle b_1, \ldots, b_n \rangle$ be a vector of $b_i \in \{0, 1, *\}$
- we set $[\vec{b}, \Phi] = b_1 \cdot \phi_1 \wedge \ldots \wedge b_n \cdot \phi_n$,
  where $0 \cdot \phi_i = \neg\phi_i$, $1 \cdot \phi_i = \phi_i$, and $* \cdot \phi_i = \top$
- let X denotes the set of original states

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

- $h(X) = \{\vec{b} \in \{0, 1\}^n \mid \exists s \in X : s \models [\vec{b}, \Phi]\}$
- 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)

- $h(X) = \min\{\vec{b} \mid \forall s \in X : s \models [\vec{b}, \Phi]\}$,
  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 used in practice

Abstract structures

Assume that

- we have a Kripke structure $M = (S, \to, S_0, L)$
- we have an abstract domain A and a mapping $h : S \to A$
- it holds that $A = \{L(s) \mid s \in S\}$ and $L = h$

To achieve the last condition, we set AP to contain only

1. abstraction based on variable domains
   an atomic proposition $(x = a)$ for each $a \in A_x$
2. predicate abstraction
   an atomic proposition (ϕ_i holds) for every ϕ_i

This abstraction is useful if and only if each abstract state determines validity of $AP(\varphi)$.

Abstract structures

Assume that

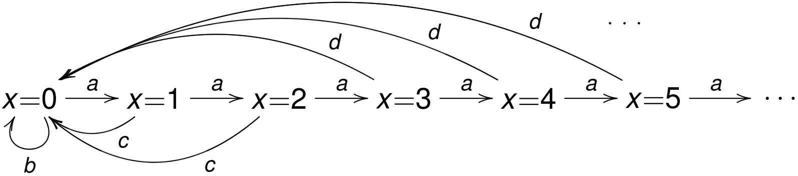
- we have a Kripke structure $M = (S, \rightarrow, S_0, L)$
- we have an abstract domain A and a mapping $h : S \rightarrow A$
- it holds that $A = \{L(s) \mid s \in S\}$ and $L = h$

We define two abstract models:

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

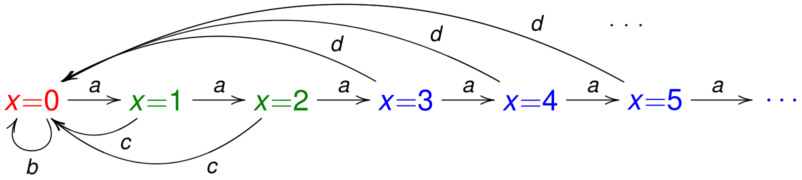
- $A_0 = \{L(s_0) \mid s_0 \in S_0\}$
- $L_A : A \rightarrow A$ such that $L_A(a) = a$
- $a_1 \rightarrow_{may} a_2$ iff there exist $s_1, s_2 \in S$ such that $L(s_1) = a_1$, $L(s_2) = a_2$, and $s_1 \rightarrow s_2$

Example M_{may}

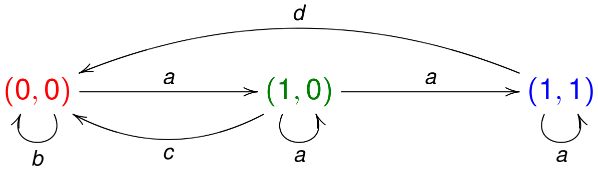


M_{may} with abstract domain $\{0, 1\}^2$ generated by predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$.

Example M_{may}



M_{may} with abstract domain $\{0, 1\}^2$ generated by predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$.



Abstract structures

Assume that

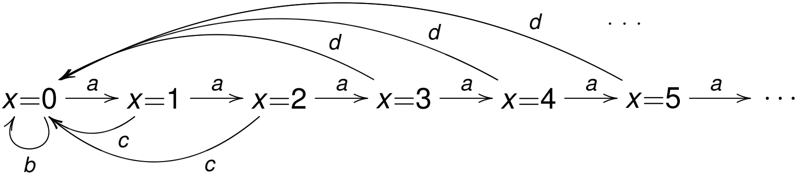
- we have a Kripke structure $M = (S, \rightarrow, S_0, L)$
- we have an abstract domain A and a mapping $h : S \rightarrow A$
- it holds that $A = \{L(s) \mid s \in S\}$ and $L = h$

We define two abstract models:

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

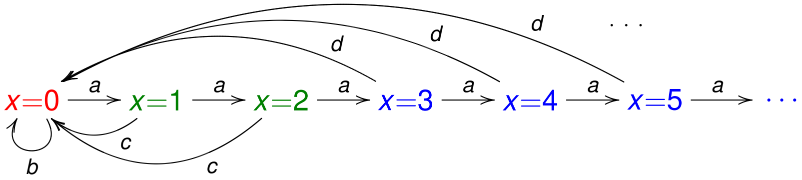
- $A_0 = \{L(s_0) \mid s_0 \in S_0\}$
- $L_A : A \rightarrow A$ such that $L_A(a) = a$
- $a_1 \rightarrow_{must} a_2$ iff for each $s_1 \in S$ satisfying $L(s_1) = a_1$ there exists $s_2 \in S$ such that $L(s_2) = a_2$ and $s_1 \rightarrow s_2$

Example M_{must}

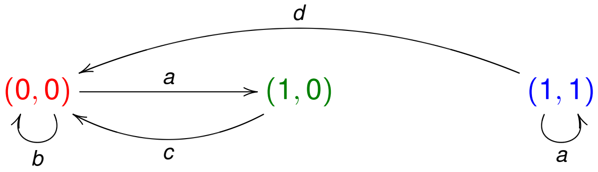


M_{must} with abstract domain $\{0,1\}^2$ generated by predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$.

Example M_{must}



M_{must} with abstract domain $\{0, 1\}^2$ generated by predicate abstraction with predicates $\phi_1 = (x > 0)$ and $\phi_2 = (x > 2)$.



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

Abstraction in practice

Guarded command language

Syntax

- let V be a finite set of integer variable
- expressions over V use standard boolean $(=, <, >)$ and binary $(+, -, \cdot, \ldots)$ operations
- *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 boolean expression over V
  - u_i is a sequence of assignments over V

Guarded command language

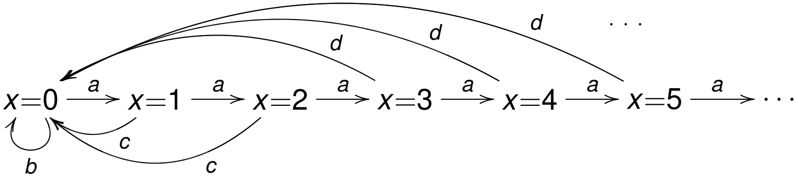
Syntax

- let V be a finite set of integer variable
- expressions over V use standard boolean $(=, <, >)$ and binary $(+, -, \cdot, \ldots)$ operations
- *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 boolean expression over V
  - u_i is a sequence of assignments over V

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 $s' = u_i(s)$

Example



implicit description in guarded command language:

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

Abstraction in practice

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

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

- we use a sound decision procedure *is_valid*, i.e.

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

(the procedure *is_valid* does not have to be complete)

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

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

Example

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

$$(a,\ \top,\ x := x + 1)$$

using the predicates $\phi_1 = (x > 0)$, $\phi_2 = (x > 2)$, we compute the transition

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

Example

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

$$(a,\ \top,\ x := x + 1)$$

using the predicates $\phi_1 = (x > 0)$, $\phi_2 = (x > 2)$, we compute the transition

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

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

Example

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

$$(a,\ \top,\ x := x + 1)$$

using the predicates $\phi_1 = (x > 0)$, $\phi_2 = (x > 2)$, we compute the transition

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

- $(x > 0) \wedge (x \leq 2) \implies (\top \implies (x + 1 > 0))$ 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
- boolean program is an implicit representation of an over-approximation of M_{may}
- it uses only boolean variables $\vec{b}$ representing the validity of abstraction predicates Φ
- boolean program can be used as an input for a suitable model checker (of finite-state systems)

Example

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

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

$V = \{b_1, b_2\}$, where b_1, b_2 represents validity of ϕ_1, ϕ_2

$$
\begin{array}{ll}
(a,\ \top, & b_1 := \mathit{if}\ b_1\ \mathit{then}\ 1\ \mathit{else}\ * \\
 & b_2 := \mathit{if}\ b_2\ \mathit{then}\ 1\ \mathit{else\ if}\ b_1\ \mathit{then}\ *\ \mathit{else}\ 0) \\
(b,\ \neg b_1, & b_1 := 0,\ b_2 := 0) \\
(c,\ b_1 \wedge \neg b_2, & b_1 := 0,\ b_2 := 0) \\
(d,\ b_2, & b_1 := 0,\ b_2 := 0)
\end{array}
$$

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

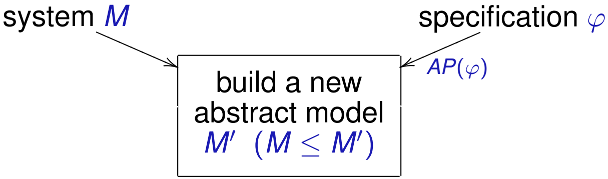
Motivation

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

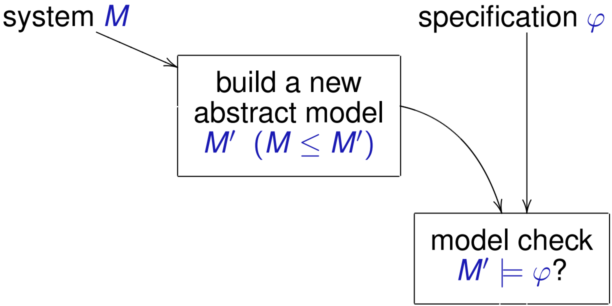
Principle

system M specification φ

Principle

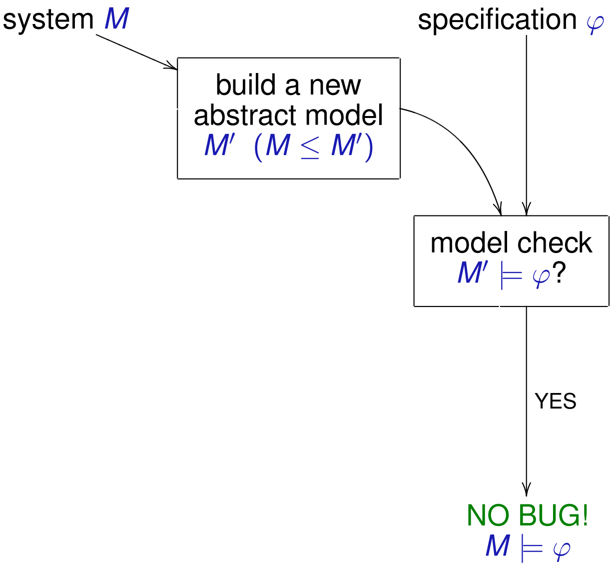

system M
specification φ
$AP(\varphi)$
build a new abstract model M' $(M \leq M')$

Principle

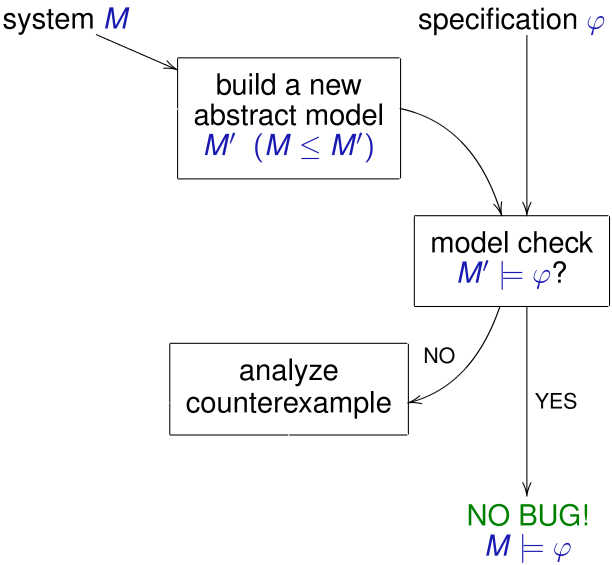


system M
specification φ
build a new abstract model M' $(M \leq M')$
model check $M' \models \varphi$?

Principle

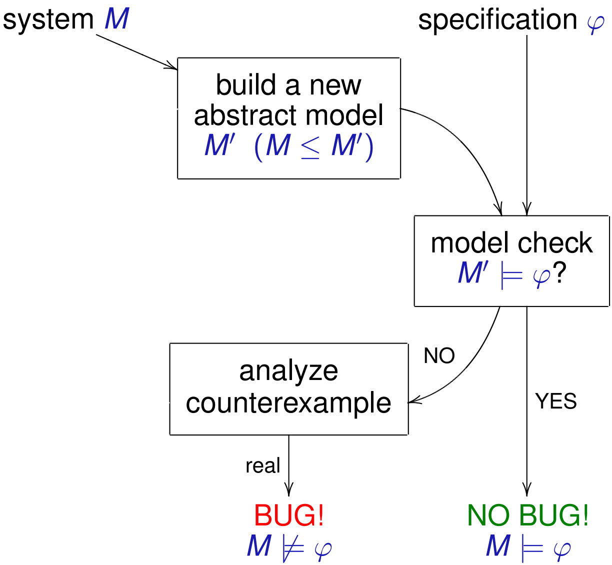

system M
specification φ
build a new abstract model M' $(M \leq M')$
model check $M' \models \varphi$?
YES
NO BUG!
$M \models \varphi$

Principle

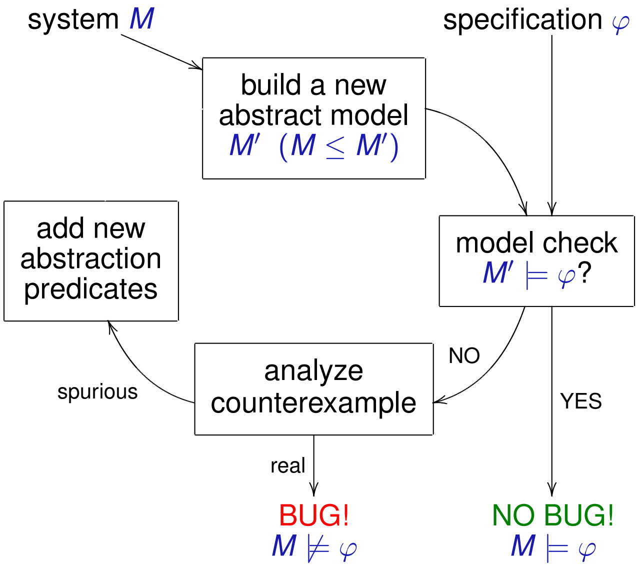


system M
specification φ
build a new abstract model M' $(M \leq M')$
model check $M' \models \varphi$?
NO
analyze counterexample
YES
NO BUG!
$M \models \varphi$

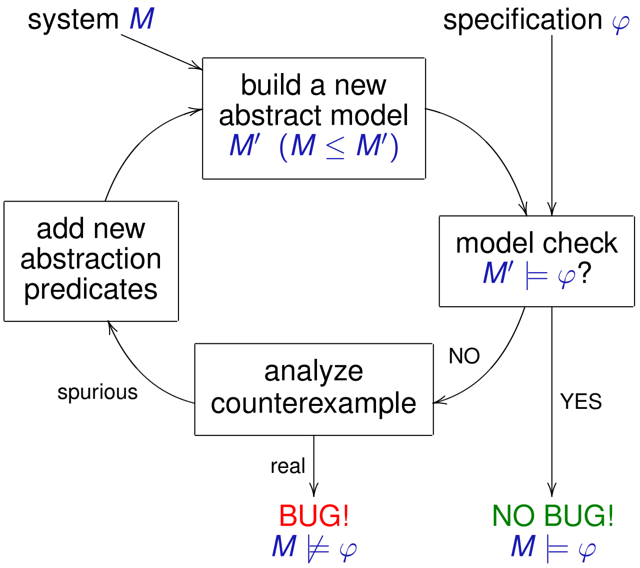
Principle


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

Principle


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

Principle



system M
specification φ
build a new abstract model M' $(M \leq M')$
model check $M' \models \varphi$?
NO
YES
analyze counterexample
spurious
add new abstraction predicates
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)

Coming next week

Symbolic execution

- Can we perform more executions simultaneously?
- Can we perform all possible executions?
- Are there any modern applications of symbolic execution?