Software Verification

Lecture 12: Software Model Checking

Carlo A. Furia
Program Verification: the very idea

P: a program

max (a, b: INTEGER): INTEGER is
  do
    if a > b then
      Result := a
    else
      Result := b
  end
end

S: a specification

require
  True
ensure
  Result >= a
  Result >= b

Does \( P \models S \) hold?

The Program Verification problem:

- **Given**: a program \( P \) and a specification \( S \)
- **Determine**: if every execution of \( P \), for any value of input arguments, satisfies \( S \)
Verification of Finite-State Program

P: a program  S: a specification

Does \( P \models S \) hold?

The Program Verification problem is decidable if \( P \) is finite-state

- With Model-checking techniques

But real programs are not finite-state

- arbitrarily complex inputs
- dynamic memory allocation
- ...


Software Model-Checking: the Very Idea

The term Software Model-Checking denotes techniques to automatically verify real programs based on finite-state models of them. It is a convergence of verification techniques developed during the late 1990's.

The term “software model checker” is probably a misnomer [...] We retain the term solely to reflect historical development.

-- R. Jhala & R. Majumdar: “Software Model Checking”
ACM CSUR, October 2009
Abstraction/Refinement Software M.-C.

Software Model-Checking based on CEGAR: Counterexample-Guided Abstraction/Refinement

- A popular framework for software model-checking

Integrates three fundamental techniques:

- Predicate abstraction of programs
- Detection of spurious counterexamples
- Refinement by predicate discovery
The Big Picture
CEGAR Software Model Checking

ABSTRACT PROGRAM

CONCRETE PROGRAM

(increasing) abstraction
CEGAR Software Model Checking

**Abstract Program**
- PROVE correct (model-check)

**Concrete Program**
- TEST (execute)

(increasing) abstraction
CEGAR Software Model Checking

- **ABSTRACT PROGRAM**
  - PROVE correct (model-check)

- **CONCRETE PROGRAM**
  - TEST (execute)

**REFINE**

(increasing) abstraction
CEGAR Software Model Checking

ABSTRACT PROGRAM

MODEL CHECK

CONCRETE PROGRAM

(increasing) abstraction
CEGAR Software Model Checking

verification fails: COUNTEREXAMPLE

execute COUNTEREXAMPLE

CONCRETE PROGRAM

COUNTEREXAMPLE not executable

(increasing) abstraction
CEGAR Software Model Checking

verification fails: COUNTEREXAMPLE

execute COUNTEREXAMPLE

REFINE abstraction

COUNTEREXAMPLE not executable

(increasing) abstraction
CEGAR Software Model Checking

REFINE abstraction

ABSTRACT PROGRAM

REFINE

CONCRETE PROGRAM

(increasing) abstraction
CEGAR Software Model Checking

ABSTRACT PROGRAM

CONCRETE PROGRAM

START OVER with new abstraction

(increasing) abstraction
Outcome 1: Successful Verification

proof SUCCEEDS: PROGRAM is VERIFIED

ABSTRACT PROGRAM

MODEL CHECK

CONCRETE PROGRAM
Outcome 2: Real Bug Found

ABSTRACT PROGRAM

MODEL CHECK

verification fails: COUNTEREXAMPLE

execute COUNTEREXAMPLE

counterexample: REAL BUG

CONCRETE PROGRAM
Outcome 3: Loop Forever

verification fails: COUNTEREXAMPLE

execute COUNTEREXAMPLE

REFINE abstraction

ABSTRACT PROGRAM

MODEL CHECK

CONCRETE PROGRAM

COUNTEREXAMPLE not executable
Integrates three fundamental techniques:

- Predicate abstraction of programs
- Detection of spurious counterexamples
- Refinement by predicate discovery

Let us now present these techniques in some detail.
Technical premises: weakest preconditions of assertion instructions and parallel conditional assignments
Assertsions and assumptions

For a straightforward presentation of the techniques, we introduce two distinct forms of annotations in the programming language.

- **Assumptions** describe postulated properties of every run reaching the annotation.
  
  ```
  assume exp end
  ```

  - A run reaching an assumption that evaluates to False is infeasible.

- **Assertions** describe properties that every run continuing after the annotation is required to have.
  
  ```
  assert exp end
  ```

  - A run reaching an assertion that evaluates to False terminates with an error.
Assertions and assumptions

The weakest precondition of assertions and assumptions is computed with the following rules.

- \{ \text{exp} \Rightarrow Q \} \text{assume exp end} \{ Q \}
- \{ \text{exp} \land Q \} \text{assert exp end} \{ Q \}

We will not use annotations directly in source programs, but only to build transformations into predicate abstractions and to describe program runs.

Sometimes, we will denote assertions or assumptions with brackets:

\[[\text{exp}]\]
Parallel assignments

For a straightforward presentation of the techniques in the following, we also introduce the parallel assignment:

\[ v_1, v_2, \ldots, v_m := e_1, e_2, \ldots, e_m \]

- First, all the expressions \( e_1, e_2, \ldots, e_m \) are evaluated on the pre state.
- Then, the computed values are orderly assigned to the variables \( v_1, v_2, \ldots, v_m \).

Example:

\[
\begin{align*}
\{ x = 3, y = 1 \} & \quad x := y ; y := x \quad \{ x = 2, y = \} \\
\{ x = 3, y = 1 \} & \quad x, y := y, x \quad \{ x = 2, y = 3 \}
\end{align*}
\]
Parallel assignments

For a straightforward presentation of the techniques, we also introduce the parallel assignment:

\[ v_1, v_2, ..., v_m := e_1, e_2, ..., e_m \]

- First, all the expressions \( e_1, e_2, ..., e_m \) are evaluated on the pre state.
- Then, the computed values are orderly assigned to the variables \( v_1, v_2, ..., v_m \).

Example:

\[
\begin{align*}
\{ x = 3, y = 1 \} & \quad x := y ; y := x & \quad \{ x = 1 , y = 1 \} \\
\{ x = 3, y = 1 \} & \quad x, y := y, x & \quad \{ x = 1 , y = 3 \}
\end{align*}
\]
Parallel conditional assignment

- The parallel assignment and the conditional can be combined into a parallel conditional assignment:

  \[ \text{if } c_1^+ \text{ then } v_1 := e_1^+ \text{ elseif } c_1^- \text{ then } v_1 := e_1^- \text{ else } v_1 := e_1? \text{ end} \]
  \[ \text{if } c_2^+ \text{ then } v_2 := e_2^+ \text{ elseif } c_2^- \text{ then } v_2 := e_2^- \text{ else } v_2 := e_2? \text{ end} \]
  ...  
  \[ \text{if } c_m^+ \text{ then } v_m := e_m^+ \text{ elseif } c_m^- \text{ then } v_m := e_m^- \text{ else } v_m := e_m? \text{ end} \]

- First, evaluate all the conditions (well-formedness requires \( c_k^+ \) and \( c_k^- \) to be mutually exclusive, for all \( k \)).

- Then, evaluate the expressions.

- Finally, perform the assignments.
Predicate Abstraction
Abstraction is a pervasive idea in computer science. It has to do with modeling some crucial (behavioral) aspects while ignoring some other, less relevant, ones.

- **Semantics** of a program $P$: a set of runs $\langle P \rangle$
  - set of all runs of $P$ for any choice of input arguments
  - a run is completely described by a list of program locations that gets executed in order, together with the value that each variable has at the location.

- **Abstraction** of a program $P$: another program $A_P$
  - $A_P$'s semantics is “similar” to $P$'s
    - define some mapping between the runs of $A_P$ and $P$
  - $A_P$ is more amenable to analysis than $P$
Over- and Under-Approximation

Two main kinds of abstraction:

- **over-approximation**: program $AO_P$
  - $AO_P$ allows “more runs” than $P$
  - for every $r \in \langle P \rangle$ there exists a $r' \in \langle AO_P \rangle$
  - intuitively: $\langle P \rangle \subseteq \langle AO_P \rangle$
  - $AO_P$ allows some runs that are “spurious” (also “infeasible”) for $P$

- **under-approximation**: program $AU_P$
  - $AU_P$ allows “fewer runs” than $P$
  - for every $r \in \langle AU_P \rangle$ there exists a $r' \in \langle P \rangle$
  - intuitively: $\langle AU_P \rangle \subseteq \langle P \rangle$
  - $AU_P$ disallows some runs that are “legal” (also “feasible”) for $P$
Over- and Under-Approximation: Example

max (x, y: INTEGER): INTEGER
  do
    if x > y
      then Result := x
    else Result := y
  end
end

AO_max (x, y: INTEGER): INTEGER
  do
    if x > y
      then Result := x
    else assume False end
  end
end

AU_max (x, y: INTEGER): INTEGER
  do
    if x > y
      then Result := x
    end
    if ? then Result := 3 end
  end
end
**Predicate Abstraction**

In **predicate abstraction**, the abstraction $A_P$ of a program $P$ uses only Boolean variables called “predicates”.

- Each predicate captures a significant **fact about the state of $P$**.
- The abstraction $A_P$ is constructed **parametrically w.r.t. a set $\text{pred}$ of chosen predicates** as an **over-approximation** of the program $P$.
  - The **arguments** of $A_P$ are the predicates in $\text{pred}$.
  - Assume arguments are both input and output arguments (this deviates from Eiffel’s standard semantics).
  - Each instruction $\text{inst}$ in $P$ is replaced by a (possibly compound) instruction $\text{inst}'$ in $A_P$ such that:
    - If executing $\text{inst}$ in $P$ leads to a concrete state $S$, then executing $\text{inst}'$ in $A_P$ leads to a state which is the **strongest over-approximation** of $S$ in terms of $\text{pred}$.
Predicate Abstraction: Informal Overview

• Each predicate corresponds to a **Boolean expression**.

• A set of **Boolean program variables in** $A_P$ **track the values** of the predicates in the abstraction.

• Translate each **instruction in** $P$ **into a (compound) instruction which updates the Boolean variables.**

• To have an **over-approximation** the instructions in $A_P$ will:
  
  • define whatever **follows with certainty** from the information given by the predicates
  
  • use **under-approximations** of arbitrary Boolean expressions through the predicates
  
  • everything else is **nondeterministically** chosen
Boolean Predicates and Expressions

Consider a set of predicates
\[ \text{pred} = \{p(1), ..., p(m)\} \]

and a set of corresponding Boolean expressions over program variables
\[ \text{exp} = \{e(1), ..., e(m)\} \]

For a generic Boolean expression \( f \) over program variables, \( \text{Pred}(f) \) denotes the weakest Boolean expression over \( \text{pred} \) that is at least as strong as \( f \) (it implies \( f \), but can be stronger).

- Substituting every atom \( p(i) \) in \( \text{Pred}(f) \) with the corresponding expression \( e(i) \) gives an expression that implies \( f \).
- \( \text{Pred}(f) \) is an under-approximation of \( f \), used to build the strongest over-approximations of instructions.
Boolean Under-Approximation: Example

- \( \text{pred} = \{ p, q, r \} \)
- \( \text{exp} = \{ x = 1, x = 2, x \leq 3 \} \)

- \( \text{Pred}(x = 1) = \)
- \( \text{Pred}(x = 0) = \)
- \( \text{Pred}(x \leq 2) = \)
- \( \text{Pred}(x \neq 0) = \)
**Boolean Under-Approximation: Example**

- $\text{pred} = \{ p, q, r \}$
- $\text{exp} = \{ x = 1, x = 2, x \leq 3 \}$

- $\text{Pred}(x = 1) = p$
- $\text{Pred}(x = 0) = \text{False}$
- $\text{Pred}(x \leq 2) = p \lor q$
- $\text{Pred}(x \neq 0) = p \lor q \lor \neg r$

- In general: $\text{Pred} (\neg f) \neq \neg \text{Pred} (f)$
We want a **weakest under-approximation**:

- Start from the **strongest under-approximation**: False
- Weaken it by adding predicates (negated or unnegated) in disjunction
- (In some cases, you may also try conjunctions of predicates)
- Add as many disjuncts as possible that preserve the under-approximation (i.e., it must always imply the original Boolean expression)
Boolean Under-Approximation: Uniqueness

Pred(f) may not be (syntactically) uniquely defined when predicates imply each other:

- \( \text{pred} = \{ p, q \} \)
- \( \text{exp} = \{ x < 2, x \leq 2 \} \)

\[
\text{Pred}(x \leq 3) = p \lor q \\
\text{equivalent to} = q
\]

- The following transformations are robust w.r.t. the choice of equivalent \( \text{Pred}(f) \).
- When predicates imply each other, however, simplifications are possible (see later), so as a rule we always include all implied facts in \( \text{Pred}(f) \).
Abstraction of Assignments

An assignment: \( x := f \)

is over-approximated by a parallel conditional assignment with \( m \) components. For \( 1 \leq i \leq m \):

\[
\begin{align*}
\text{if} \ Pred(+f(i)) \text{ then } \\
p(i) & := \text{True} \\
\text{elseif} \ Pred(-f(i)) \text{ then } \\
p(i) & := \text{False} \\
\text{else } & p(i) := ? \quad \text{end}
\end{align*}
\]

- \( +f(i) \) is the backward substitution of \( e(i) \) through \( x := f \)
- \( -f(i) \) is the backward substitution of \( \neg e(i) \) through \( x := f \)
Abstraction of Assignments: Example

- \( \text{pred} = \{ p, q, r \} \)
- \( \text{exp} = \{ x > y, \ \text{Result} \geq x, \ \text{Result} \geq y \} \)

- Result := \( x \) is over-approximated by:
  - if \( p \) then \( p := \text{True} \) elseif not \( p \) then \( p := \text{False} \) else \( p := ? \) end
    - which does nothing
  - if \( \text{True} \) then \( q := \text{True} \) elseif \( \text{False} \) then \( q := \text{False} \) else \( q := ? \) end
    - which is equivalent to: \( q := \text{True} \)
  - if \( p \) then \( r := \text{True} \) elseif \( \text{False} \) then \( r := \text{False} \) else \( r := ? \) end
    - which is equivalent to: if \( p \) then \( r := \text{True} \) else \( r := ? \) end
Abstraction of Assignments: Example

- \( \text{pred} = \{ p, q, r \} \)
- \( \text{exp} = \{ x = 1, y = 1, x > y \} \)

\( y := x \)

is over-approximated by

\( q := p ; r := \text{False} \)

\[ \{ x = y \} \]

is over-approximated by

\[ \{ x \leq y \} \cap (\{ x = y = 1 \} \cup \{ x, y \neq 1 \}) \]

or, equivalently,

\[ \{ x \leq y \} \]
Parallel assignments are necessary

The conditional assignments must be executed in parallel to guarantee that the abstraction is sound in general.

Example for: \( p \ (x = \text{True}), \ q \ (x = \text{False}) \)

```plaintext
concrete (x: BOOLEAN) do
  x := not x
end

abstract_ok (p, q: BOOLEAN) do
  p, q := q, p
end

abstract_ko (p, q: BOOLEAN) do
  p := q
  q := p
end
```
Abstraction of Assumptions

An assumption: `assume ex end`

is over-approximated by one assumption:

`assume not Pred(not ex) end`

and a parallel conditional assignment with \( m \) components.

For \( 1 \leq i \leq m \):

- if \( Pred(+ex(i)) \) then
  - \( p(i) := True \)
- elseif \( Pred(-ex(i)) \) then
  - \( p(i) := False \)
- else \( p(i) := ? \)

- \( +ex(i) \) is the backward sub. of \( e(i) \) through `assume ex end`
- \( -ex(i) \) is the backward sub. of \( \neg e(i) \) through `assume ex end`
Abstraction of Assumptions: Example

The double negation is used to get an over-approximation from the under-approximation given by \( \text{Pred} \):

- the complement of an under-approximation of \( x \) is an over-approximation of the complement of \( x \).

- \( \{ p \, (x=1), \, q \, (x=2), \, r \, (x\leq3) \} \)
- \( \text{Pred}(x \leq 2) = p \lor q \)
- \( \text{Pred}(x > 2) = \neg r \)
- assume \( x \leq 2 \) end
- assume \( p \lor q \) end is
- assume \( x=1 \lor x=2 \) end
- assume \( \neg(\neg r) \) end is
- assume \( x \leq 3 \) end
Abstraction of Assumptions: Simplification

Except in the cases where \( \text{ex} \Rightarrow \text{ex}(i) \) or \( \text{ex} \Rightarrow \text{not ex}(i) \) are (unconditionally) valid, the i-th conditional assignment does not have any effect, hence it can be omitted.

In fact:

\[
\begin{align*}
\text{Pred}(\text{+ex}(i)) &= \text{Pred}(\text{not ex} \lor \text{ex}(i)) \\
&= \text{Pred}(\text{not ex}) \lor \text{Pred}(\text{ex}(i)) \\
&= \text{not Pred}(\text{not ex}) \Rightarrow \text{p}(i)
\end{align*}
\]

Which, given the assumption, implies: \( \text{p}(i) \)

\[
\begin{align*}
\text{Pred}(\text{-ex}(i)) &= \text{Pred}(\text{not ex} \lor \text{not ex}(i)) \\
&= \text{Pred}(\text{not ex}) \lor \text{Pred}(\text{not ex}(i)) \\
&= \text{not Pred}(\text{not ex}) \Rightarrow \text{not p}(i)
\end{align*}
\]

Which, given the assumption, implies: \( \text{not p}(i) \)

In all:

\[
\text{if } \text{p}(i) \text{ then } \text{p}(i) := \text{True elseif not p}(i) \text{ then } \text{p}(i) := \text{False else } \text{p}(i) := ? \text{ e end}
\]
Abstraction of Assumptions: Simplification

An assumption: assume ex end

is over-approximated by one simplified assumption:

assume not Pred(not ex) end

where not Pred(not ex) includes:

- a disjunct \( p(i) \) such for every \( i \) such that \( \text{ex} \Rightarrow \text{ex}(i) \) is valid

- a disjunct \( \text{not } p(i) \) such for every \( i \) such that \( \text{ex} \Rightarrow \text{not ex}(i) \) is valid
Abstraction of Assertions

An assertion: \( \text{assert } \text{ex } \text{end} \)
is over-approximated with the same schema as assumptions, namely by one assertion:
\[
\text{assert not Pred(not ex) } \text{end}
\]
and a parallel conditional assignment with \( m \) components. For \( 1 \leq i \leq m \):
\[
\begin{align*}
\text{if Pred(}+\text{ex}(i)) \text{ then} & \\
& p(i) := \text{True} \\
\text{elseif Pred(}+\text{ex}(i)) \text{ then} & \\
& p(i) := \text{False} \\
\text{else } p(i) := ? & \text{ end}
\end{align*}
\]

- \( +\text{ex}(i) \) is the backward sub. of \( e(i) \) through \( \text{assert ex end} \)
- \( -\text{ex}(i) \) is the backward sub. of \( \neg e(i) \) through \( \text{assert ex end} \)
Abstraction of Conditionals

A conditional:

```
if cond then
  -- then branch
else
  -- else branch
end
```

is over-approximated by first transforming it into normal form:

```
if ? then
  assume cond end
  -- then branch
else
  assume not cond end
  -- else branch
end
```

and then applying the other transformations.
Abstraction of Loops

A loop:

from
    -- initialization
until cond loop
    -- loop body
end

is over-approximated by first transforming it into normal form:
from
    -- initialization
until ? loop
    assume not cond end
    -- loop body
end
    assume cond end

and then applying the other transformations.
Abstractions of pre and postconditions

Preconditions are treated as assume instructions and postconditions as assert instructions.

(In abstracting the postcondition, the if instructions can be omitted).

In all our examples we will always choose predicates which completely describe the pre and postcondition, hence no abstraction will be introduced there.
Predicate Abstraction: Example

$max (x, y: INTEGER): INTEGER do$

    if $x > y$ then
        Result := $x$
    else
        Result := $y$
    end

ensure Result $\geq x$ and Result $\geq y$ end

Predicates:

- $p$: $x > y$
- $q$: Result $\geq x$
- $r$: Result $\geq y$

$Apqr_max (p, q, r: BOOLEAN) do$

    if $? \text{ then}$
        assume $x > y$ end ; Result := $x$
    else
        assume $x \leq y$ end ; Result := $y$
    end

ensure Result $\geq x$ and Result $\geq y$ end
Predicate Abstraction: Example

Predicates:

- $p$: $x > y$
- $q$: $\text{Result} \geq x$
- $r$: $\text{Result} \geq y$

Apqr_max (p, q, r: BOOLEAN) do

if ? then
    assume p end
    Result := x
else
    assume not p end
    Result := y
end

ensure q and r end
Predicate Abstraction: Example

Predicates:

- \( p: x > y \)
- \( q: \text{Result} \geq x \)
- \( r: \text{Result} \geq y \)

\[
\text{Apqr\_max}(p, q, r: \text{BOOLEAN}) \text{ do}
\]

\[
\text{if } ? \text{ then}
\]

\[
\text{assume } p \text{ end}
\]

\[
q := \text{True}
\]

\[
\text{if } p \text{ then } r := \text{True} \text{ else } r := ? \text{ end}
\]

\[
\text{else}
\]

\[
\text{assume not } p \text{ end}
\]

\[
\text{Result} := y
\]

\[
\text{end}
\]

\[
\text{ensure } q \text{ and } r \text{ end}
\]
Predicate Abstraction: Example

Predicates:

- p: \( x > y \)
- q: Result \( \geq x \)
- r: Result \( \geq y \)

```
Apqr_max (p, q, r: BOOLEAN) do
    if ? then
        assume p end
        q := True
        if p then r := True else r := ? end
    else
        assume not p end
        r := True
        if not p then q := True else q := ? end
    end
ensure q and r end
```
Predicate Abstraction: Example

Predicates:

- \( p: x > y \)
- \( q: Result \geq x \)
- \( r: Result \geq y \)

```plaintext
Apqr_max (p, q, r: BOOLEAN) do

    if ? then
        assume p end
        q := True
        r := True
    end

    else
        assume not p end
        r := True
        q := True
    end

ensure q and r end
```
Predicate Abstraction: Example

max (x, y: INTEGER): INTEGER do
    if x > y then
        Result := x
    else
        Result := y
    end
ensure Result ≥ x and Result ≥ y end

Apqr_max (p, q, r: BOOLEAN) do
    if p then
        q := True ; r := True
    else
        r := True ; q := True
    end
ensure q and r end

Predicates:
- p: x > y
- q: Result ≥ x
- r: Result ≥ y
What does it mean to verify the predicate abstraction $A_P$ of a program $P$?

- $A_P$ is finite state
  - verification is decidable: we can verify $A_P$ automatically
- $A_P$ is an over-approximation of $P$
  - if $A_P$ is correct then so is $P$
    - any run of $P$ is abstracted by some run of $A_P$
    - if $A_P$ is not correct we can't conclude about the correctness of $P$
  - a counterexample run of $A_P$: the abstract counterexample $r$
    - if $r$ is also the abstraction of some run of $P$ then $P$ is also not correct
    - if $r$ is a run which infeasible for $P$ then $r$ is a spurious counterexample
Model-checking a Boolean Program

For a Boolean program $P$ over predicates $\text{pred} = \{p(1), \ldots, p(m)\}$

- $P$'s body: a sequence $\text{loc} = [L(1), \ldots, L(n)]$ of instructions or conditional jumps
- $P$'s postcondition: $\text{post}$

Build an $\text{FSA} = [\Sigma, S, I, \rho, F]$ where:

- $\Sigma = \text{loc}$
- $S = \{\text{True}, \text{False}\}^m \times (\text{loc} \cup \{\text{halt}\})$
  - each state in $S$ denotes a program state:
    - a truth value for every Boolean variable in $\text{pred}$
    - a program location which represents the next line to be executed, or $\text{halt}$ if the execution has terminated
- $I = \{[v(1), \ldots, v(m), L(1)] \in S\}$
  - the initial states are for any value of the input Boolean arguments
  - $L(1)$ is the next instruction to be executed
- $[v'(1), \ldots, v'(m), L'] \in \rho ([v(1), \ldots, v(m), L], L)$ iff one of the following holds:
  - $L$ is a conditional jump and: $[v(1), \ldots, v(m)]$ satisfies the condition; $v'(i) = v(i)$ for all $1 \leq i \leq m$; $L'$ is the target of the jump when successful.
  - $L$ is a conditional jump and: $[v(1), \ldots, v(m)]$ does not satisfy the condition; and $v'(i) = v(i)$ for all $1 \leq i \leq m$; $L'$ is the target of the jump when unsuccessful.
  - $L$ is an instruction and: $[v'(1), \ldots, v'(m)]$ is the state resulting from executing $L$ on state $[v(1), \ldots, v(m)]$; and $L'$ is the successor of $L$ (or $\text{halt}$ if the program halts after executing $L$)

- $F = \{[v(1), \ldots, v(m), \text{halt}] \in S \mid \text{post does not hold for } [v(1), \ldots, v(m)]\}$
  - error states: halting states where the postcondition doesn't hold
Predicate Abstraction: Example

**Apqr max (p, q, r: BOOLEAN) do**

1: if p
2: then q := True
3: r := True
4: else r := True
5: q := True

**end**

**ensure q and r end**
Predicate Abstraction: Example

\[ \text{Apqr\_max (p, q, r: BOOLEAN) do} \]

1: if p
2: then q := True
3: r := True
4: else r := True
5: q := True
end

ensure q and r end

- Error states: including predicates \( \neg q \) or \( \neg r \) without outgoing edges
- There are clearly no accepting (error) runs because the error states are not even connected
- Apqr\_max is correct and so is max
Detection of Spurious Counterexamples
Predicate Abstraction and Verification

What does it mean to verify the predicate abstraction $A_P$ of a program $P$?

$A_P$ is an over-approximation of $P$

- if $A_P$ is not correct we can't conclude about the correctness of $P$
- a counterexample run of $A_P$: the abstract counterexample $r$
  - if $r$ is also the abstraction of some run of $P$ then $P$ is also not correct
  - if $r$ is a run which infeasible for $P$ then $r$ is a spurious counterexample

Let us show an automated technique to detect spurious counterexamples.
Abstract Counterexamples

Consider an abstract counterexample (c.e.), i.e. a run of the finite-state predicate abstraction $A_P$

\[
\begin{align*}
\{ & \text{Pred}(0) \} \\
\text{inst}(1) & \\
\{ & \text{Pred}(1) \} \\
\text{inst}(2) & \\
\ldots & \\
\text{inst}(N) & \\
\{ & \text{Pred}(N) \} \\
\end{align*}
\]

\[
\begin{align*}
\{ & \text{Abstract initial state} \} \\
\text{Instruction or test} & \\
\{ & \text{Abstract state} \} \\
\text{Instruction or test} & \\
\ldots & \\
\text{Instruction or test} & \\
\{ & \text{Abstract final state} \} \\
\end{align*}
\]

**Goal**: find whether there exists a concrete run of $P$ which is abstracted by this abstract counterexample.
Abstract Counterexamples: Example

\[ \text{max} \ (x, y: \text{INTEGER}): \text{INTEGER} \text{ do} \]
  \[ \text{if } x > y \text{ then} \]
  \[ \quad \text{Result} := x \]
  \[ \text{else} \]
  \[ \quad \text{Result} := y \]
  \[ \text{end} \]
\[ \text{ensure Result} \geq x \text{ and Result} \geq y \text{ end} \]

Predicates:

- \( q: \text{Result} \geq x \)
- \( r: \text{Result} \geq y \)

\[ \text{Aqr}_{\text{max}} \ (q, r: \text{BOOLEAN}) \text{ do} \]
  \[ \text{if } ? \text{ then} \]
  \[ \quad q := \text{True} ; r := ? \]
  \[ \text{else} \]
  \[ \quad r := \text{True} ; q := ? \]
  \[ \text{end} \]
\[ \text{ensure } q \text{ and } r \text{ end} \]
Abstract Counterexamples: Example

Aqr_max (q, r: BOOLEAN) do
    if ? then
        q := True; r := ?
    else
        r := True; q := ?
    end
ensure q and r end

- Error states: including ¬q or ¬r and without outgoing edges
- An abstract counterexample trace in green
Concrete Run of Abstract C.E.

Because of how $A_P$ has been built, there exists a instruction in $P$ for every (possibly compound) instruction in $A_P$

Abstract run:

\[
\begin{align*}
\{\text{Pred}(0)\} \\
\text{inst}(1) \\
\{\text{Pred}(1)\} \\
\text{inst}(2) \\
\ldots \\
\text{inst}(N) \\
\{\text{Pred}(N)\}
\end{align*}
\]

Concrete run:

\[
\begin{align*}
\text{Concrete-inst}(1) \\
\text{Concrete-inst}(2) \\
\ldots \\
\text{Concrete-inst}(N)
\end{align*}
\]

Let us check whether the concrete run is infeasible, according to the semantics of $P$. 
Feasibility of a Concrete Run

Compute the weakest precondition of \text{Pred}(N) over the concrete run with conditions (\text{assume}, \text{conditionals}, or exit conditions) interpreted as \text{assert} (this is doable automatically, modulo undecidability of the used logic fragment, because there are no loops in the run):

Abstract run:

\[
\begin{align*}
\{ \text{Pred}(0) \} \\
\text{inst}(1) \\
\{ \text{Pred}(1) \} \\
\text{inst}(2) \\
\text{...} \\
\text{inst}(N) \\
\{ \text{Pred}(N) \}
\end{align*}
\]

Concrete run:

\[
\begin{align*}
\{ \text{WP}(0) \} \\
\text{Concrete-inst}(1) \\
\{ \text{WP}(1) \} \\
\text{Concrete-inst}(2) \\
\text{...} \\
\text{Concrete-inst}(N) \\
\{ \text{Pred}(N) \}
\end{align*}
\]

Every formula \text{WP}(i) characterizes the states of \text{P} reaching a final state where \text{Pred}(N) holds and hence where the postcondition fails.
Feasibility of a Concrete Run

The concrete run is infeasible if WP(i) and Pred(i) is unsatisfiable for some 1 ≤ i ≤ N.

Concrete run:

\{ \text{Pred}(0) \text{ and } \text{WP}(0) \}

Concrete-inst(1)

\{ \text{Pred}(1) \text{ and } \text{WP}(1) \}

Concrete-inst(2)

...

Concrete-inst(N)

\{ \text{Pred}(N) \text{ and } \text{Pred}(N) \}
Spurious Counterexamples: Example

Abstract c.e. trace:

\{q, \neg r\}
[?]
\{q, \neg r\}

\(q := \text{True} ; r := ?\)

\{q, \neg r\}

Concrete trace:

\{x > y \text{ and } x < y\}

assert \(x > y\) end

\{x \geq x \text{ and } x < y\}

Result := x

\{\text{Result} \geq x \text{ and } \text{Result} < y\}

The counterexample is infeasible because:

\{q \text{ and } x > y \text{ and } x < y\} \text{ is inconsistent}
as \{q \text{ and } x > y\} \text{ implies } \{x \geq y\}
Abstract Counterexamples: Example

neg_pow (x, y: INTEGER): INTEGER do
require x < 0 and y > 0
from Result := 1
until y ≤ 0
loop
Result := Result * x
y := y - 1
end
ensure Result > 0 end

Apqr_neg_pow (p, q, r: BOOLEAN) do
require p and q
from r := True
until ~q
loop
if p and r then r := False else r := ? end
q := ?
end
ensure r end

Predicates:
- p: x < 0
- q: y > 0
- r: Result > 0

Predicates:
- \( p: x < 0 \)
- \( q: y > 0 \)
- \( r: Result > 0 \)
**Abstract Counterexamples: Example**

**Predicates:**
- p: \(x < 0\)
- q: \(y > 0\)
- r: Result > 0

**Abstract c.e. trace:**

\[
\{p, q, \neg r\} \\
\quad r := \text{True} \\
\{p, q, r\} \\
\quad [q] \\
\{p, q, r\} \\
\quad [p \text{ and } r] \\
\{p, q, r\} \\
\quad r := \text{False} \\
\{p, q, \neg r\} \\
\quad q := ? \\
\{p, \neg q, \neg r\} \\
\quad [\neg q] \\
\{p, \neg q, \neg r\}
\]
Abstract Counterexamples: Example

Abstract c.e. trace:
\{p, q, \neg r\}
   r := True
\{p, q, r\}
   [q]
\{p, q, r\}
   [p and r]
\{p, q, r\}
   r := False
\{p, q, \neg r\}
   q := ?
\{p, \neg q, \neg r\}
   [\neg q]
\{p, \neg q, \neg r\}

Concrete trace:
\{x < 0 and y = 1\}
   Result := 1
\{x < 0 and y = 1 and Result*x \leq 0\}
   assert y > 0 end
\{x < 0 and y \leq 1 and Result*x \leq 0\}

   Result := Result * x
\{x < 0 and y \leq 1 and Result \leq 0\}
   y := y - 1
\{x < 0 and y \leq 0 and Result \leq 0\}
   assert y \leq 0 end
\{x < 0 and y \leq 0 and Result \leq 0\}
Abstract Counterexamples: Example

Concrete trace:

\{x < 0 \text{ and } y = 1\}
Result := 1
\{x < 0 \text{ and } y = 1 \text{ and } \text{Result} \times x \leq 0\}
assert y > 0 end
\{x < 0 \text{ and } y \leq 1 \text{ and } \text{Result} \times x \leq 0\}

\text{Predicates:}

- \(p: x < 0\)
- \(q: y > 0\)
- \(r: \text{Result} > 0\)

\text{The counterexample is feasible. We have found a real bug in the concrete program occurring for input } y = 1 \text{ (and any } x < 0).
Predicate Discovery and Refinement
A spurious counterexample shows that the used abstraction is too coarse.

We build a finer abstraction by adding new predicates to the set pred.

These new predicates must be chosen so that the spurious counterexample is not allowed in the new abstraction.
Syntax-based Predicate Discovery

The simplest way to find new predicates is syntactic:

Concrete run:

\{ \text{Pred}(0) \text{ and } \text{WP}(0) \} \text{ Concrete-inst}(1) \text{ \{ WP}(0) \} \backslash \text{ \{ Pred}(0) \} \\
\{ \text{Pred}(1) \text{ and } \text{WP}(1) \} \text{ Concrete-inst}(2) \text{ \{ WP}(1) \} \backslash \text{ \{ Pred}(1) \} \\
\ldots \text{ Concrete-inst}(N) \\
\{ \text{Pred}(N) \text{ and } \text{Pred}(N) \} \text{ \{ Pred}(N) \} \backslash \text{ \{ Pred}(N) \} \\

Look for predicates that:

- hold in the concrete run
- are not traced by any predicate in the abstract run
- contradict the predicates in the abstract run
Syntax-based Predicate Discovery: Example

Concrete trace:
\[
\{x > y, \neg r\} \setminus \{q, \neg r\}
\]
assert \(x > y\) end
\[
\{\text{True}, \neg r\} \setminus \{q, \neg r\}
\]
Result := \(x\)
\[
\{q, \neg r\} \setminus \{q, \neg r\}
\]

Predicates:
- \(q\): Result \(\geq x\)
- \(\neg r\): Result \(< y\)

The predicate from the concrete run that is not traced in the abstract run is:
- \(p = x > y\)

Predicate \(p\) contradicts \(\{q, \neg r\}\). It is enough to verify the program with the new abstraction.
Summary, Tools, and Extensions
CEGAR: Summary

- Finite-state predicate abstraction of real programs
  - Static analysis & abstract interpretation
- Automated verification of finite-state programs
  - Model checking of reachability properties
- Detection of spurious counterexamples
  - Axiomatic semantics & automated theorem proving
- Automated counterexample-based refinement
  - Symbolic model-checking techniques
Software Model-Checking Tools

CEGAR software model-checkers

- **SLAM** -- Ball and Rajamani, ~2001
  - first full implementation of CEGAR software m-c
  - used at Microsoft for device driver verification
- **BLAST** -- Henzinger et al., ~2002
  - does lazy abstraction: partial refinement of abstract program
  - several extensions for arrays, recursive routines, etc.
- **Magic** -- Clarke et al., ~2003
  - modular verification of concurrent programs
- **F-Soft** -- Gupta et al., ~2005
  - Combines software model-checking with abstract interpretation techniques
- **CBMC & SATABS** -- Kroening et al., ~2005
  - Use bounded model-checking techniques
Other (non CEGAR) software model-checking tools

- **Verisoft** -- Godefroid et al. ~2001
- **Java PathFinder** -- Visser et al., ~2000
- **Bandera** -- Hatcliff, Dwyers, et al., ~2000
Software Model-Checking: Extensions

- Inter-procedural analysis
- Complex data structures
- Concurrent programs
- Recursive routines
- Heap-based languages
- Termination analysis
- Integration with other verification techniques
  - Static analysis
  - Testing
- ...

None of these directions is exclusive domain of software model-checking, of course...