Software Verification

Lecture 10:
Software Model Checking

Carlo A. Furia
Program Verification: the very idea

Program P:

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

Specification S:

```plaintext
require True
ensure Result >= a
    Result >= b
```

Does P ⊩ 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 parameters, satisfies S
Verification of Finite-State Program

\[ P: \text{a program} \quad S: \text{a specification} \]

Does \[ P \models S \]

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

- Model-checking techniques

But real programs are not finite-state.
Software Model-Checking: the Very Idea

The term *Software Model-Checking* denotes an array of techniques to automatically verify real programs based on *finite-state models* of them.

It is a convergence of verification techniques which started happening 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
Software Model-Checking based on CEGAR: Countereample-Guided Abstraction/Refinement

- A successful framework for software model-checking

Integrates three fundamental techniques:

- Predicate abstraction of programs
- Detection of spurious counterexamples
- Refinement by predicate discovery
The Big Picture
**CounterExample Guided Abstraction Refinement**

ABSTRACT PROGRAM

CONCRETE PROGRAM

(increasing) abstraction
CounterExample Guided Abstraction Refinement

ABSTRACT PROGRAM

CONCRETE PROGRAM

PROVE correct

execute

(increasing) abstraction
CounterExample Guided Abstraction Refinement

ABSTRACT PROGRAM → REFINE → CONCRETE PROGRAM

PROVE correct → execute

(increasing) abstraction
CounterExample Guided Abstraction Refinement

ABSTRACT PROGRAM \rightarrow CONCRETE PROGRAM

MODEL-CHECK

(increasing) abstraction
CouterExample Guided Abstraction Refinement

verification fails: COUNTEREXAMPLE

ABSTRACT PROGRAM

MODEL-CHECK

CONCRETE PROGRAM

(increasing) abstraction
CouterExample Guided Abstraction Refinement

verification fails: COUNTEREXAMPLE is COUNTEREAMPLE executable?

CONCRETE PROGRAM

ABSTRACT PROGRAM

MODEL-CHECK

(increasing) abstraction

COUNTEREREMPLE not executable
Counterexample Guided Abstraction Refinement

verification fails: COUNTEREXAMPLE

is COUNTEREXAMPLE executable?

CONCRETE PROGRAM

COUNTEREXAMPLE not executable

REFINE by ruling out concrete execution

MODEL-CHECK

ABSTRACT PROGRAM

(increasing) abstraction
**CounterExample Guided Abstraction Refinement**

ABSTRACT PROGRAM ➔ CONCRETE PROGRAM

REFINE

(increasing) abstraction
CouterExample Guided Abstraction Refinement

ABSTRACT PROGRAM CONCRETE PROGRAM

(increasing) abstraction
Outcome: Successful Verification

proof SUCCEEDS: PROGRAM is VERIFIED

ABSTRACT PROGRAM  →  CONCRETE PROGRAM

MODEL-CHECK
Outcome: Real Bug Found

verification fails: COUNTEREXAMPLE

is COUNTEREXAMPLE executable?

ABSTRACT PROGRAM

MODEL-CHECK

CONCRETE PROGRAM

COUNTEREXAMPLE executable: REAL BUG
Outcome: Loop Forever

verification fails: COUNTEREXAMPLE

is COUNTEREXAMPLE executable?

MODEL-CHECK

REFINE by ruling out concrete execution

ABSTRACT PROGRAM

CONCRETE PROGRAM

COUNTEREXAMPLE not executable

(increasing) abstraction
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 statements and parallel conditional assignments
Assertions and assumptions

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

- **Assumptions** describe information that every run reaching the statement has.
  
  ```
  assume exp end
  ```
  
  - A run reaching an **assumption** that evaluates to `False` is infeasible.

- **Assertions** describe information that every run continuing after the statement must 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 } \text{exp} \text{ end } \{ Q \}
- \{ \text{exp} \land Q \} \text{ assert } \text{exp} \text{ 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:

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

- 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:

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

For a straightforward presentation of the techniques in the following, 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

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 variables 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

AO_max (x, y: INTEGER): INTEGER
do
  if x > y
    then Result := x
    else Result := y
  end
  if ? then Result := 3 end
end

AU_max (x, y: INTEGER): INTEGER
do
  if x > y
    then Result := x
    else assume False 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 parameters (this deviates from Eiffel's semantics)
  - each statement $\text{stmt}$ in $P$ is replaced by a (possibly compound) statement $\text{stmt}'$ in $A_P$ such that:
    - if executing $\text{stmt}$ in $P$ leads to a concrete state $S$, then executing $\text{stmt}'$ in $A_P$ leads to a state which is the strongest over-approximation of $S$ in terms of $\text{pred}$
Predicate Abstraction: Informal Overview

1. Each predicate corresponds to a Boolean expression.

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

3. Translate each statement in $P$ into a (compound) statement which updates the Boolean variables.

4. To have an over-approximation the statements in $A_P$ will:
   
   a) define whatever follows with certainty from the information given by the predicates
      
      • use under-approximations of arbitrary Boolean expressions through the predicates
   
   b) 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 \( f \).

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

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

- Pred(x = 1) =
- Pred(x = 0) =
- Pred(x \leq 2) =
- 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: \( \text{Pred} (\neg f) \neq \neg \text{Pred} (f) \)
Abstraction of Assignments

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

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

- \( +f(i) \) 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

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

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

- $p$ representing $x = \text{True}$; $q$ representing $x = \text{False}$

```
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: \( \text{assume ex end} \)
is over-approximated by one assumption:
\( \text{assume not Pred(\neg ex) end} \)
and a parallel conditional assignment with \( m \) components.
For \( 1 \leq i \leq m \):
\[
\text{if Pred(+ex(i)) then} \\
\quad \text{p(i) := True} \\
\text{elseif Pred(-ex(i)) then} \\
\quad \text{p(i) := False} \\
\text{else p(i) := ? end}
\]

- \(+\text{ex}(i)\) is the backward sub. of \( e(i) \) through assume \( \text{ex end} \)
- \(-\text{ex}(i)\) is the backward sub. of \( \neg e(i) \) through assume \( \text{ex end} \)
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 $x$.
- $\{ p \ (x=1), \ q \ (x=2), \ r \ (x \leq 3) \}$
- $\text{Pred}(x \leq 2) = p \lor q$
- $\text{Pred}(x > 2) = \neg r$
- 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 Assertions

An assertion: `assert ex end`

is over-approximated with the same schema as assumptions, namely by one assertion:

```
assert 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) := ?
end
```

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

A conditional:

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

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

```plaintext
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 statements and postconditions as assert statements.

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

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

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

Predicates:

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

\[
\text{Apqr\_max} \ (p, q, r: \text{BOOLEAN}) \ do \\
\quad \text{if } ? \text{ then} \\
\quad \quad \text{assume } x > y \text{ end ; Result} := x \\
\quad \text{else} \\
\quad \quad \text{assume } x \leq y \text{ end ; Result} := y \\
\quad \text{end} \\
\text{ensure Result} \geq x \text{ and Result} \geq y \text{ end}
\]
Predicate Abstraction: Example

Predicates:

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

Apqr$_{\text{max}}$ ($p$, $q$, $r$: 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: Result ≥ x
- r: Result ≥ 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
    Result := y
end

ensure q and r end
Predicate Abstraction: Example

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

Predicates:

- p: x > y
- q: Result ≥ x
- r: Result ≥ y
Predicate Abstraction: Example

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

- if ? then
  - assume `p` end
  - `q := True`
  - `r := True`
- else
  - assume not `p` end
  - `r := True`
  - `q := True`

end

`ensure q and r end`
Predicate Abstraction: Example

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

Predicates:

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

\[
\text{Apqr\text{\_}max}(p, q, r: \text{BOOLEAN}) \text{ do }
\begin{align*}
\text{if } p \\
\text{then } q &:= \text{True} \; ; \; r := \text{True} \\
\text{else } r &:= \text{True} \; ; \; q := \text{True}
\end{align*}
\text{ end }
\text{ ensure } q \text{ and } r \text{ end}
\]
Predicate Abstraction and Verification

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 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
    - \( L \) is a conditional jump and:
      - \( [v(1), \ldots, v(m)] \) satisfies the condition; and
      - \( v'(i) = v(i) \) for all \( 1 \leq i \leq m \); and
      - \( L' \) 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 \( 1 \leq i \leq m \); and
      - \( L' \) 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

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

1: if \(p\)
2: then \(q := \text{True}\)
3: \(r := \text{True}\)
4: else \(r := \text{True}\)
5: \(q := \text{True}\)
end

\text{ensure } q \text{ and } r \text{ end}
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**

- **Error states**: including predicates ¬q or ¬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 $r$
  1. if $r$ is also the abstraction of some run of $P$ then $P$ is also not correct
  2. 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 $A_P$

\[
\begin{align*}
\{ \text{Pred}(0) \} & \quad \{ \text{Abstract initial state} \} \\
\text{Stmt}(1) & \quad \text{Instruction or test} \\
\{ \text{Pred}(1) \} & \quad \{ \text{Abstract state} \} \\
\text{Stmt}(2) & \quad \text{Instruction or test} \\
\ldots & \quad \ldots \\
\text{Stmt}(N) & \quad \text{Instruction or test} \\
\{ \text{Pred}(N) \} & \quad \{ \text{Abstract final state} \}
\end{align*}
\]

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

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

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

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

\[ A_{qr\_max} (q, r: BOOLEAN) \]

\[
\begin{align*}
\text{if } ? & \\
\text{then } q := \text{True} ; r := ? & \\
\text{else } r := \text{True} ; q := ? & \\
\text{end &}
\end{align*}
\]

\[
\text{ensure } q \text{ and } r \text{ end}
\]

- Error states: including \( \neg q \) or \( \neg r \) 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 statement in $P$ for every (possibly compound) statement in $A_P$

Abstract run:

```
{ Pred(0) }
Stmt(1)
{ Pred(1) }
Stmt(2)
...
Stmt(N)
{ Pred(N) }
```

Concrete run:

```
Concrete-stmt(1)
Concrete-stmt(2)
...
Concrete-stmt(N)
```

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 True over the concrete run with conditions (assume, conditionals, or exit conditions) interpreted as assert (this is doable automatically because there are no loops):

Abstract run:

Concrete run:

<table>
<thead>
<tr>
<th>Stmt(1)</th>
<th>Concrete-stmt(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stmt(2)</td>
<td>Concrete-stmt(2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Stmt(N)</td>
<td>Concrete-stmt(N)</td>
</tr>
</tbody>
</table>

Every formula WP(i) characterizes the states of P reaching a final state where Pred(N) holds and hence where the postcondition fails.
The concrete run is infeasible if $WP(i)$ and $Pred(i)$ is unsatisfiable for some $1 \leq i \leq N$.

Concrete run:

\[
\begin{align*}
\{ \texttt{Pred}(0) & \quad \text{and} \quad \texttt{WP}(0) \} \\
\text{Concrete-stmt}(1) \\
\{ \texttt{Pred}(1) & \quad \text{and} \quad \texttt{WP}(1) \} \\
\text{Concrete-stmt}(2) \\
\vdots \\
\text{Concrete-stmt}(N) \\
\{ \texttt{Pred}(N) & \quad \text{and} \quad \text{True} \}
\end{align*}
\]
Abstract c.e. trace:
\{q, \neg r\}

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

\textcolor{red}{q := True ; r := ?}

\{q, \neg r\}

Concrete trace:
\{x > y\}

\textcolor{green}{\text{assert } x > y \text{ end}}

\{True\}

\textcolor{blue}{Result := x}

\{True\}

The counterexample is infeasible because:
\{x > y \text{ and } q \text{ and } \neg r\} \text{ is inconsistent}

as \{x > y \text{ and } q\} \text{ implies } \{r\}
The condition for infeasibility is only sufficient:

- If $WP(i)$ and $Pred(i)$ is satisfiable for all $1 \leq i \leq N$, further analysis may be needed, in general, to determine if the run is feasible.

- There are additional techniques to decide feasibility automatically (assuming satisfiability is decidable for the first-order fragment used in the annotations).

- In our examples, we will simply determine by manual inspection if a run that passes the infeasibility test is feasible or not.
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:
- $p: x < 0$
- $q: y > 0$
- $r: \text{Result} > 0$

Apqr_neg_pow \ ((p, q, r: \text{BOOLEAN}) \ do$

  \begin{align*}
  &\text{require } p \text{ and } q \\
  &\text{from } r := \text{True} \\
  &\text{until } \neg q \\
  &\text{loop} \\
  &\quad \text{if } p \text{ and } r \text{ then } r := \text{False} \text{ else } r := ? \text{ end} \\
  &\quad q := ? \\
  &\text{end} \\
  &\text{ensure } r \text{ end}
  \end{align*}

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

Apqr_neg_pow \ ((p, q, r: \text{BOOLEAN}) \ do
**Abstract Counterexamples: Example**

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

**Concrete trace:**
\[
\{y = 1\}
\]
\[\text{Result} := 1\]
\[
\{y = 1\}
\]
\[\text{assert } y > 0 \text{ end}\]
\[
\{y \leq 1\}
\]
\[\text{Result} := \text{Result} \ast x\]
\[
\{y \leq 1\}
\]
\[y := y - 1\]
\[
\{y \leq 0\}
\]
\[\text{assert } y \leq 0 \text{ end}\]
\[
\{\text{True}\}
Abstract Counterexamples: Example

Concrete trace:
{y = 1}
Result := 1
{y = 1}
assert y > 0 end
{y ≤ 1}

Result := Result * x
{y ≤ 1}
y := y - 1
{y ≤ 0}
assert y ≤ 0 end
{True}

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

The counterexample is feasible. We have found a real bug in the concrete program occurring for input y = 1 (and any x < 0).
Predicate Discovery and Refinement
Predicate Discovery

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:

Concrete-stmt(1)

Concrete-stmt(2)

...

Concrete-stmt(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\} \setminus \{q, \neg r\}

\text{assert } x > y \text{ end}

\{\text{True}\} \setminus \{q, \neg r\}

\text{Result := x}

\{\text{True}\} \setminus \{q, \neg r\}

Predicates:

- \text{q: Result }\geq x
- \text{\neg r: Result }< y

The predicate from the concrete run that is not traced in the abstract run is:

- \text{p = x > y}

Predicate \text{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
Software Model-Checking Tools

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...