Software Verification

Lecture 12: Software Model Checking

Carlo A. Furia

Program Verification: the very idea

P: a program

S: a specification

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

```
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 $P \vDash S$ hold?

The Program Verification problem is decidable if P is finite-state

– 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 an array of 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

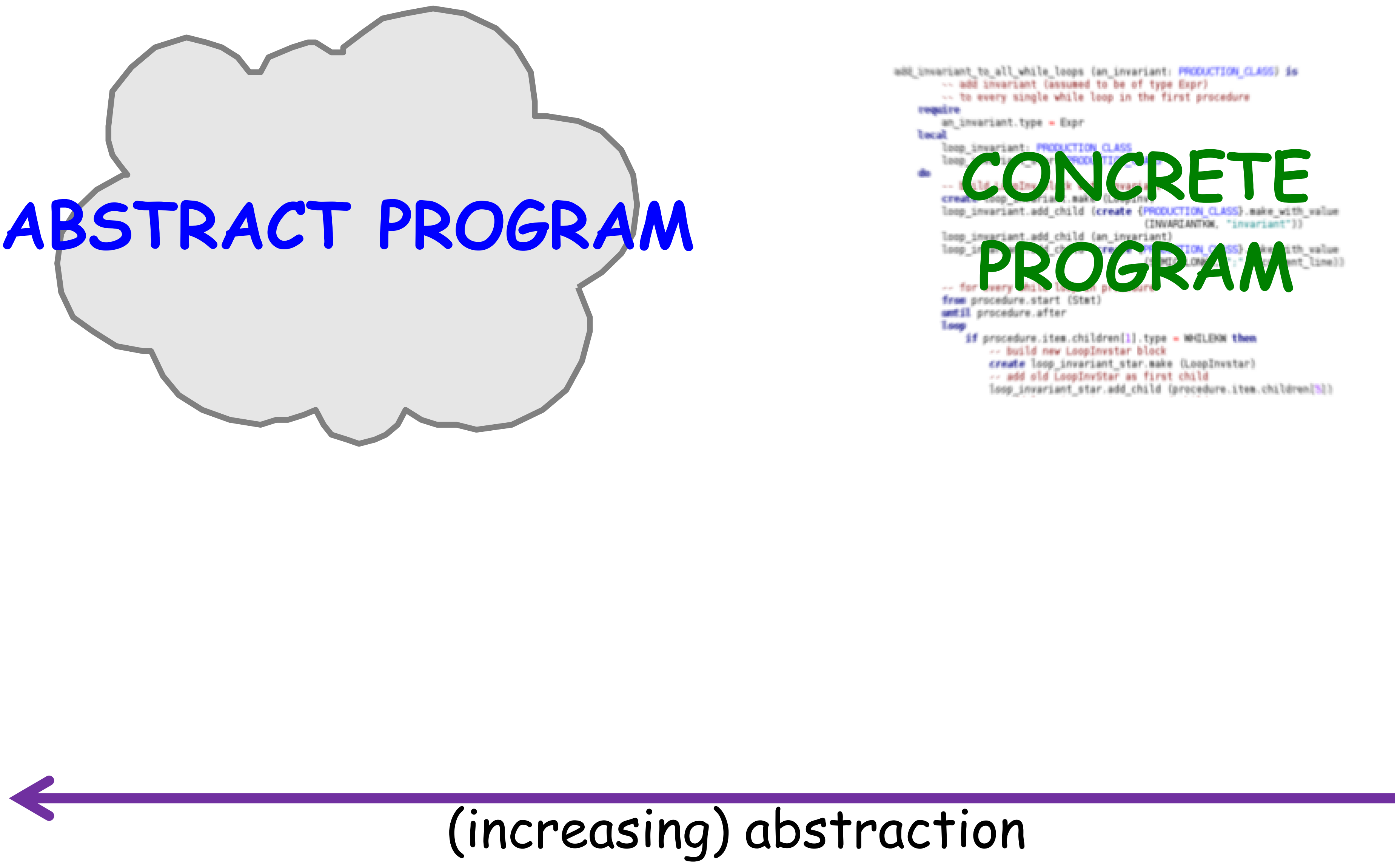
- A popular framework for software model-checking

Integrates three fundamental techniques:

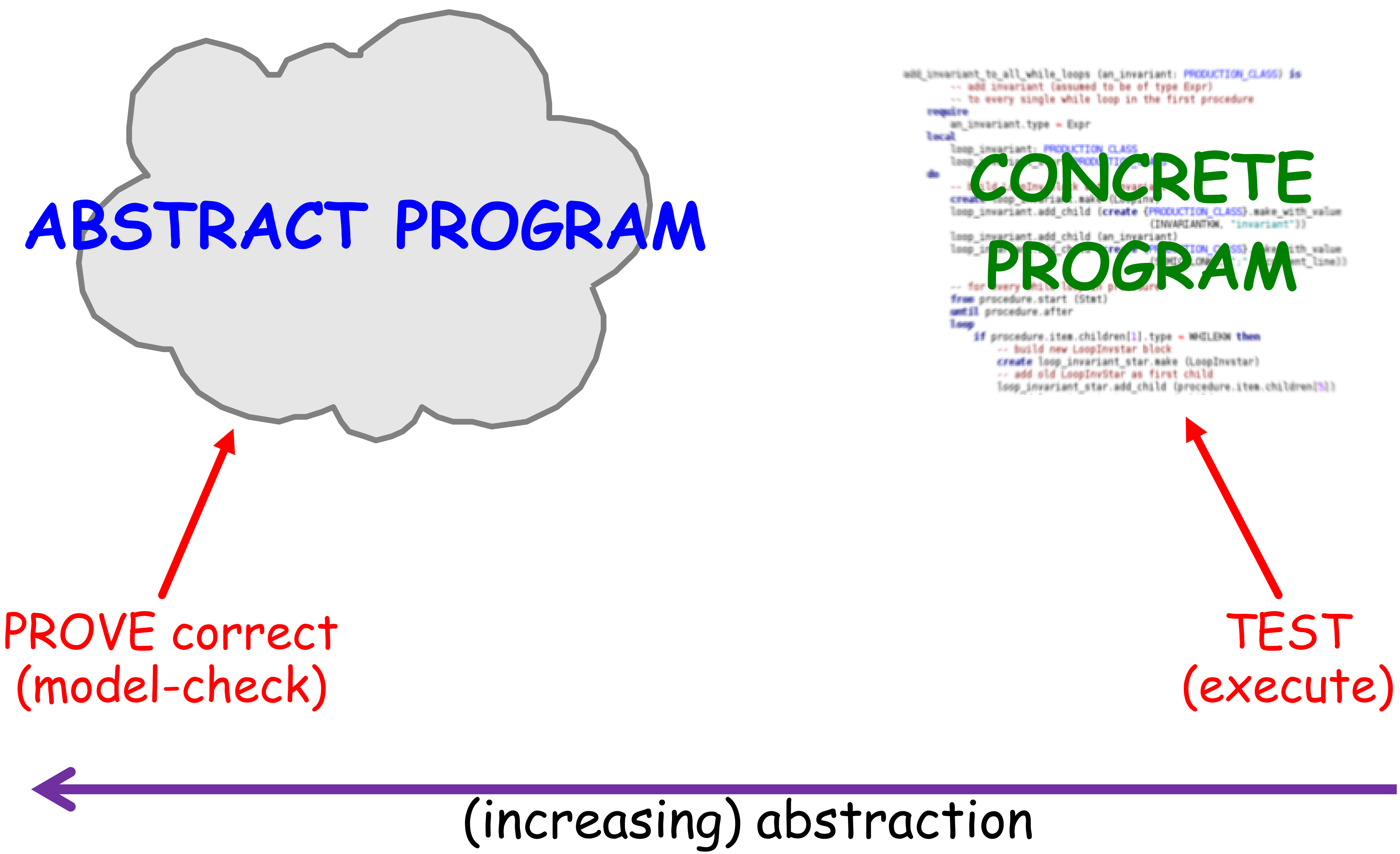
- Predicate abstraction of programs
- Detection of spurious counterexamples
- Refinement by predicate discovery

The Big Picture

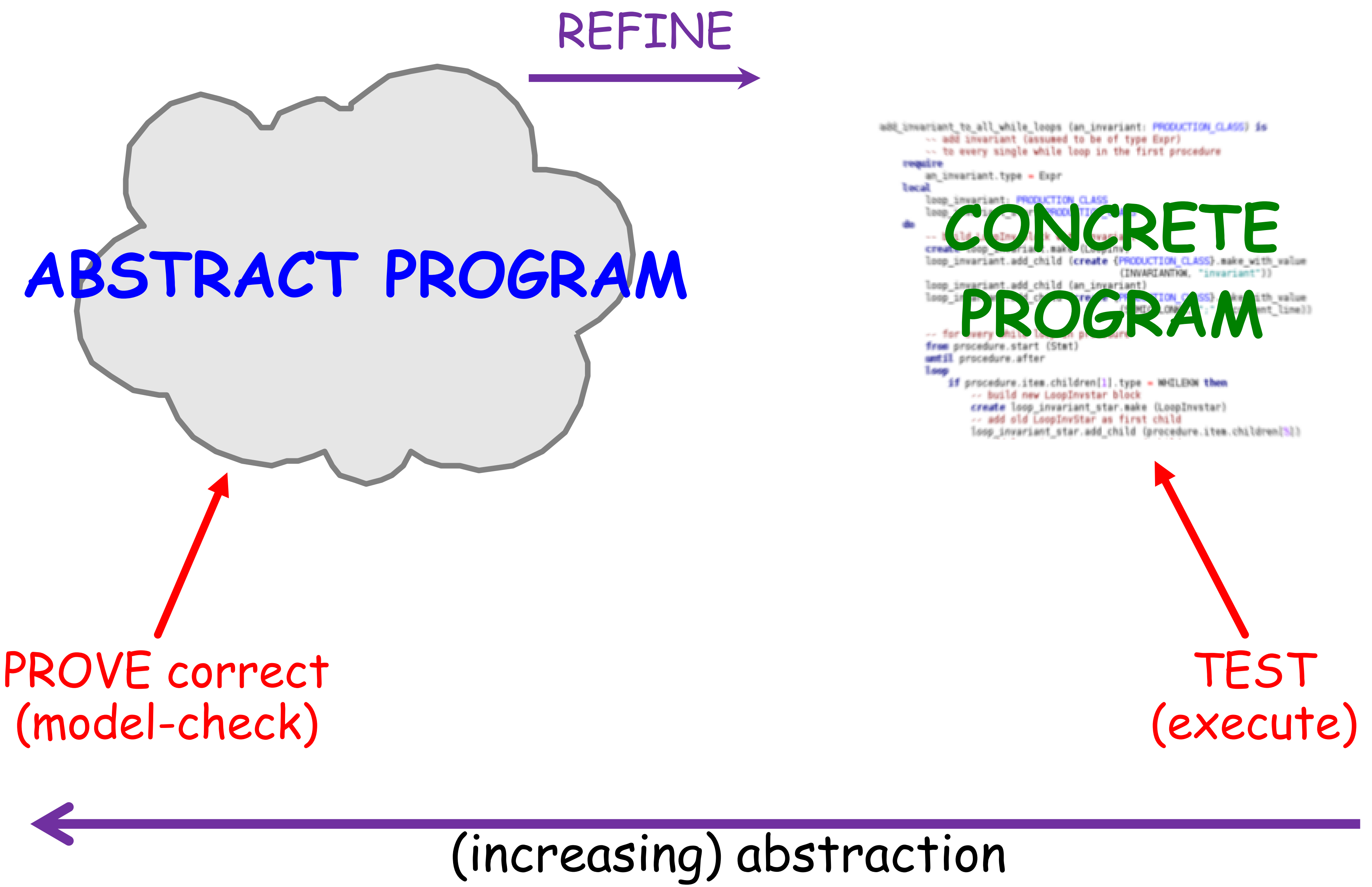
CEGAR Software Model Checking



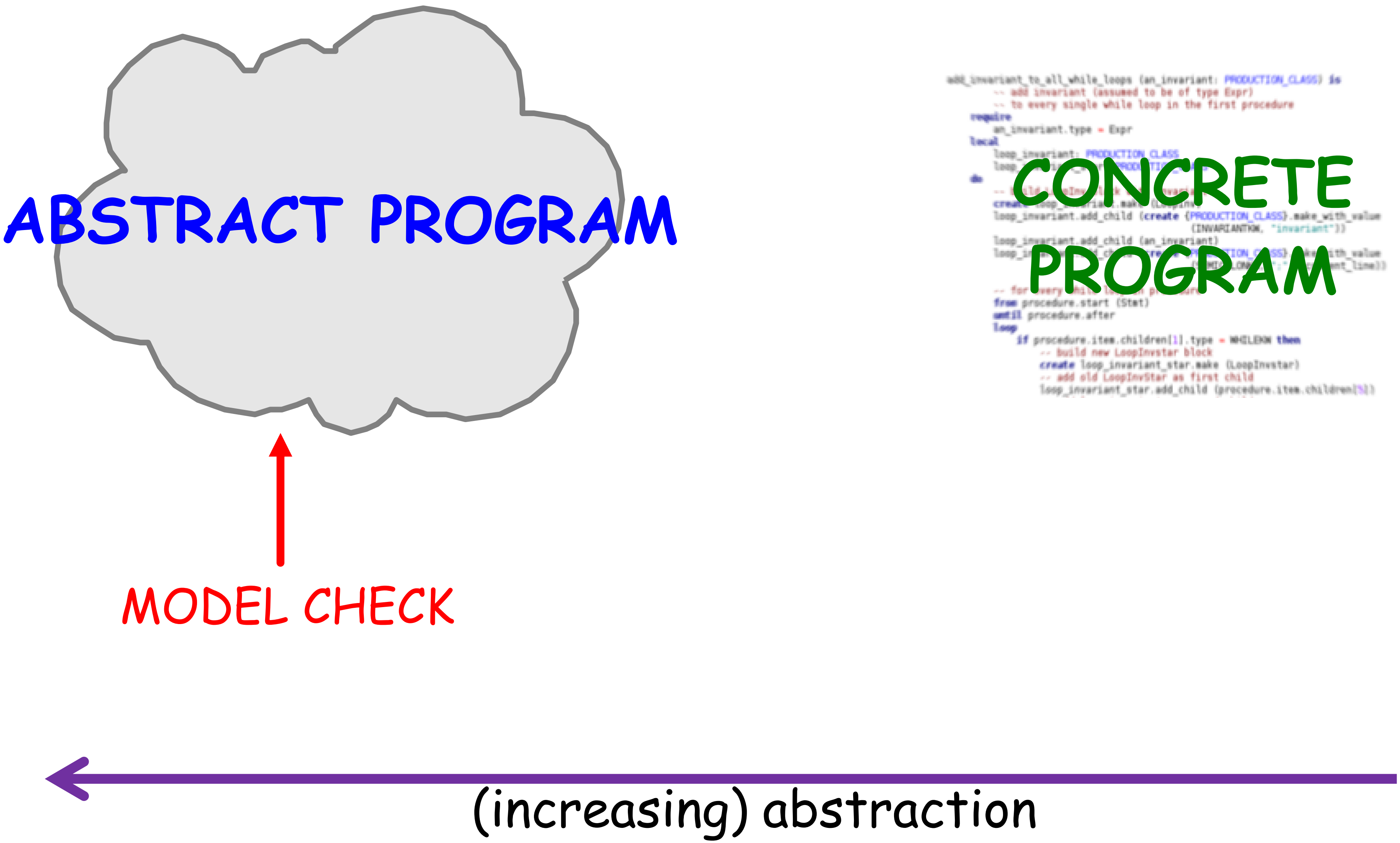
CEGAR Software Model Checking



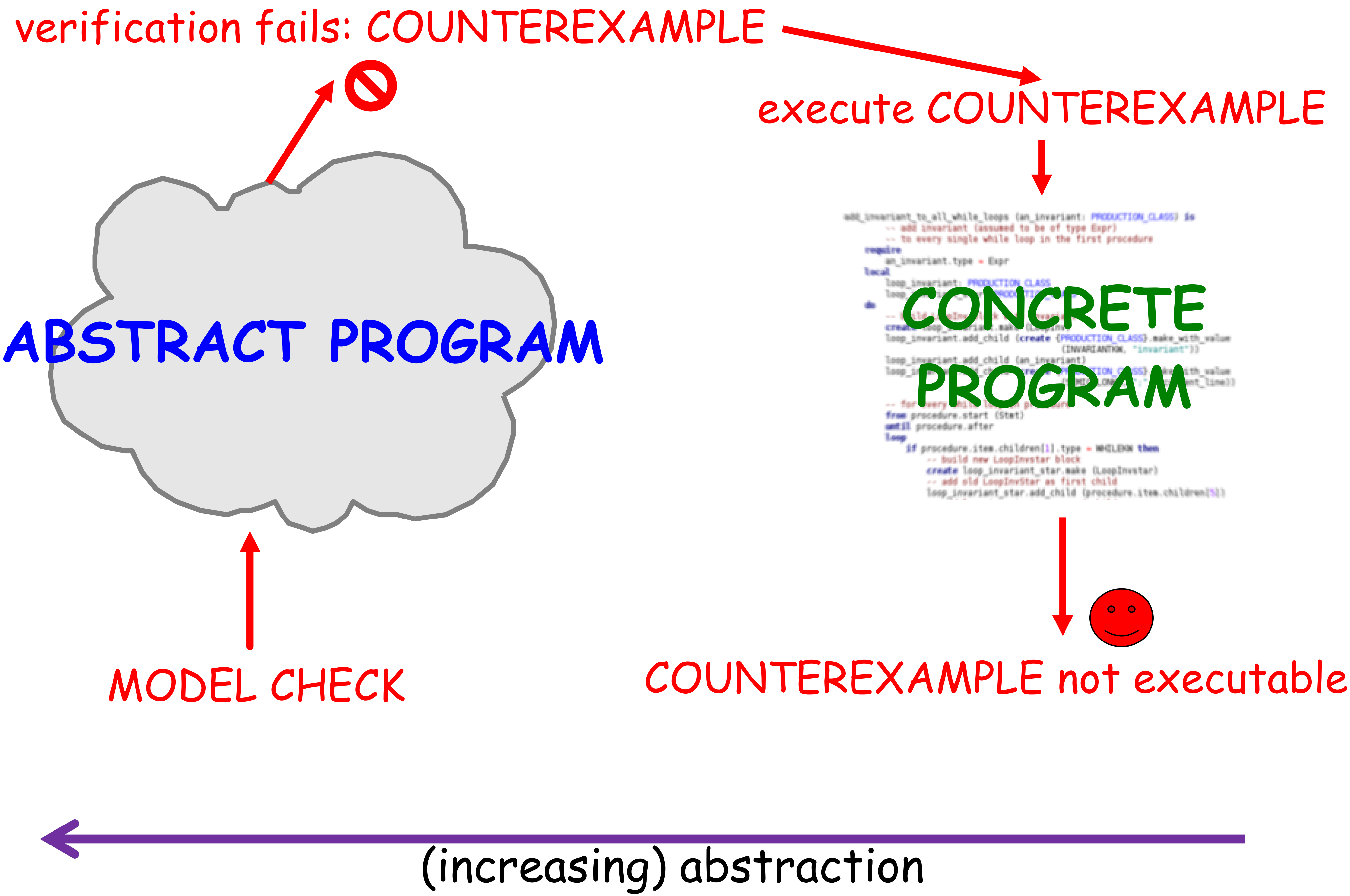
CEGAR Software Model Checking



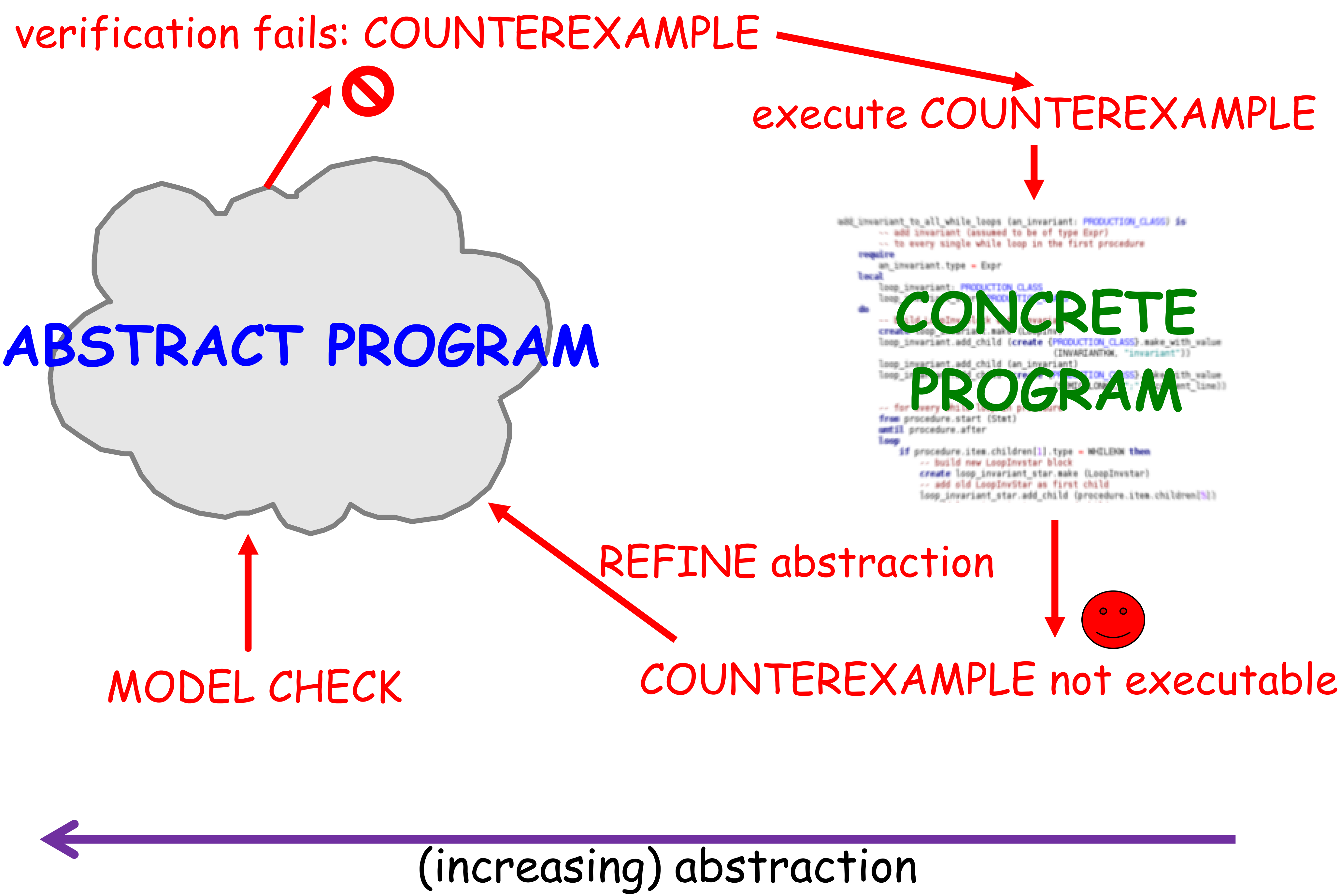
CEGAR Software Model Checking



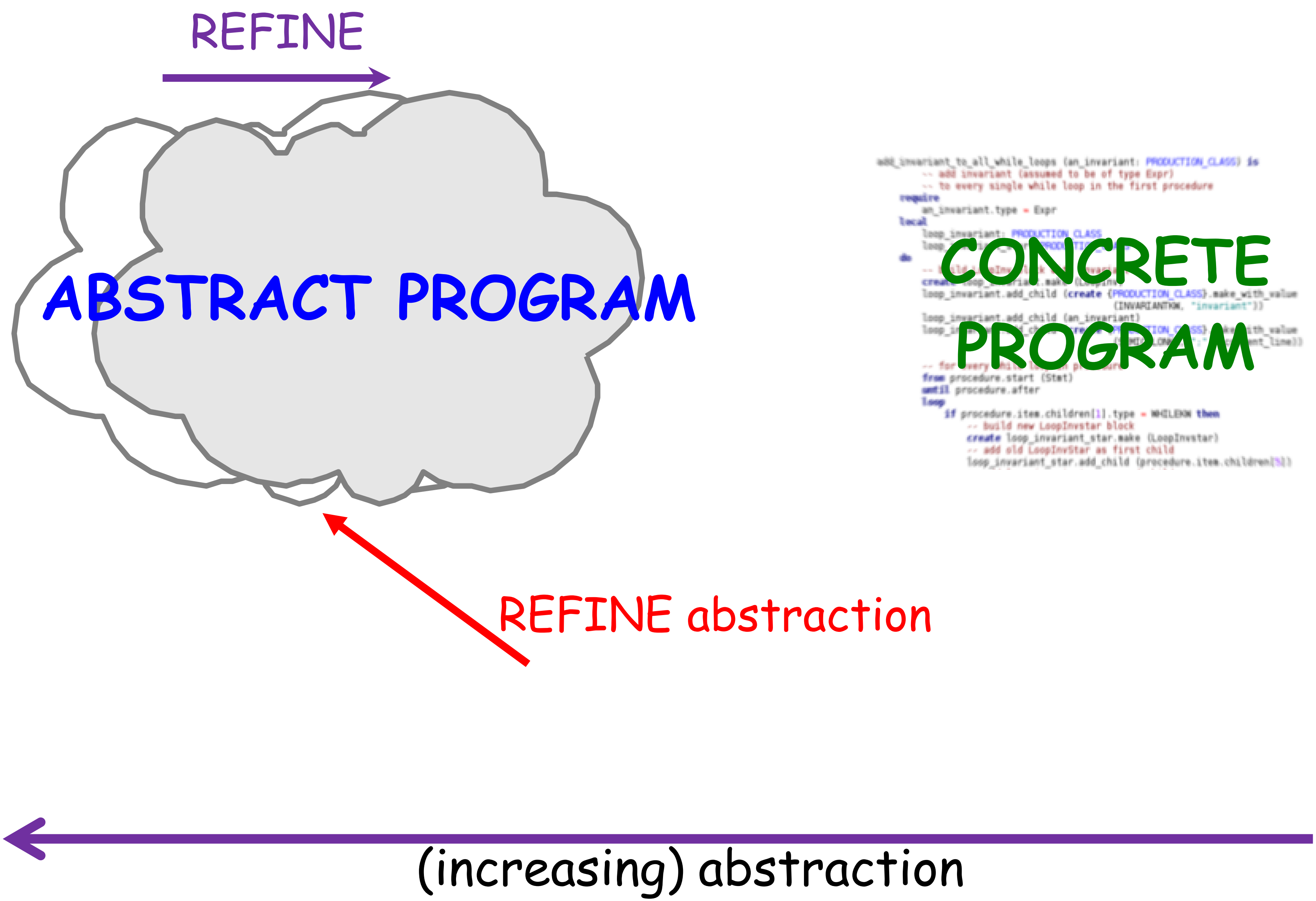
CEGAR Software Model Checking



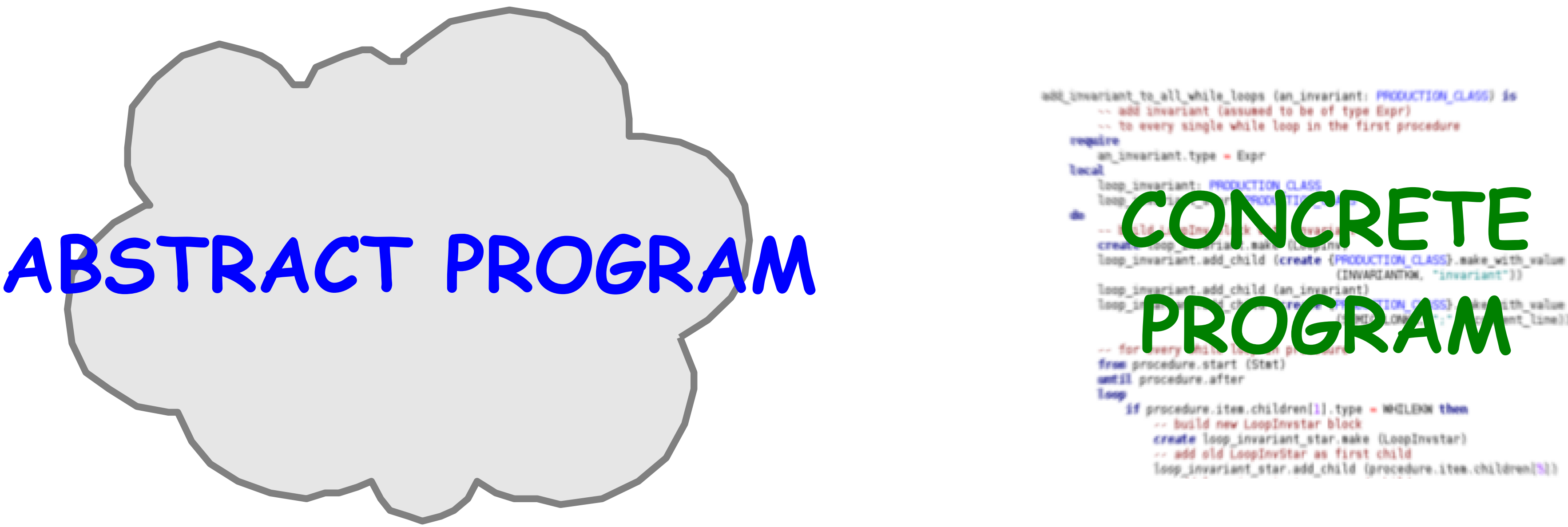
CEGAR Software Model Checking



CEGAR Software Model Checking



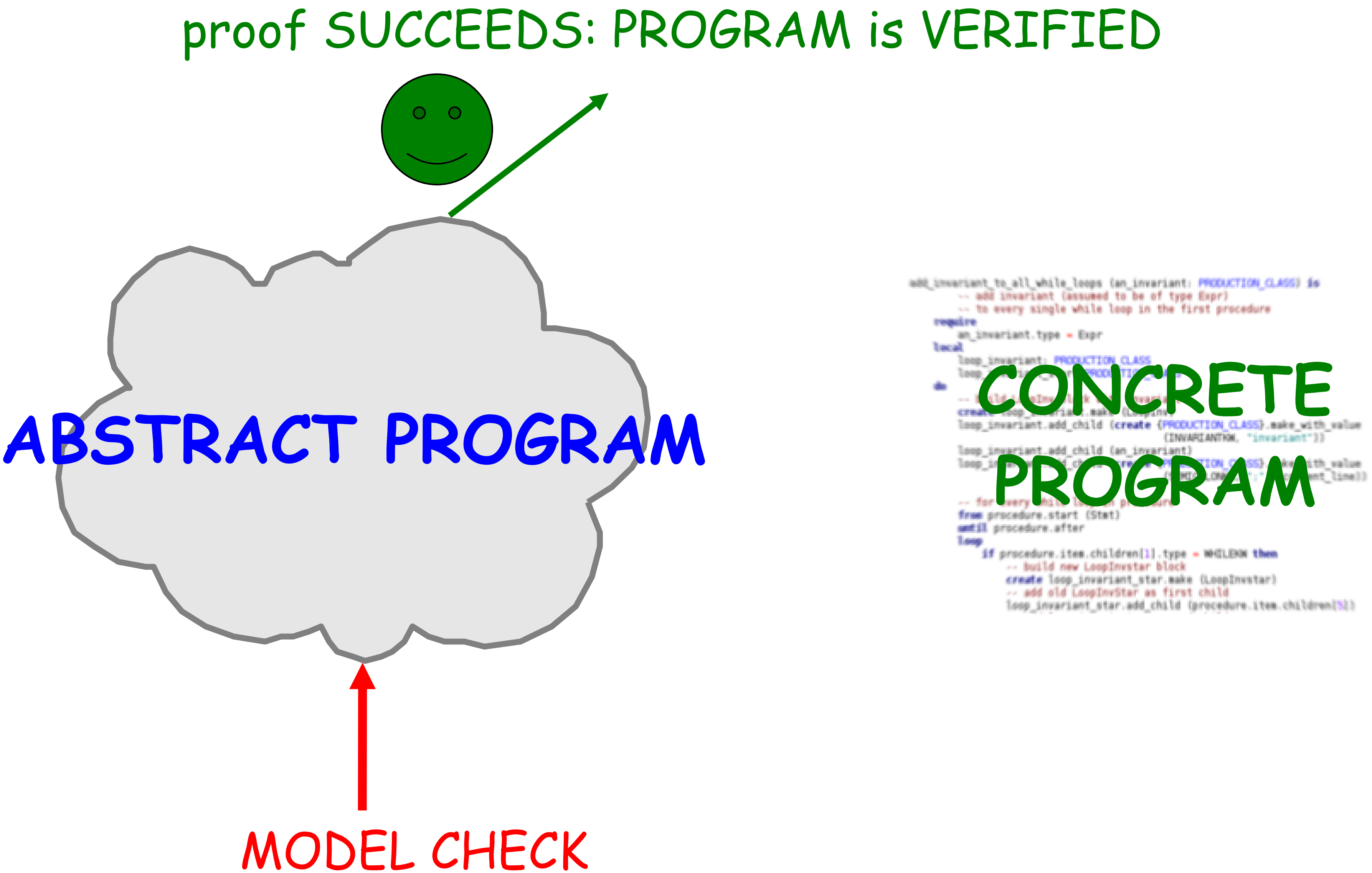
CEGAR Software Model Checking



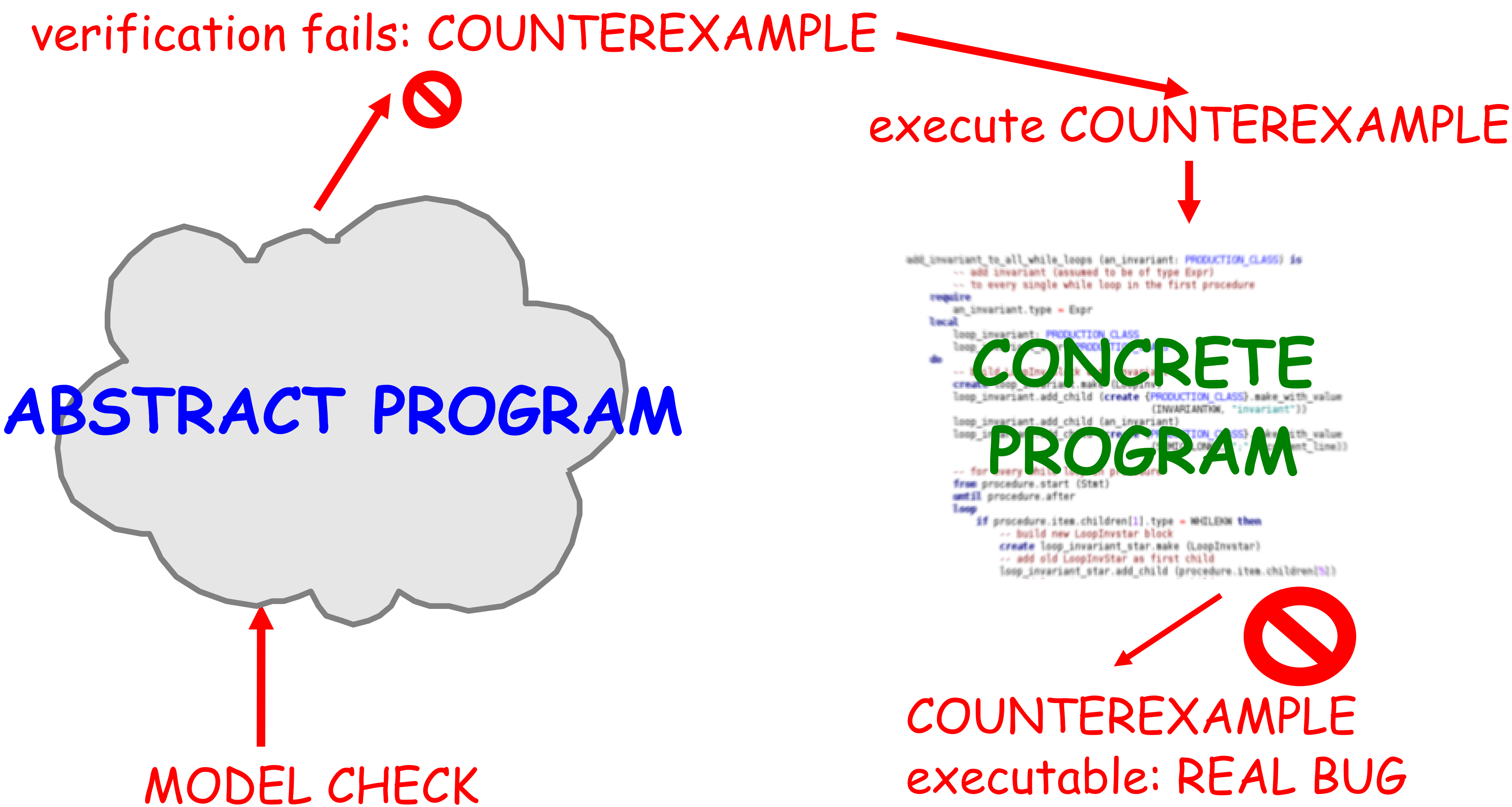
START OVER with new abstraction

(increasing) abstraction

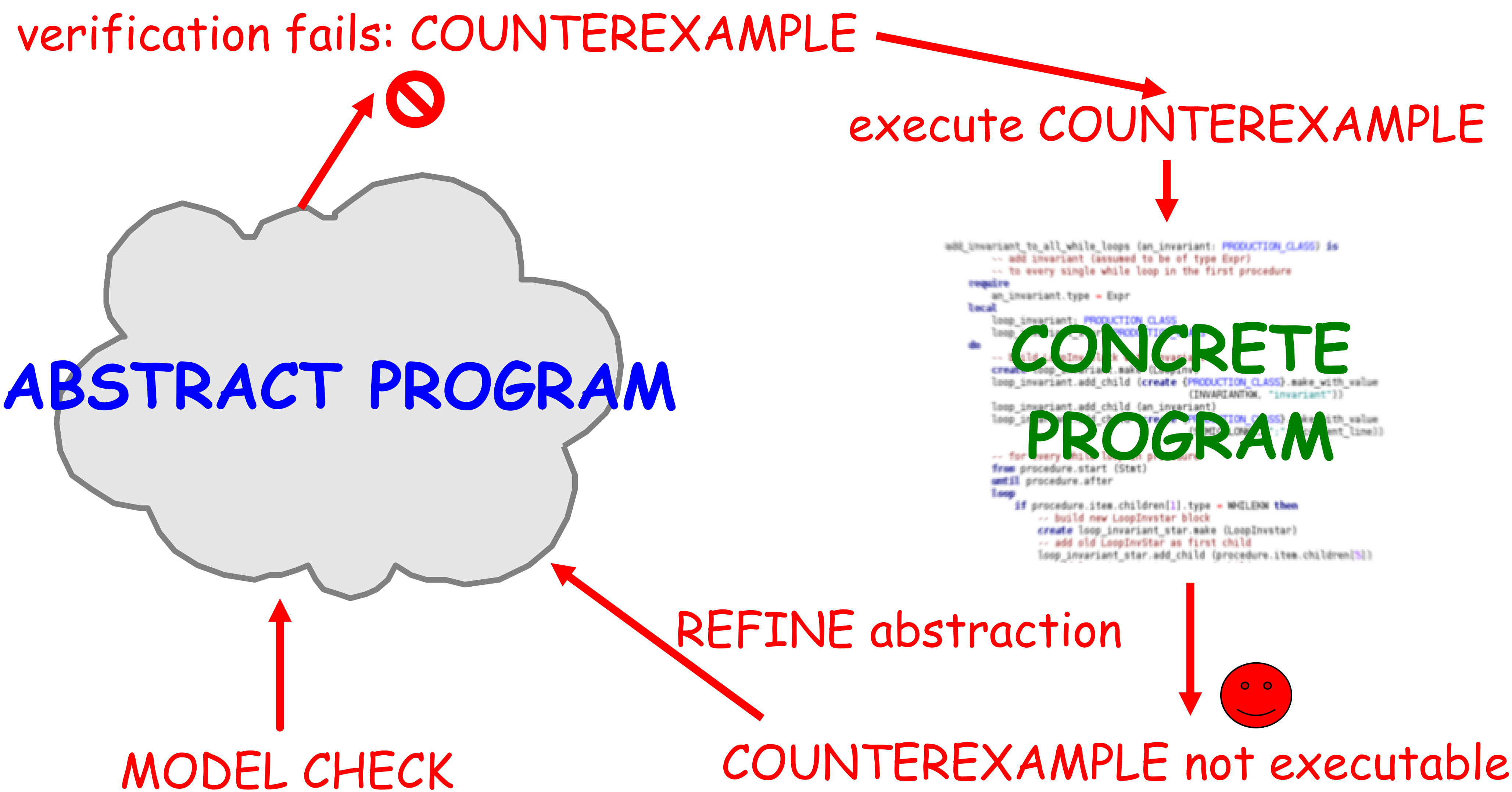
Outcome 1: Successful Verification



Outcome 2: Real Bug Found



Outcome 3: Loop Forever



CEGAR Software Model-Checking

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

Assertions and assumptions

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

- Assumptions describe information that every run reaching the instruction has.

  assume exp end

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

- Assertions describe information that every run continuing after the instruction 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.

- { exp $\Rightarrow$ Q } assume exp end { Q }
- { exp $\wedge$ Q } 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:

[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:

$\{ x = 3, y = 1 \}$	$x := y ; y := x$	$\{ x = \quad , y = \quad \}$
$\{ x = 3, y = 1 \}$	$x, y := y, x$	$\{ x = \quad , y = \quad \}$

Parallel assignments

For a straightforward presentation of the techniques, 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:

$\{ x = 3, y = 1 \}$	$x := y \,;\, y := x$	$\{ x = 1, y = 1 \}$
$\{ x = 3, y = 1 \}$	$x, y := y, x$	$\{ x = 1, y = 3 \}$

Parallel conditional assignment

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

if c_1^+ then $v_1 := e_1^+$ elseif c_1^- then $v_1 := e_1^-$ else $v_1 := e_1^?$ end

if c_2^+ then $v_2 := e_2^+$ elseif c_2^- then $v_2 := e_2^-$ else $v_2 := e_2^?$ end

...

if c_m^+ then $v_m := e_m^+$ elseif c_m^- then $v_m := e_m^-$ else $v_m := e_m^?$ 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

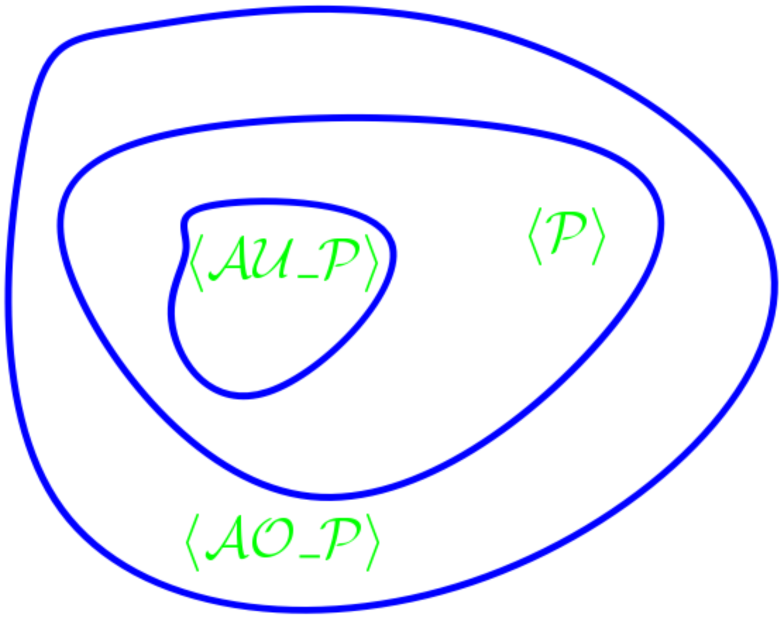
Abstraction is a pervasive idea in computer science. It has to do with modeling some crucial (behavioral) aspects while ignoring some other, less relevant, ones.

- Semantics of a program P: a set of runs ⟨P⟩
  - 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 ∈ ⟨P⟩ there exists a r' ∈ ⟨AO_P⟩
  - intuitively: ⟨P⟩ ⊆ ⟨AO_P⟩
  - 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 ∈ ⟨AU_P⟩ there exists a r' ∈ ⟨P⟩
  - intuitively: ⟨AU_P⟩ ⊆ ⟨P⟩
  - 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 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 pred of chosen predicates as an over-approximation of the program P
  - the arguments of A_P are the predicates in pred

    assume arguments are both input and output arguments (this deviates from Eiffel's semantics)
  - each instruction inst in P is replaced by a (possibly compound) instruction inst' in A_P such that:

    if executing inst in P leads to a concrete state S, then executing inst' in A_P leads to a state which is the strongest over-approximation of S in terms of 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

$$pred = \{p(1), ..., p(m)\}$$

and a set of corresponding Boolean expressions over program variables

$$exp = \{e(1), ..., e(m)\}$$

For a generic Boolean expression f over program variables, Pred(f) denotes the weakest Boolean expression over pred that is at least as strong as f.

- Substituting every atom p(i) in Pred(f) with the corresponding expression e(i) gives an expression that implies f.
- Pred(f) is an under-approximation of f, used to build the strongest over-approximations of instructions.

Boolean Under-Approximation: Example

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

- Pred($x = 1$) =
- Pred($x = 0$) =
- Pred($x \le 2$) =
- Pred($x \ne 0$) =

Boolean Under-Approximation: Example

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

- $Pred(x = 1) = p$
- $Pred(x = 0) = False$
- $Pred(x \leq 2) = p \vee q$
- $Pred(x \neq 0) = p \vee q \vee \neg r$

- In general: $Pred(\neg f) \neq \neg Pred(f)$

Boolean Under-Approximation: rule of thumb

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:

- pred = { p, q }
- exp = { $x < 2$, $x \leq 2$ }

  Pred($x \leq 3$) = p v q
  equivalent to = q

- The following transformations are robust w.r.t. the choice of equivalent Pred(f).
- When predicates imply each other, however, simplifications are possible (see later), so we'd better always include all implied facts in Pred(f).

Abstraction of Assignments

An assignment: x := f

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

```
if Pred(+f(i)) then
        p(i) := True
elseif Pred(-f(i)) then
        p(i) := False
else p(i) := ?          end
```

- +f(i) is the backward substitution of e(i) through x := f
- -f(i) is the backward substitution of ¬e(i) through x := f

Abstraction of Assignments: Example

- pred = { p, q, r }
- exp = { x > y, Result ≥ x, Result ≥ y }

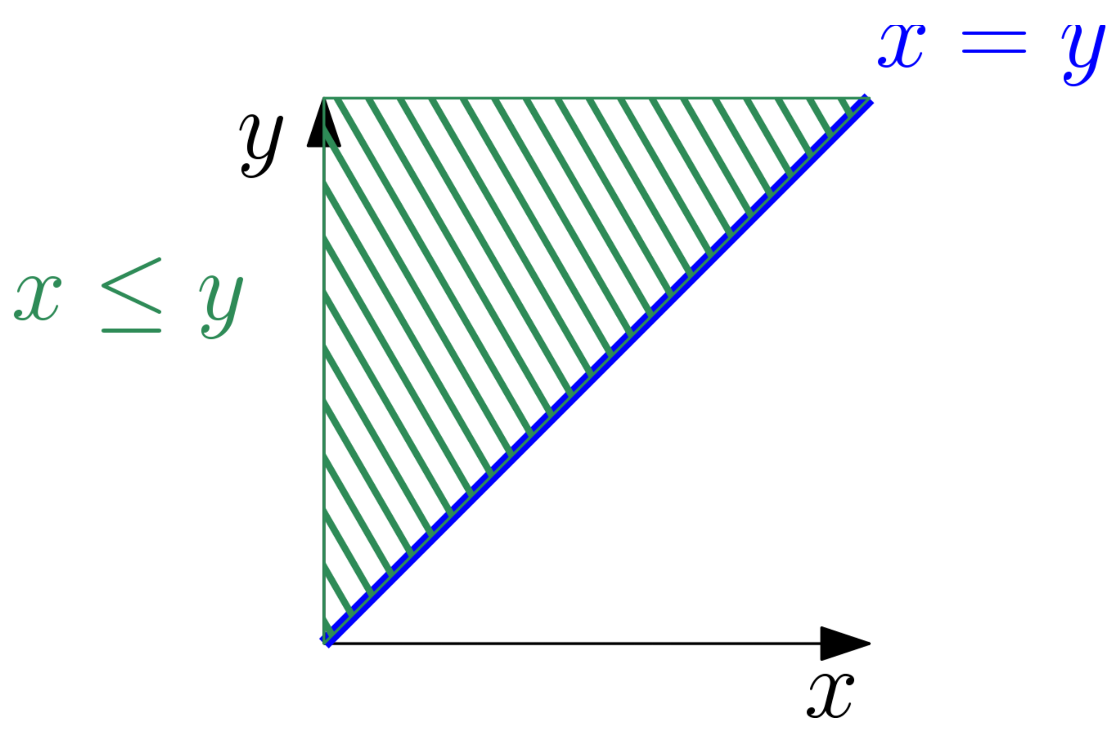
- Result := x is over-approximated by:
  - if p then p := True elseif not p then p := False else p := ? end
    - which does nothing
  - if True then q := True elseif False then q := False else q := ? end
    - which is equivalent to: q := True
  - if p then r := True elseif False then r := False else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

Abstraction of Assignments: Example

- pred = { p, q, r }
- exp = { x = 1, y = 1, x > y }

y := x
is over-approximated by
q := p ; r := False

{ x = y }
is over-approximated by
{ x ≤ y } ∩
({ x = y = 1 } ∪ { x, y ≠ 1 })
or, equivalently,
{ x ≤ 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 = True), q (x = 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: 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 ≤ i ≤ m:

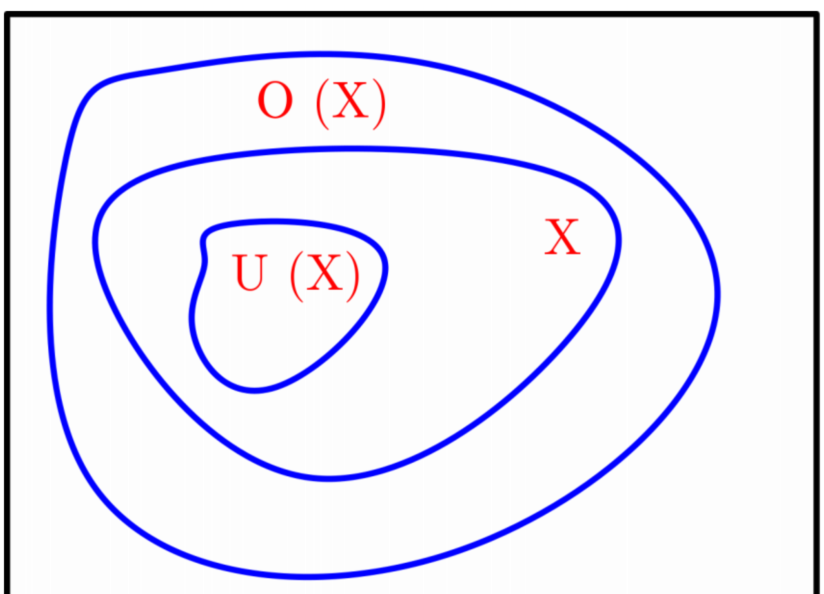
```
if Pred(+ex(i)) then
        p(i) := True
elseif Pred(-ex(i)) then
        p(i) := False
else p(i) := ?          end
```

- +ex(i) is the backward sub. of e(i) through assume ex end
- -ex(i) is the backward sub. of ¬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 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≤3) }
- Pred(x ≤ 2) = p ∨ q
- Pred(x > 2) = ¬r
- assume x ≤ 2 end
- assume p ∨ q end is assume x=1 ∨ x=2 end
- assume ¬(¬r) end is assume x ≤ 3 end

Abstraction of Assumptions: Simplification

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

In fact:

Pred(+ex(i)) = Pred(not ex $\vee$ ex(i))
= Pred(not ex) $\vee$ Pred(ex(i)) (can you prove this?)
= not Pred(not ex) $\Rightarrow$ p(i)

Which, given the assumption, implies: p(i)

Pred(-ex(i)) = Pred(not ex $\vee$ not ex(i))
= Pred(not ex) $\vee$ Pred(not ex(i))
= not Pred(not ex) $\Rightarrow$ not p(i)

Which, given the assumption, implies: not p(i)

In all:

if p(i) then p(i) := True elseif not p(i) then p(i) := False else p(i) := ? 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

  ex $\Rightarrow$ ex(i) is valid

- a disjunct not p(i) such for every i such that

  ex $\Rightarrow$ not ex(i) is valid

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 $1 \le i \le m$:

```
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 ¬e(i) through 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 real abstraction will be introduced.

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
```

Predicates:

- p: x > y
- q: Result ≥ x
- r: Result ≥ y

```
Apqr_max (p, q, r: BOOLEAN) do
    if ? then
      assume x > y end  ;  Result := x
    else
      assume x ≤ y end  ;  Result := y
    end
ensure Result ≥ x and Result ≥ y 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
        Result := x

    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
        Result := y

    end

ensure q and r end
```

Predicates:

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

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

Predicates:

- p: x > y
- q: Result ≥ x
- r: Result ≥ y

```
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

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

Predicates:

- p: x > y
- q: Result ≥ x
- r: Result ≥ y

```
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
```

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 pred = {p(1), ..., p(m)}

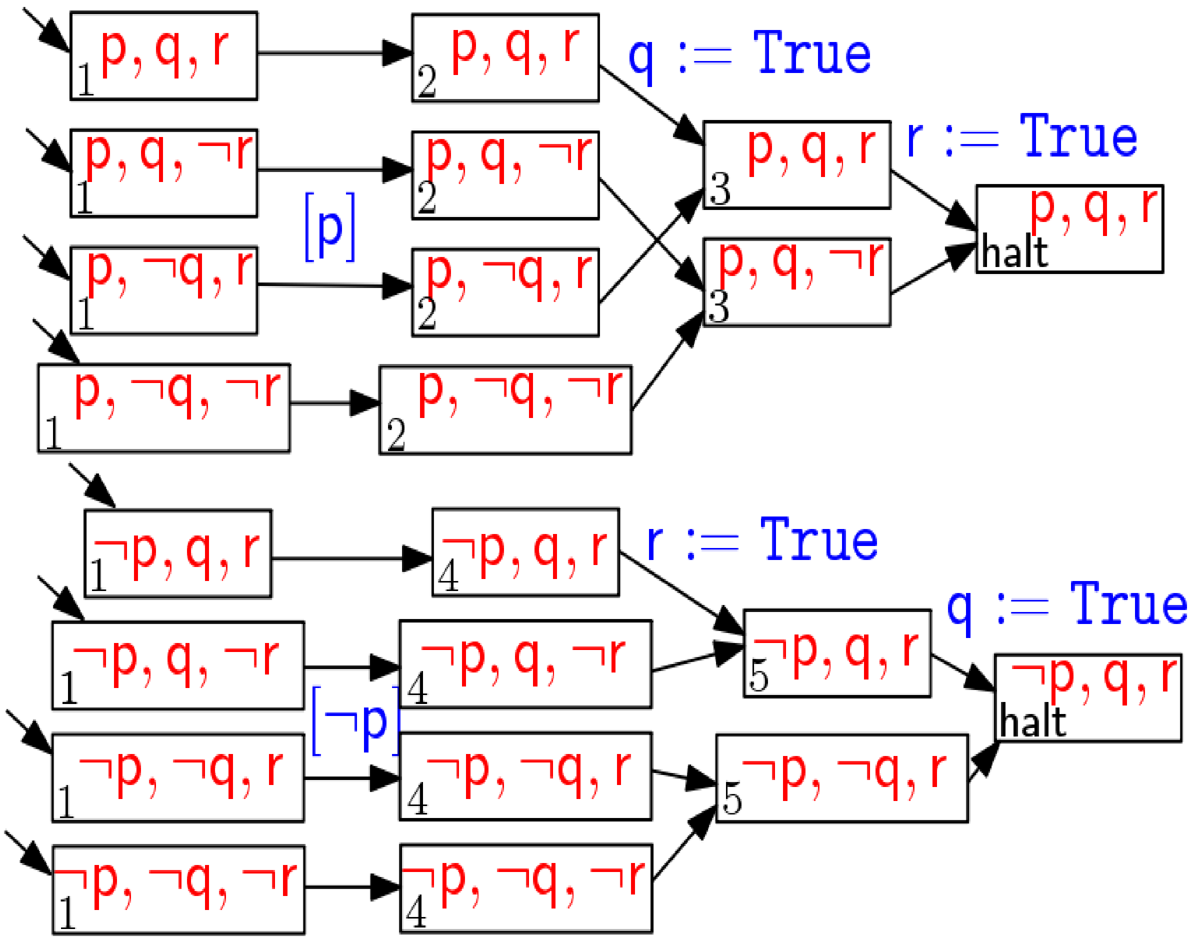
- P's body: a sequence loc = [L(1), ..., L(n)] of instructions or conditional jumps
- P's postcondition: post

Build an FSA = [Σ, S, I, ρ, F] where:

- Σ = loc
- S = {True, False}^{m} x (loc U {halt})
  - each state in S denotes a program state:
  - a truth value for every Boolean variable in pred
  - a program location which represents the next line to be executed, or halt if the execution has terminated
- I = { [v(1), ..., v(m), L(1)] ∈ S }
  - the initial states are for any value of the input Boolean arguments
  - L(1) is the next instruction to be executed
- [v'(1), ..., v'(m), L'] ∈ ρ ([v(1), ..., v(m), L], L) iff one of the following holds:
  - L is a conditional jump and: [v(1), ..., v(m)] satisfies the condition; v'(i) = v(i) for all 1 ≤ i ≤ m; L' is the target of the jump when successful.
  - L is a conditional jump and: [v(1), ..., v(m)] does not satisfy the condition; and v'(i) = v(i) for all 1 ≤ i ≤ m; L' is the target of the jump when unsuccessful
  - L is an instruction and: [v'(1), ..., v'(m)] is the state resulting from executing L on state [v(1), ..., v(m)]; and L' is the successor of L (or halt if the program halts after executing L)
- F = { [v(1), ..., v(m), halt] ∈ S | post does not hold for [v(1), ..., v(m)] }
  - error states: halting states where the postcondition doesn't hold

Predicate Abstraction: Example

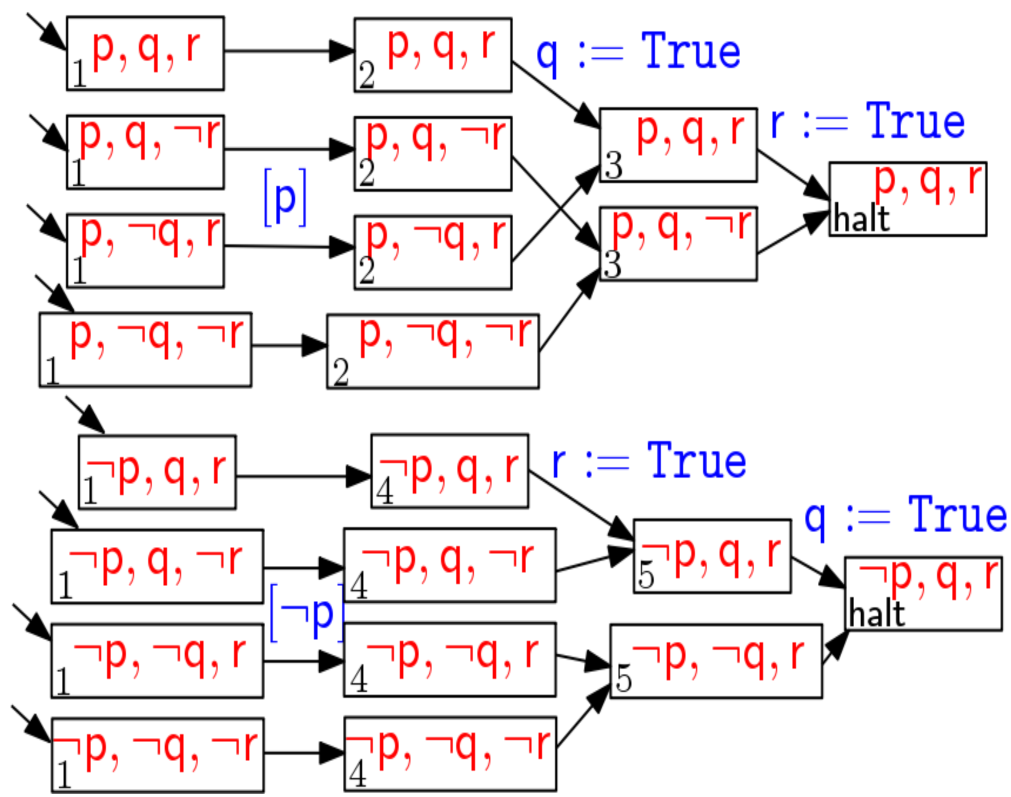
```
Apqr_ max (p, q, r: BOOLEAN) do
    1: if p
    2:   then  q := True
    3:         r := True
    4:   else  r := True
    5:         q := True
      end
ensure q and r end
```



Predicate Abstraction: Example

```
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
  - 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 (partial) 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

{ Pred(0) }	{ Abstract initial state }
inst(1)	Instruction or test
{ Pred(1) }	{ Abstract state }
inst(2)	Instruction or test
…	…
inst(N)	Instruction or test
{ Pred(N) }	{ Abstract final state }

Goal: find whether there exists a concrete run of P which is abstracted by this abstract counterexample

Abstract Counterexamples: Example

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

Predicates:

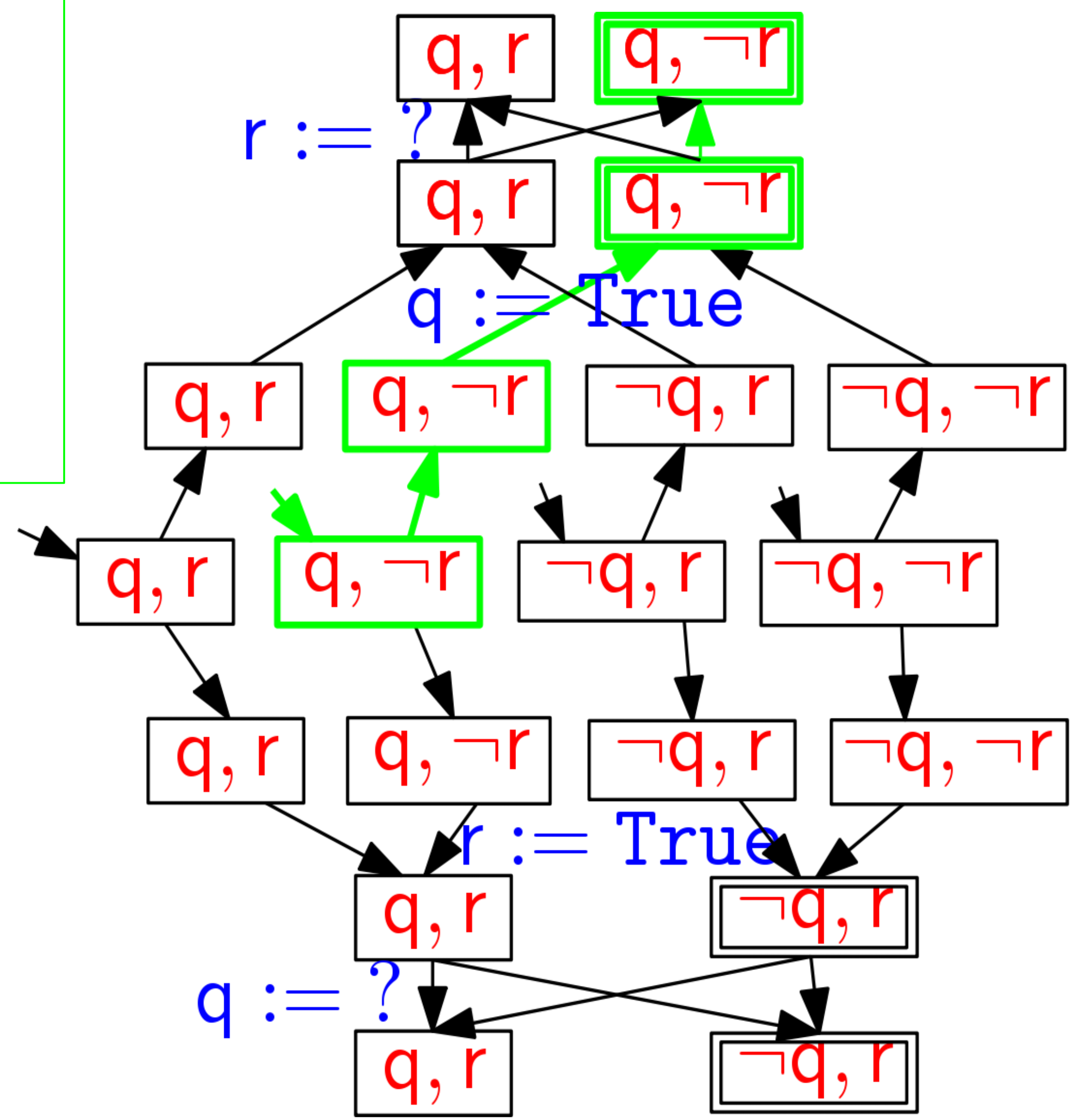
- q: Result ≥ x
- r: Result ≥ y

```
Aqr_max (q, r: BOOLEAN) do
    if ? then
      q := True ; r := ?
    else
      r := True ; q := ?
    end
ensure q and r 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:	Concrete run:
{ Pred(0) }	
inst(1)	Concrete-inst(1)
{ Pred(1) }	
inst(2)	Concrete-inst(2)
…	…
inst(N)	Concrete-inst(N)
{ Pred(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, modulo undecidability of the used logic fragment):

Abstract run:	Concrete run:
{ Pred(0) }	{ WP(0) }
inst(1)	Concrete-inst(1)
{ Pred(1) }	{ WP(1) }
inst(2)	Concrete-inst(2)
...	...
inst(N)	Concrete-inst(N)
{ Pred(N) }	{ True }

Every formula WP(i) characterizes the states of P reaching a final state where 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 $1 \leq i \leq N$.

Concrete run:

{ Pred(0) and WP(0) }

Concrete-inst(1)

{ Pred(1) and WP(1) }

Concrete-inst(2)

...

Concrete-inst(N)

{ Pred(N) and True }

Spurious Counterexamples: Example

Abstract c.e. trace:

$\{q, \neg r\}$

[?]

$\{q, \neg r\}$

q := True ; r := ?

$\{q, \neg r\}$

Concrete trace:

$\{x > y\}$

assert x > y end

{True}

Result := x

{True}

The counterexample is infeasible because:

$\{x > y$ and q and $\neg r\}$ is inconsistent

as $\{x > y$ and $q\}$ implies $\{r\}$

Sufficient condition for infeasibility

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).
  - Essentially, we just have to deal with ? values appropriately
- 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
```

Predicates:

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

```
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
```

Abstract Counterexamples: Example

```
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

Abstract c.e. trace:

{p, q, ¬r}
  r := True
{p, q, r}
  [q]
{p, q, r}
  [p and r]
{p, q, r}
  r := False
{p, q, ¬r}
  q := ?
{p, ¬q, ¬r}
  [¬q]
{p, ¬q, ¬r}

Abstract Counterexamples: Example

Abstract c.e. trace:

{p, q, ¬r}
r := True
{p, q, r}
[q]
{p, q, r}
[p and r]
{p, q, r}
r := False
{p, q, ¬r}
q := ?
{p, ¬q, ¬r}
[¬q]
{p, ¬q, ¬r}

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}

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:

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

Concrete-inst(1)

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

Concrete-inst(2)

...

Concrete-inst(N)

{ Pred(N) and True } { True } \ { 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} \ {q, ¬r}

assert x > y end

{True} \ {q, ¬r}

Result := x

{True} \ {q, ¬r}

Predicates:

- q: Result >= x
- ¬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, ¬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...