A Pyramid Of (Formal) Software Verification

Martin Brain and Elizabeth Polgreen

City St. George, University of London
martin.brain@city.ac.uk
University of Edinburgh
elizabeth.polgreen@ed.ac.uk

February 22, 2025

I’m Going To Say Some Things...

...but not as many things as I would like to say

- These slides:
  `http://polgreen.github.io/pdfs/pyramid-slides.pdf`
- Tutorial paper:
  `http://polgreen.github.io/pdfs/pyramid-paper.pdf`
- Tutorial on youtube:
  `https://youtu.be/BlGZuQIESRU?si=qEzNGt6wvqtq91m1`

What is Verification?

A process that produces evidence that a system complies with a specification.

The Role of System and Specification

Properties

(Temporal logic)

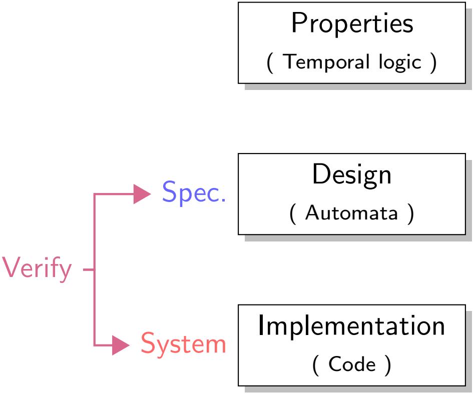
Design

(Automata)

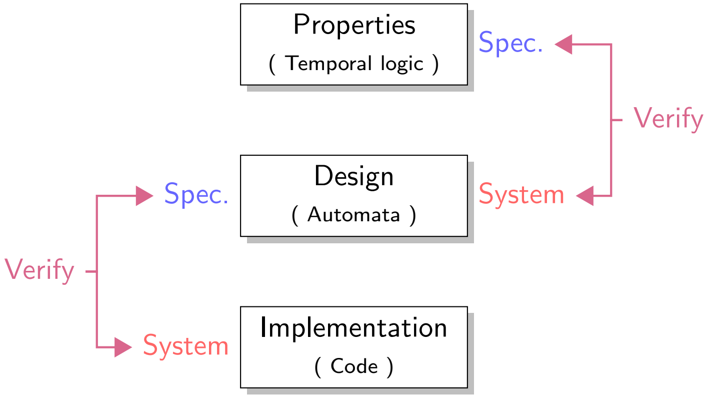
Implementation

(Code)

The Role of System and Specification



The Role of System and Specification



Verification vs. Validation

Verification

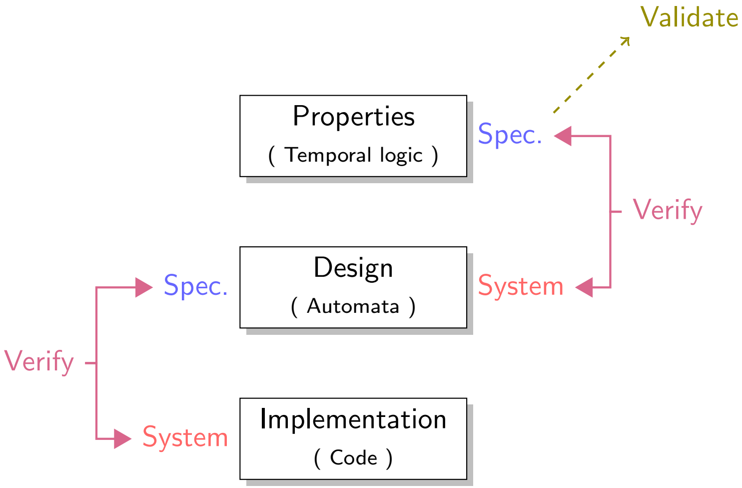
Are We Building The Thing Right?

Validation

Are We Building The Right Thing?

"Correctly building the wrong thing!" – verification without validation!

The Role of System and Specification



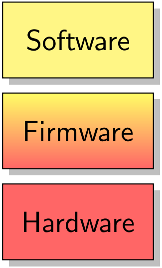
When To Verify

Writing Verified Software

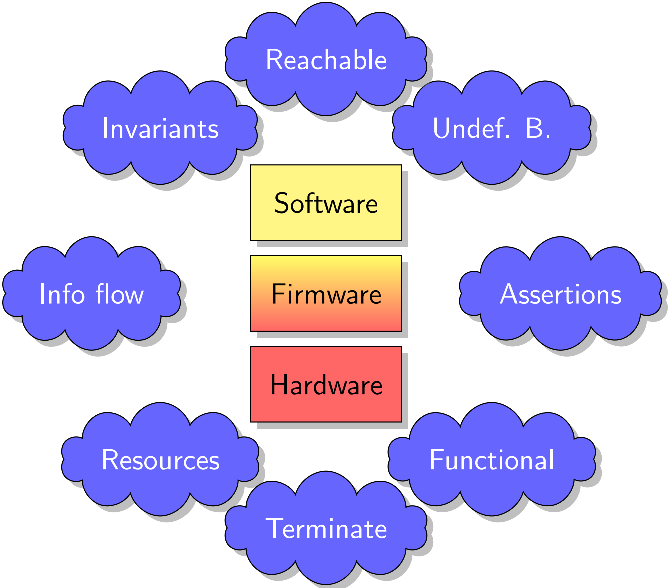
VS.

Verifying Written Software

System and Specification



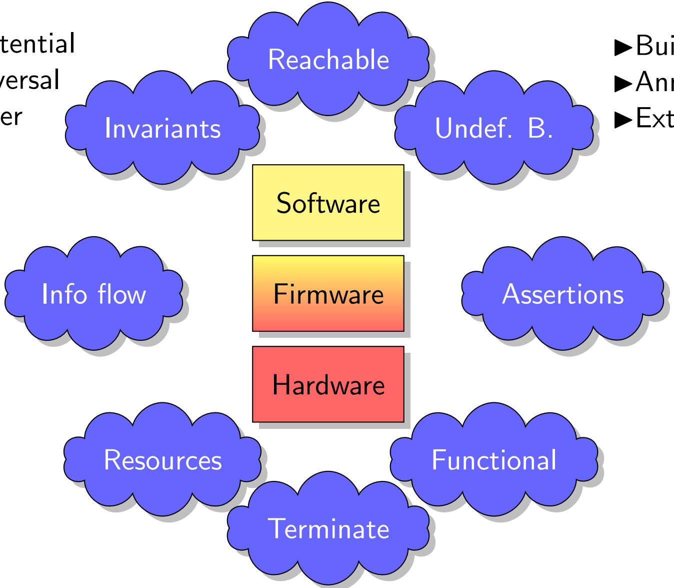
System and Specification



System and Specification

- Existential
- Universal
- Hyper

- Built in
- Annotation
- External



Why bother?

Why bother?

Why bother?

Why bother?


The Royal Courts of Justice
IDENTIFY & HOLD THE INDIVIDUALS
FULL RECOMPENSE
SUPPORT OUR SUB - POSTMASTERS

Why bother?

FROM LEFT: MEDIA STAND IN A PARKING LOT, WAITING FOR A PRIUS TO RUN AWAY; RECALLS KEPT TOYOTA SERVICE BAYS PLENTY BUSY

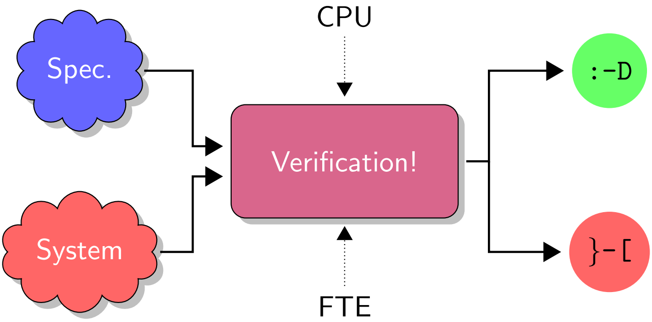
Why bother?

...

Verification Tools

The Key Question

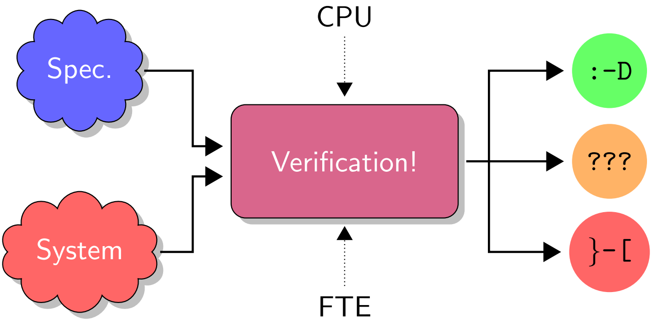
Does (every run of) system P satisfy specification S?



Verification Tools

The Key Question

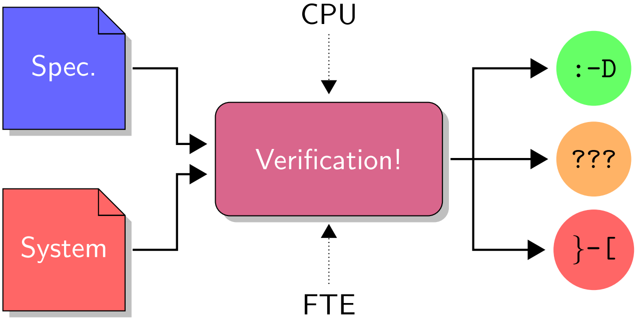
Does (every run of) system P satisfy specification S?



Verification Tools

The Key Question

Does (every run of) system P satisfy specification S?

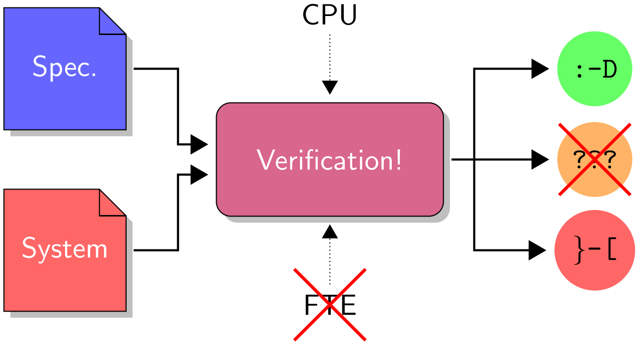


If program and specification are formal → can automate!

The Ideal Tool

The ideal verification tool . . .

- fully automated
- never misses bugs
- never gives false alarms



The Key Trade-Off

Turing's Work Implies . . .

It is impossible to build an *automatic* verification system for:

- *any* specification that includes reachability
- *all* software including loops / recursion

The Key Trade-Off

The Key Trade-Off

Choosing Verification Tools

1. Pick *two* attributes based on project:
   automatic, no missed bugs, no false alarms
2. Use CPU and human effort to get enough of the third.
   (for your *specific* software/specifications.)

The Key Trade-Off

Choosing Verification Tools

1. Pick *two* attributes based on project:
   automatic, no missed bugs, no false alarms
2. Use CPU and human effort to get enough of the third.
   (for your *specific* software/specifications.)

OR

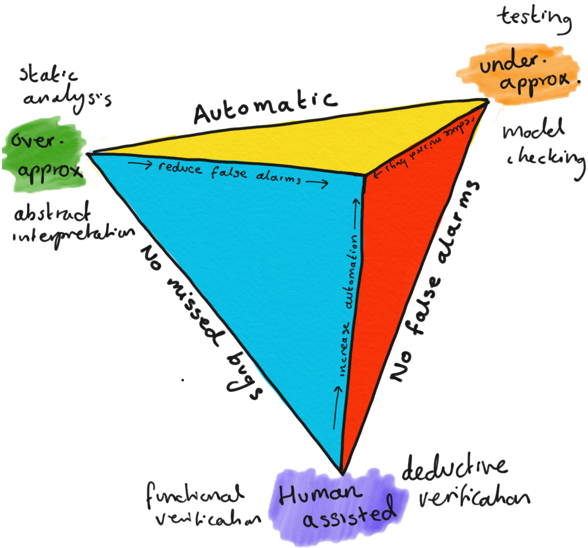
Work out which one you hate the least:

False Alarms Sentencing potential bug reports.

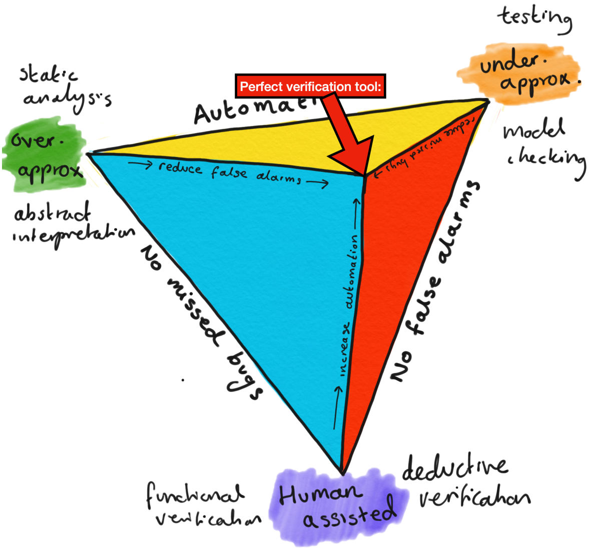
Missed Bugs Measuring coverage and writing harnesses.

Manual Supplying annotation or proof.

The Pyramid Model of Software Verification

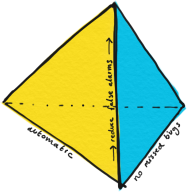


The Pyramid Model of Software Verification



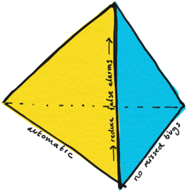
Six Schools: Over-approximate

- Static Analysis, e.g., Lint
  - Spurious warnings are fine, as long as there aren't too many
  - Lexical scanners: look for patterns in code that are likely to be bugs



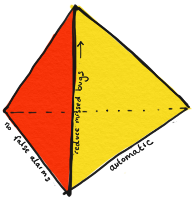
Six Schools: Over-approximate

- Static Analysis, e.g., Lint
  - Spurious warnings are fine, as long as there aren't too many
  - Lexical scanners: look for patterns in code that are likely to be bugs
- Abstract Interpretation, e.g., Infer
  - Run the program with representations of sets of possible values (the domain)
  - e.g., intervals



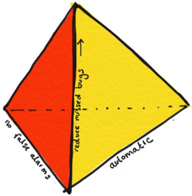
Six Schools: Under-approximate

- Testing and Symbolic Execution e.g., KLEE
  - Run the program!
  - Or, run the program with symbols instead of concrete inputs



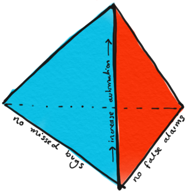
Six Schools: Under-approximate

- Testing and Symbolic Execution e.g., KLEE
  - Run the program!
  - Or, run the program with symbols instead of concrete inputs
- Model checking e.g., CBMC
  - Build a model of the program.
  - Use the model to build a formula that represents all paths in the program.
  - Use a SAT solver (or BDDs) to see if there is a path that violates the spec.



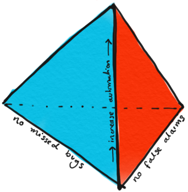
Six Schools: Human-assisted

- Deductive Verification e.g., SPARK
  - Describe the set of states with a predicate
  - Use logic to link these together (e.g., Hoare logic)
  - Use proof by induction for loops (unbounded proof!)



Six Schools: Human-assisted

- Deductive Verification e.g., SPARK
  - Describe the set of states with a predicate
  - Use logic to link these together (e.g., Hoare logic)
  - Use proof by induction for loops (unbounded proof!)
- Functional Verification e.g., Agda
  - Define a programming language that *only* lets you build correct programs
  - Specifications are captured in types
  - Only good for verifying programs as you write them

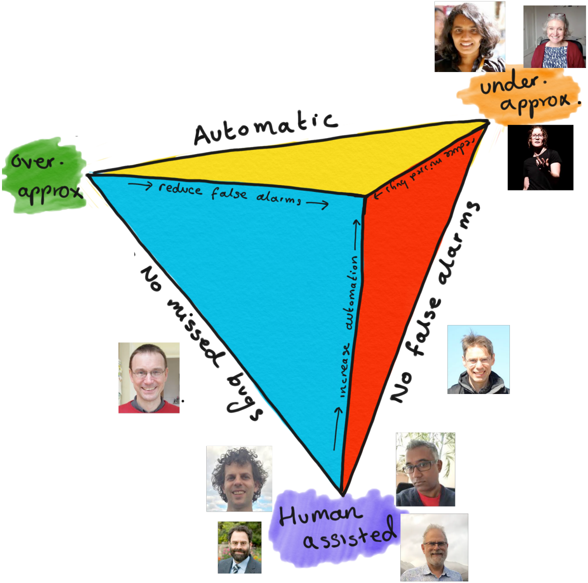


Six Schools

	Over-Approximate		Under-Approximate		Human-Assisted	
	Static Analysis	Abstract Interpretation	Testing & Symbolic Execution	Model Checking	Deductive Verification	Functional Verification
Program	Procedural or O.O.	Procedural	Procedural or O.O.	Procedural or O.O.	Subsets of procedural	Functional
Commmon Means of Specification	Builtin	Annotation linked to the abstraction	Generally annotation	Annotation or external	Annotation	Type as annotation
Common Type of Specification	Data flow, aliasing, type, shape, taint	Value, shape, resource, data flow	Value, WCET, resource	Value, temporal, modal, liveness	Value, shape, termination, resource	Type, termination
Mathematical Foundations	Ad-hoc / operational semantics	Order theory	Ad-hoc / transition systems	Transition systems	Logic	Type theory
User Skill Required	Minimal	Low/Medium	Low	Medium	High	Very high
Compute Required	Minimal	Low/Medium upwards	Medium upwards	Medium/High upwards	Low/Medium	Low
Typical Output	Algorithm dependent	Alarms or abstract domains	Error traces	Error traces	Proof or local counter-examples	Type-checking errors
Major Systems	Lint[?], Coverity[?], Fortify[?], FindBugs[?], CPPCheck[?]	Astrée[?], Polyspace[?], Infer[?], Code Contracts[?]	CREST[?], JPF[?], Pex[?], KLEE[?]	CBMC[?], Blast[?], *SMV[?], CPAchecker[?]	SPARK[?], Dafny[?], Frama-C[?], Malpas[?], Esc/Java[?]	Coq[?], PVS [?], Agda[?], Isabelle/Hol[?]

Table: Cultural attributes of the six schools.

Six Schools



Automatic
over. approx
under. approx.
Human assisted
reduce false alarms
increase automation
No missed bugs
No false alarms

Running Example: Formalise and Verify

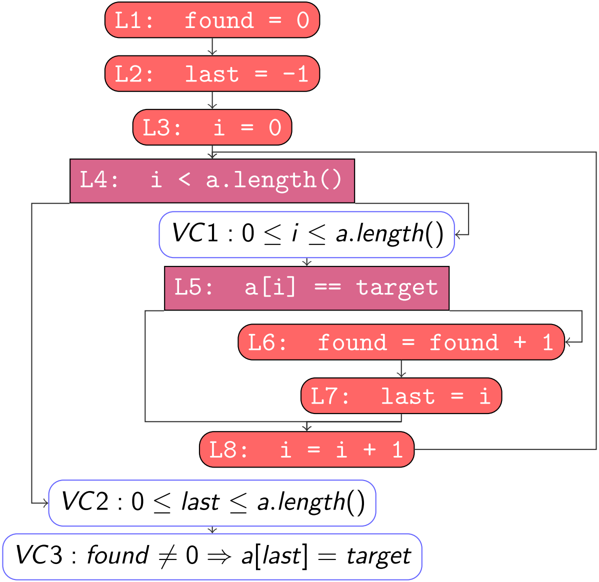
Spec.

1. All array accesses in bounds
2. Returned `last` is in `a`
3. If `found` then `a[last]` is `target`

System

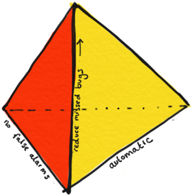
```
(int, int)
count (Array a, int target)
{
  int found = 0;
  int last = -1;
  int i = 0;

  while (i < a.length()) {
    if (a[i] == target) {
      found = found + 1;
      last = i;
    }
    i = i + 1;
  }

  return (found, last);
}
```

Running Example: Formalise and Verify



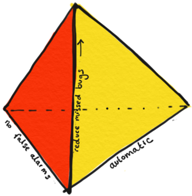
Under-Approximate Family Tree

- Testing
  - Symbolic Execution
- Explicit state
  - Symbolic state
    - BMC / *k*-Induction
      - CEGAR / Interpolation



Under-Approximate Family Tree

- Testing
  - Symbolic Execution
- Explicit state
  - Symbolic state
    - BMC / k-Induction
      - CEGAR / Interpolation



Symbolic Execution: Foundations (Ideas)

Can run one test case which gives an execution trace...

Symbolic Execution: Foundations (Ideas)

Can run one test case which gives an execution trace...
Can we generalise this to "similar" traces?

Symbolic Execution: Foundations (Ideas)

Can run one test case which gives an execution trace...
Can we generalise this to “similar” traces?
Use logic to describe a *set* of traces (that take the same path).

Symbolic Execution: Foundations (Symbols)

X is set of variables, $I \subset X$ is a set of input variables
$Expr(I)$ is the set of expressions over I
$Prd(I)$ is the set of predicates over I

Symbolic Execution: Foundations (Symbols)

X is set of variables, $I \subset X$ is a set of input variables
$Expr(I)$ is the set of expressions over I
$Prd(I)$ is the set of predicates over I

Representation A map $Env : X \to Expr(I)$ and a set $PC \subset Prd(I)$.

Symbolic Execution: Foundations (Symbols)

X is set of variables, $I \subset X$ is a set of input variables
$Expr(I)$ is the set of expressions over I
$Prd(I)$ is the set of predicates over I

Representation A map $Env : X \to Expr(I)$ and a set $PC \subset Prd(I)$.

Assign If `v = x op y`
then update $Env(v)$ with $Env(x)$ op $Env(y)$.

Symbolic Execution: Foundations (Symbols)

X is set of variables, $I \subset X$ is a set of input variables
$Expr(I)$ is the set of expressions over I
$Prd(I)$ is the set of predicates over I

Representation A map $Env : X \to Expr(I)$ and a set $PC \subset Prd(I)$.

Assign If `v = x op y`
then update $Env(v)$ with $Env(x)$ op $Env(y)$.

Branch If branching on `x rel y`
then add $Env(x)$ rel $Env(y)$ to PC.

Symbolic Execution: Foundations (Symbols)

X is set of variables, $I \subset X$ is a set of input variables
$Expr(I)$ is the set of expressions over I
$Prd(I)$ is the set of predicates over I

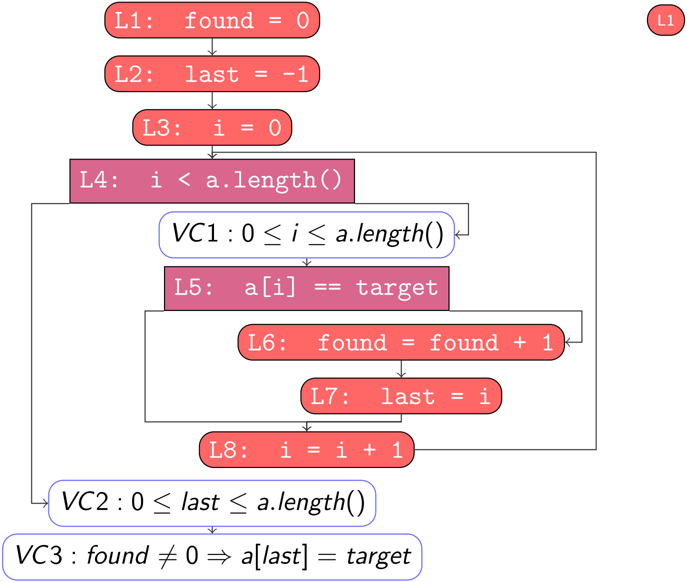
Representation A map $Env : X \rightarrow Expr(I)$ and a set $PC \subset Prd(I)$.

Assign If `v = x op y`
then update $Env(v)$ with $Env(x)$ op $Env(y)$.

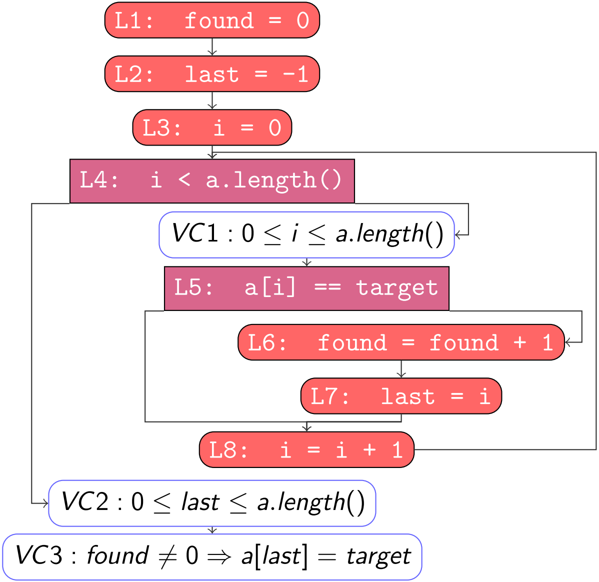
Branch If branching on `x rel y`
then add $Env(x)$ rel $Env(y)$ to PC.

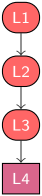
Check Satisfiability check PC.
If *unsat* then discard the trace.

Symbolic Execution: Running Example

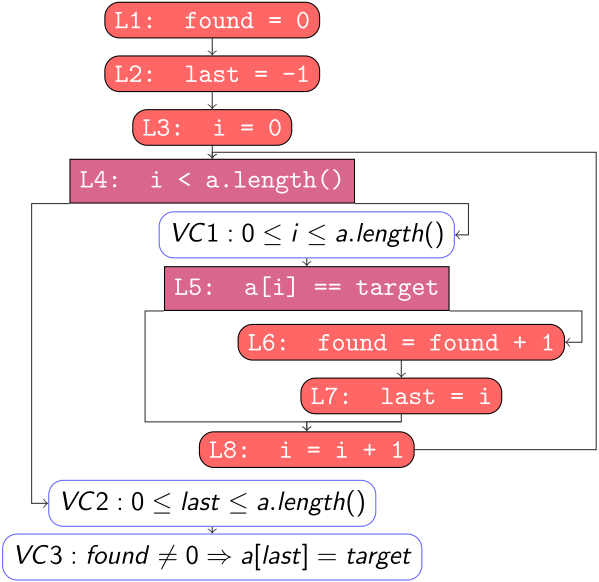


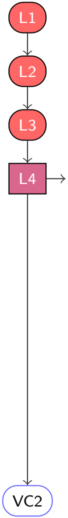
Symbolic Execution: Running Example



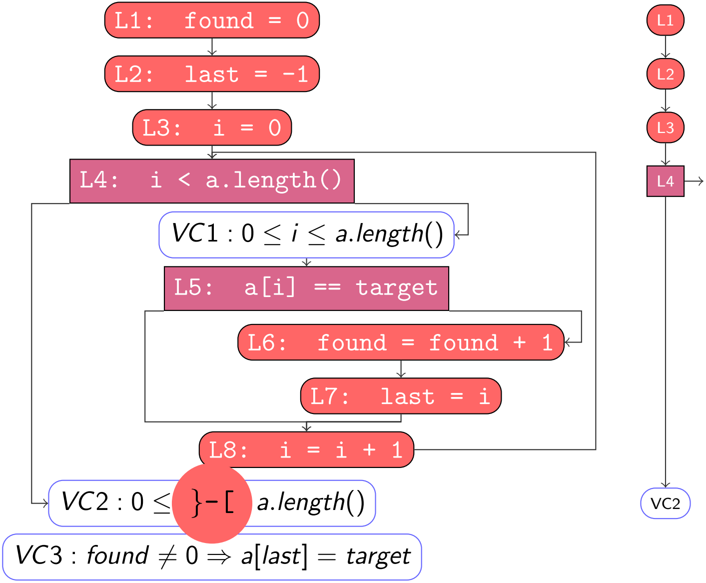


Symbolic Execution: Running Example

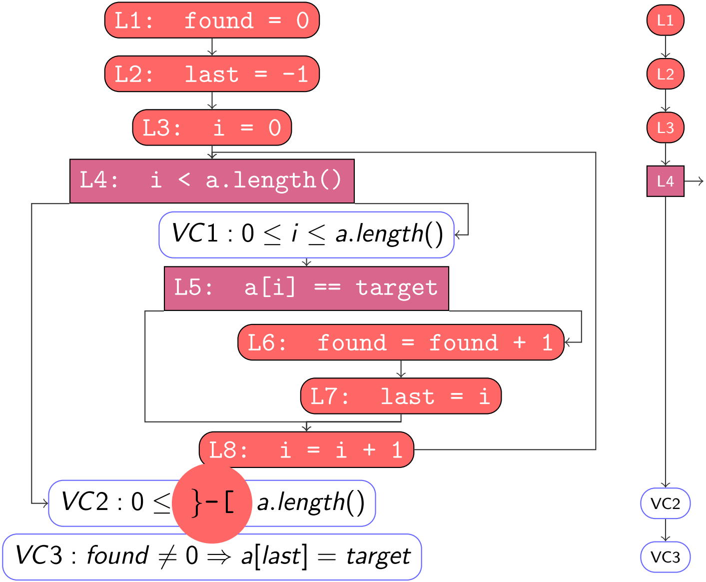




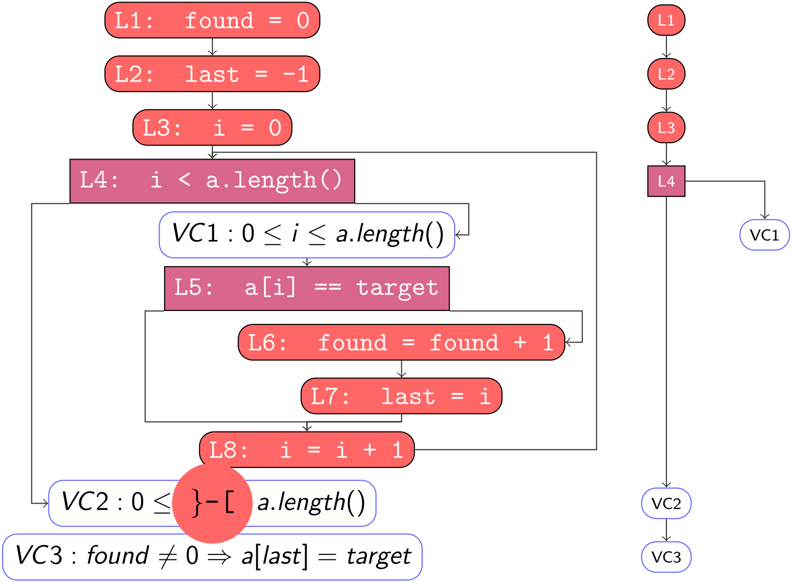
Symbolic Execution: Running Example



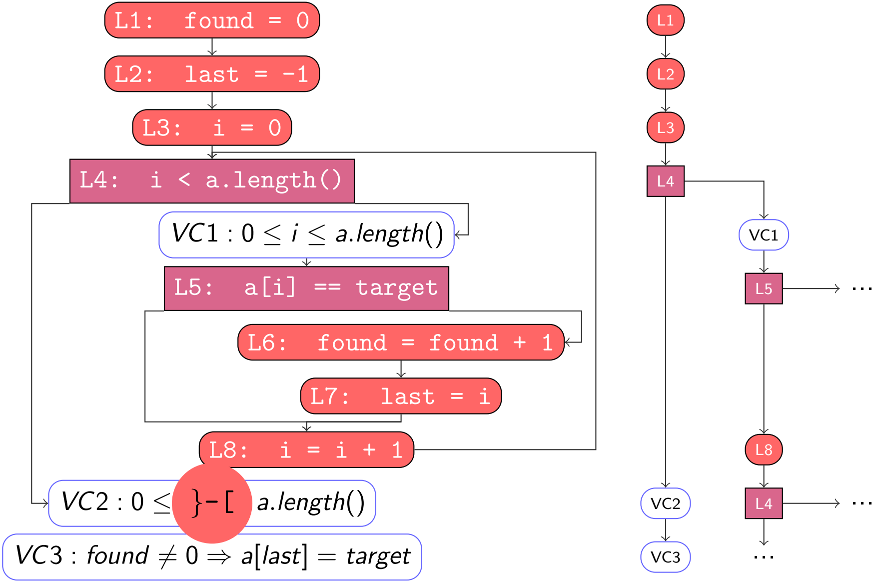
Symbolic Execution: Running Example



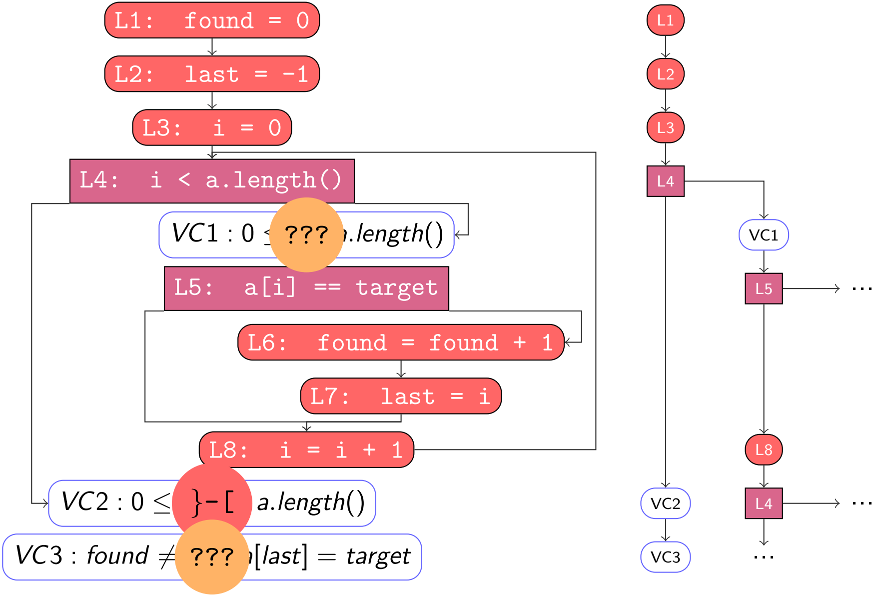
Symbolic Execution: Running Example



Symbolic Execution: Running Example



Symbolic Execution: Running Example



Symbolic Execution: Pros and Cons

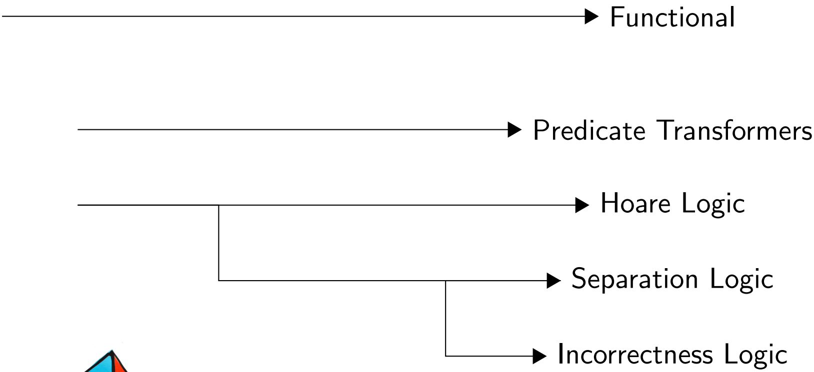
Pros

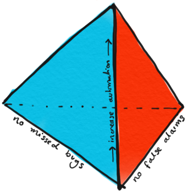
- Counter-examples!
- Concretisation
- Anytime

Cons

- Combinatorial explosion!
- Non-modular
- Need complete program

Human-Assisted Family Tree





Human-Assisted Family Tree

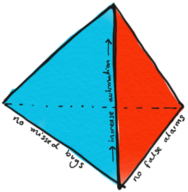
Functional

Predicate Transformers

Hoare Logic

Separation Logic

Incorrectness Logic



Deductive Verification: Foundations (Ideas)

Describe the set of possible states (at a program location) with a *predicate*

Deductive Verification: Foundations (Ideas)

Describe the set of possible states (at a program location) with a *predicate*
Use logic to link these together...

Deductive Verification: Foundations (Ideas)

Describe the set of possible states (at a program location) with a *predicate*
Use logic to link these together...
and use proof by induction for loops!

Deductive Verification: Foundations (Symbols)

Hoare Triples

$$\{Pre\}\ \texttt{Program}\ \{Post\}$$

"If the state of the program meets the precondition (*Pre* is true) then after `Program` has been run the state will meet the postcondition (*Post* is true)"

Deductive Verification: Foundations (Symbols)

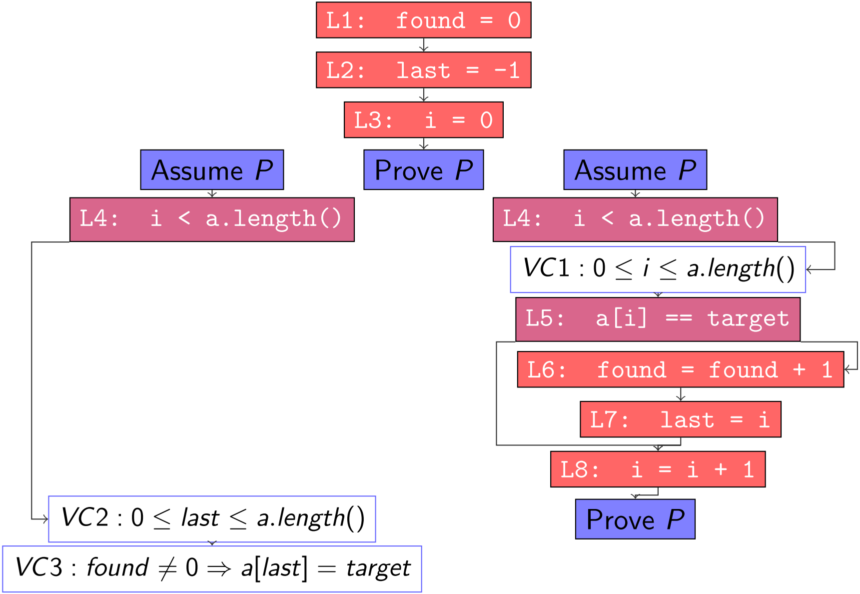
Hoare Triples

$$\{Pre\}\ \texttt{Program}\ \{Post\}$$

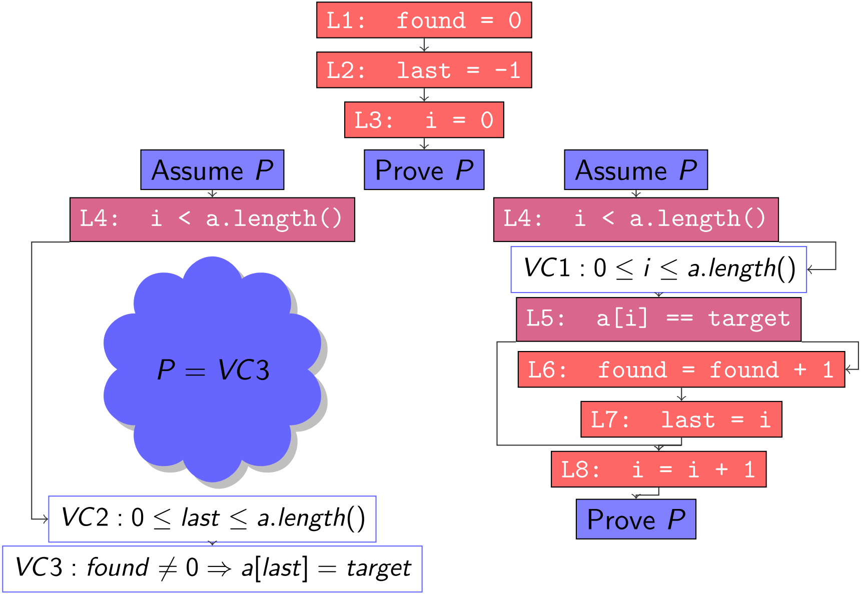
"If the state of the program meets the precondition (*Pre* is true) then after `Program` has been run the state will meet the postcondition (*Post* is true)"

$$\frac{\{Inv \land Cond\}\ \texttt{Body}\ \{Inv\}}{\{Inv\}\ \texttt{while (Cond) Body}\ \{Inv \land \neg Cond\}}$$

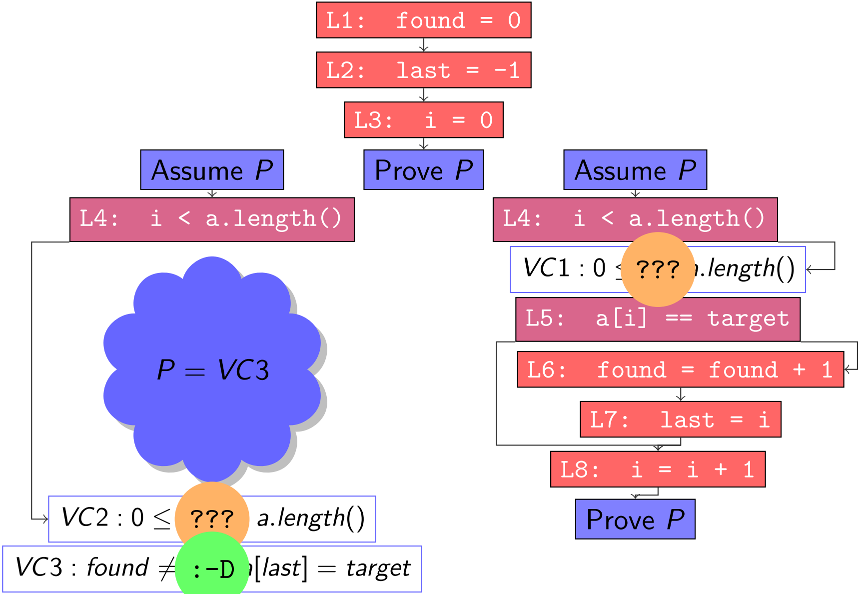
Deductive Verification: Running Example



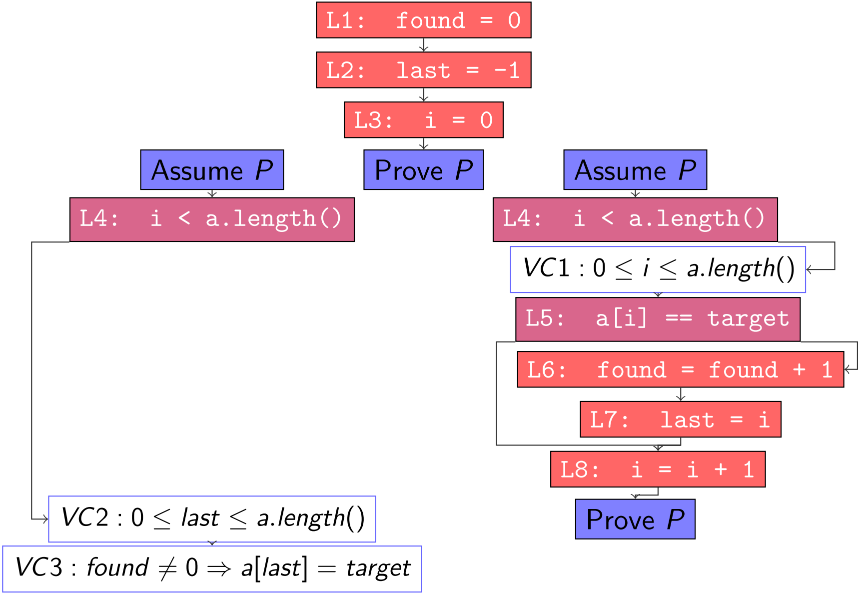
Deductive Verification: Running Example



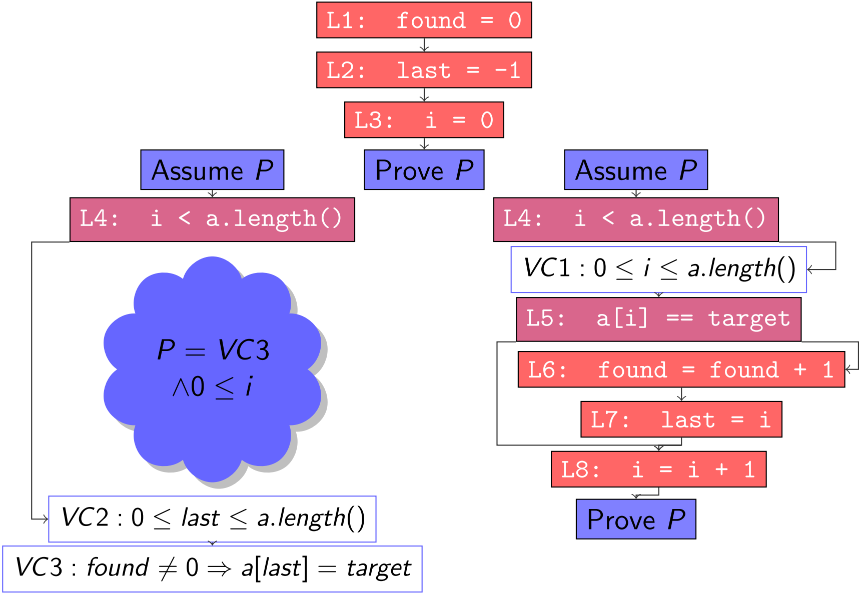
Deductive Verification: Running Example



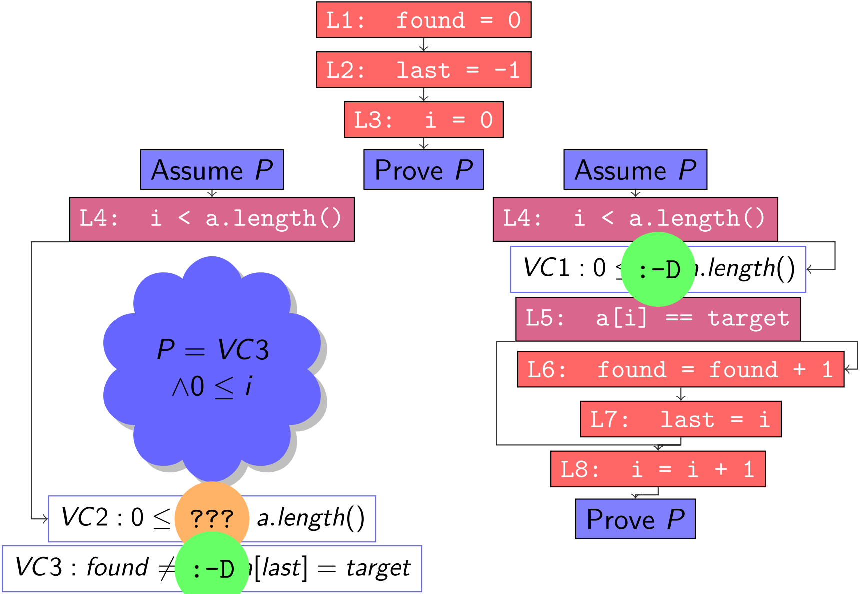
Deductive Verification: Running Example



Deductive Verification: Running Example



Deductive Verification: Running Example



Deductive Verification: Pros and Cons

Pros

- Certainty
- Scalable (compute)
- Incremental

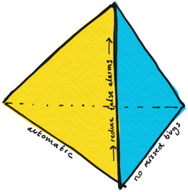
Cons

- Maintainance
- Scalable (human)
- False vs. not provable

Over-Approximate Family Tree

Static Analysis

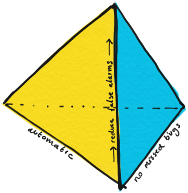
Abstract Interpretation



Over-Approximate Family Tree

Static Analysis

Abstract Interpretation



Abstract Interpretation: Foundations (Ideas)

Want to reason about *all* possible executions...

Abstract Interpretation: Foundations (Ideas)

Want to reason about *all* possible executions...
What if we run the program with sets of possible values?

Abstract Interpretation: Foundations (Ideas)

Want to reason about *all* possible executions...
What if we run the program with sets of possible values?
That's too big and too slow but what if we run the program with representations of sets of possible values?

Abstract Interpretation: Foundations (Symbols)

X is set of variables

$Env = X \to \mathbb{Q}$ is set of program states

$Instr = Env \to Env$ is set of program instructions

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

Representation Set L of representations $\gamma : L \to 2^{Env}$
$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

Representation Set L of representations $\gamma : L \to 2^{Env}$
$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$
$\ll i \in [0, 4], j \in [0, 4], n \in [5, 5], m \in [5, 5] \gg \in L$

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

Representation Set L of representations $\gamma : L \to 2^{Env}$
$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$
$\ll i \in [0,4], j \in [0,4], n \in [5,5], m \in [5,5] \gg \in L$

Transform $T : Instr \times L \to L$ with $f(\gamma(l)) \subseteq \gamma(T(f,l))$

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

Representation Set L of representations $\gamma : L \to 2^{Env}$

$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$

$\ll i \in [0, 4], j \in [0, 4], n \in [5, 5], m \in [5, 5] \gg \in L$

Transform $T : Instr \times L \to L$ with $f(\gamma(l)) \subseteq \gamma(T(f, l))$

$T(\texttt{i=i+1}, \ll \ldots i \in [0, 4] \cdots \gg) = \ll \ldots i \in [1, 5] \cdots \gg$

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

Representation Set L of representations $\gamma : L \to 2^{Env}$
$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$
$\ll i \in [0,4], j \in [0,4], n \in [5,5], m \in [5,5] \gg \in L$

Transform $T : Instr \times L \to L$ with $f(\gamma(l)) \subseteq \gamma(T(f,l))$
$T(\texttt{i=i+1}, \ll \ldots i \in [0,4] \cdots \gg) = \ll \ldots i \in [1,5] \cdots \gg$

Merge $\sqcup : L \times L \to L$ with $l_1 \sqsubseteq l_1 \sqcup l_2,\ l_2 \sqsubseteq l_1 \sqcup l_2$

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

Representation Set L of representations $\gamma : L \to 2^{Env}$
$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$
$\ll i \in [0,4], j \in [0,4], n \in [5,5], m \in [5,5] \gg \in L$

Transform $T : Instr \times L \to L$ with $f(\gamma(l)) \subseteq \gamma(T(f,l))$
$T(\texttt{i=i+1}, \ll \ldots i \in [0,4] \cdots \gg) = \ll \ldots i \in [1,5] \cdots \gg$

Merge $\sqcup : L \times L \to L$ with $l_1 \sqsubseteq l_1 \sqcup l_2,\ l_2 \sqsubseteq l_1 \sqcup l_2$
$\ll \ldots i \in [0,4] \cdots \gg \sqcup \ll \ldots i \in [1,5] \cdots \gg =$
$\ll \ldots i \in [0,5] \cdots \gg$

Abstract Interpretation: Foundations (Symbols)

X is set of variables
$Env = X \to \mathbb{Q}$ is set of program states
$Instr = Env \to Env$ is set of program instructions

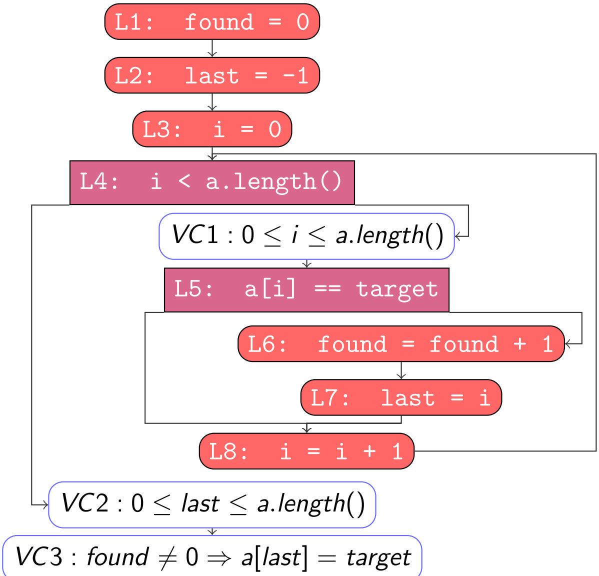
Representation Set L of representations $\gamma : L \to 2^{Env}$
$l_1 \sqsubseteq l_2 \Leftrightarrow \gamma(l_1) \subseteq \gamma(l_2)$
$\ll i \in [0,4], j \in [0,4], n \in [5,5], m \in [5,5] \gg \in L$

Transform $T : Instr \times L \to L$ with $f(\gamma(l)) \subseteq \gamma(T(f,l))$
$T(\texttt{i=i+1}, \ll \dots i \in [0,4] \cdots \gg) = \ll \dots i \in [1,5] \cdots \gg$

Merge $\sqcup : L \times L \to L$ with $l_1 \sqsubseteq l_1 \sqcup l_2,\ l_2 \sqsubseteq l_1 \sqcup l_2$
$\ll \dots i \in [0,4] \cdots \gg \sqcup \ll \dots i \in [1,5] \cdots \gg =$
$\ll \dots i \in [0,5] \cdots \gg$

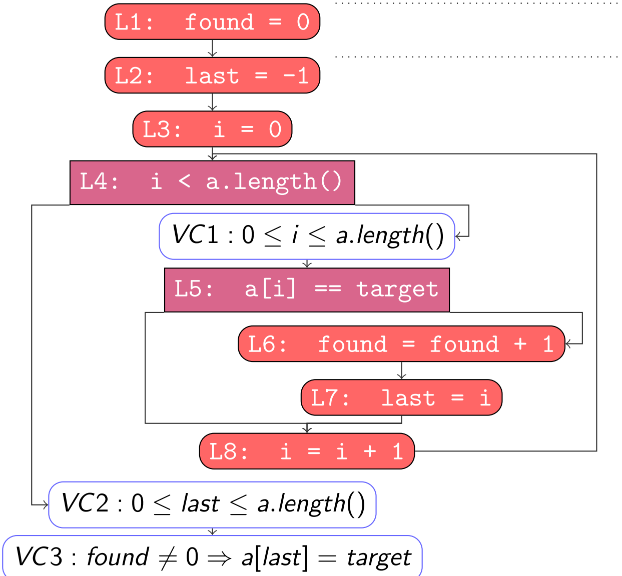
Widen $\nabla : L \times L \to L$ with $l_1 \sqsubseteq l_1 \sqcup l_2,\ l_2 \sqsubseteq l_1 \sqcup l_2$
"guarantees termination"
$\ll \dots i \in [0,4] \cdots \gg \nabla \ll \dots i \in [1,5] \cdots \gg =$
$\ll \dots i \in [0, \text{inf}] \cdots \gg$

Abstract Interpretation: Running Example



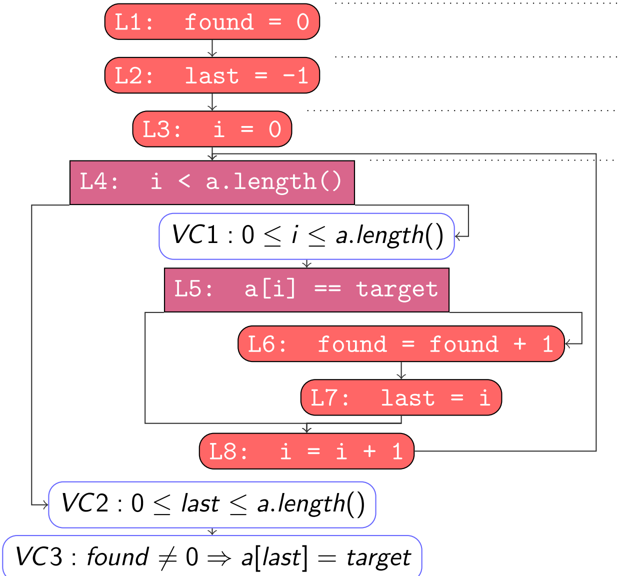
found	last	i

Abstract Interpretation: Running Example



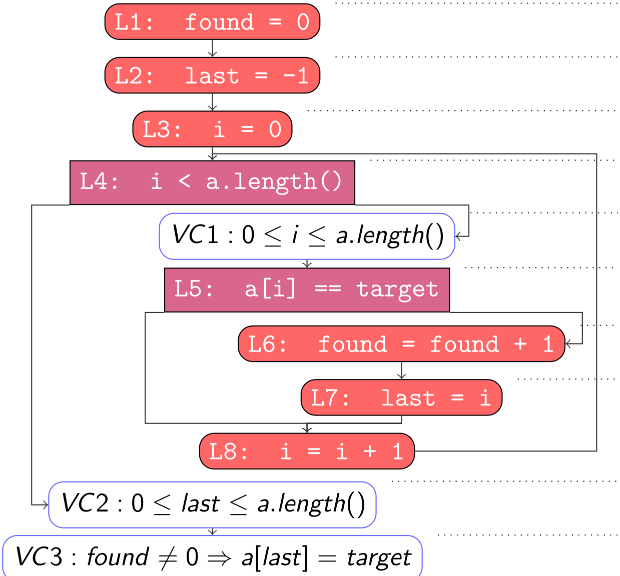
found	last	i
$[0, 0]$		

Abstract Interpretation: Running Example



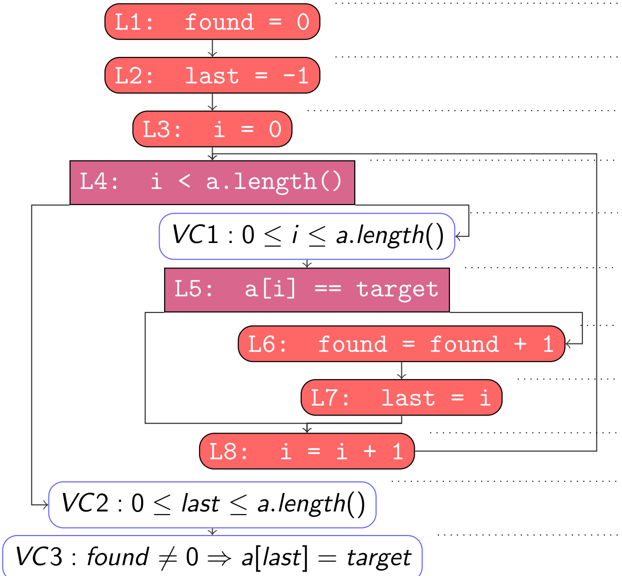
found	last	i
$[0, 0]$		
$[0, 0]$	$[-1, -1]$	
$[0, 0]$	$[-1, -1]$	$[0, 0]$

Abstract Interpretation: Running Example



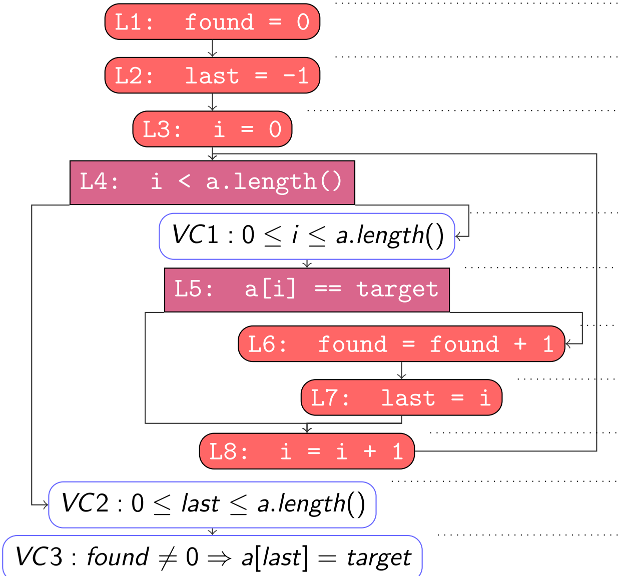
found	last	i
$[0,0]$		
$[0,0]$	$[-1,-1]$	
$[0,0]$	$[-1,-1]$	$[0,0]$
$[0,0]$	$[-1,-1]$	$[0,0]$
$[0,0]$	$[-1,-1]$	$[0,0]$
$[0,0]$	$[-1,-1]$	$[0,0]$
$[1,1]$	$[-1,-1]$	$[0,0]$
$[0,0]$	$[-1,-1]$	$[0,0]$
$[0,0]$	$[-1,-1]$	$[0,0]$

Abstract Interpretation: Running Example



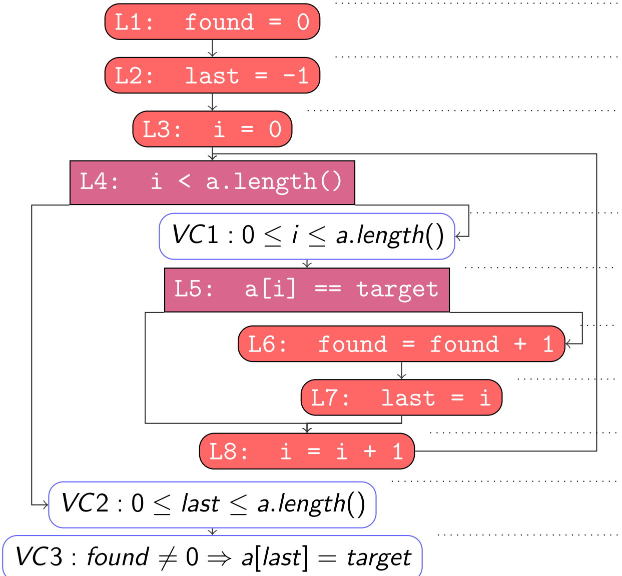
found	last	i
$[0, 0]$		
$[0, 0]$	$[-1, -1]$	
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[1, 1]$	$[-1, -1]$	$[0, 0]$
$[0, 1]$	$[-1, 0]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$

Abstract Interpretation: Running Example



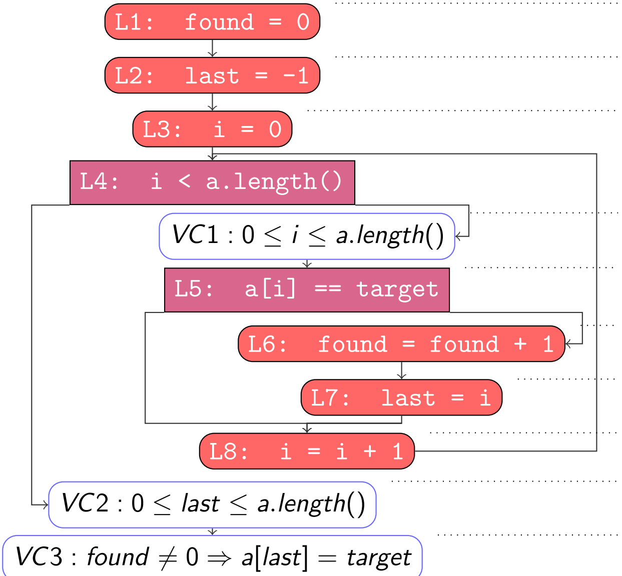
found	last	i
$[0, 0]$	–	–
$[0, 0]$	$[-1, -1]$	–
$[0, 1]$	$[-1, 0]$	$[0, 1]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[1, 1]$	$[-1, -1]$	$[0, 0]$
$[0, 1]$	$[-1, 0]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$
$[0, 0]$	$[-1, -1]$	$[0, 0]$

Abstract Interpretation: Running Example



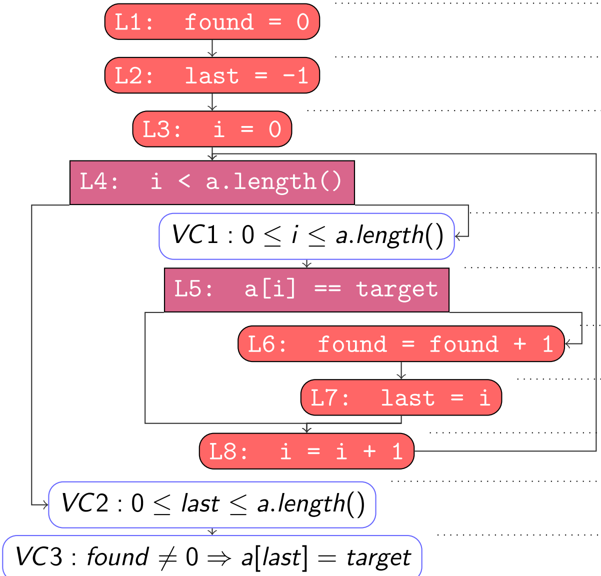
found	last	i
$[0,0]$		
$[0,0]$	$[-1,-1]$	
$[0,1]$	$[-1,0]$	$[0,1]$
$[0,1]$	$[-1,0]$	$[0,1]$
$[0,1]$	$[-1,0]$	$[0,1]$
$[0,1]$	$[-1,0]$	$[0,1]$
$[1,2]$	$[-1,0]$	$[0,1]$
$[0,2]$	$[-1,1]$	$[0,1]$
$[0,1]$	$[-1,0]$	$[0,1]$
$[0,1]$	$[-1,0]$	$[0,1]$

Abstract Interpretation: Running Example



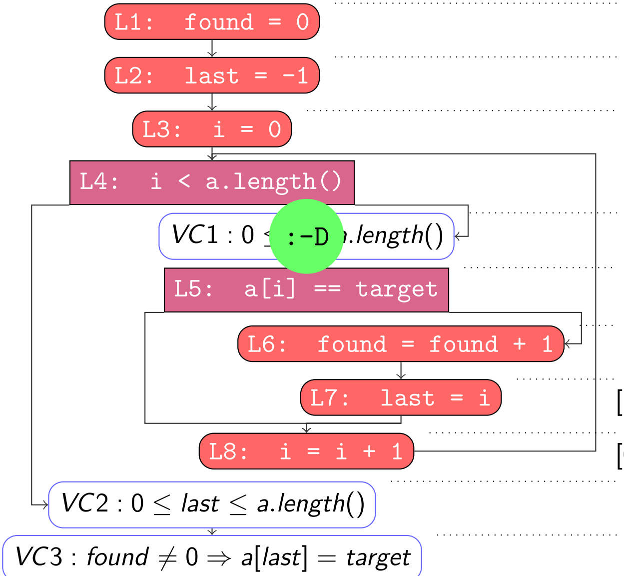
found	last	i
$[0, 0]$		
$[0, 0]$	$[-1, -1]$	
$[0, 2]$	$[-1, 1]$	$[0, 2]$
$[0, 2]$	$[-1, 1]$	$[0, 2]$
$[0, 2]$	$[-1, 1]$	$[0, 2]$
$[0, 2]$	$[-1, 1]$	$[0, 2]$
$[1, 3]$	$[-1, 1]$	$[0, 2]$
$[0, 3]$	$[-1, 2]$	$[0, 2]$
$[0, 2]$	$[-1, 1]$	$[0, 2]$
$[0, 2]$	$[-1, 1]$	$[0, 2]$

Abstract Interpretation: Running Example



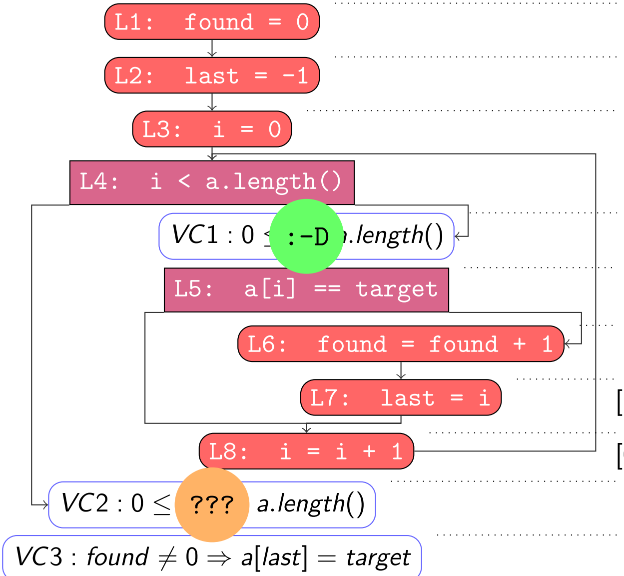
found	last	i
$[0,0]$		
$[0,0]$	$[-1,-1]$	
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[1,x+1]$	$[-1,x-1]$	$[0,x]$
$[0,x+1]$	$[-1,x]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$

Abstract Interpretation: Running Example



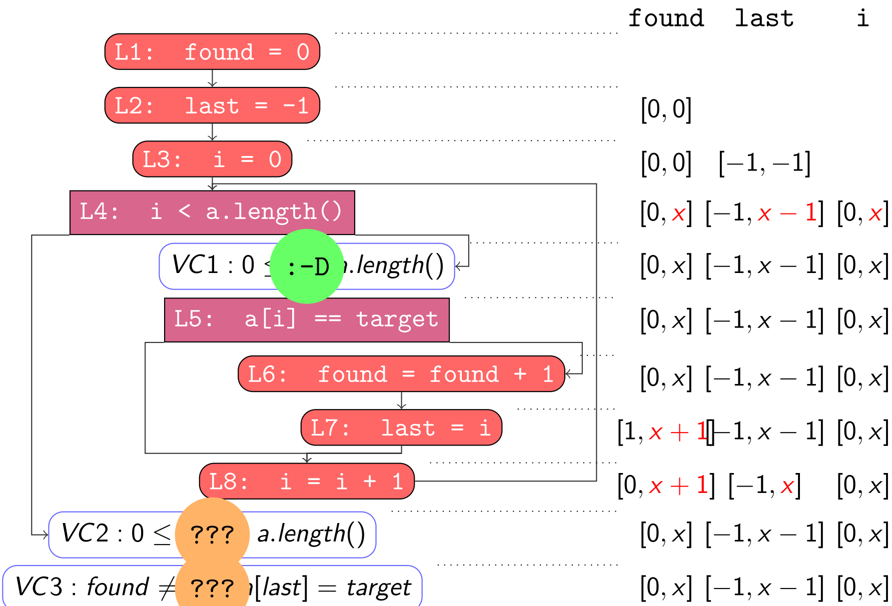
found	last	i
$[0,0]$		
$[0,0]$	$[-1,-1]$	
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[1,x+1]$	$[-1,x-1]$	$[0,x]$
$[0,x+1]$	$[-1,x]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$

Abstract Interpretation: Running Example



found	last	i
$[0,0]$		
$[0,0]$	$[-1,-1]$	
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[1,x+1]$	$[-1,x-1]$	$[0,x]$
$[0,x+1]$	$[-1,x]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$
$[0,x]$	$[-1,x-1]$	$[0,x]$

Abstract Interpretation: Running Example



Abstract Interpretation: Pros and Cons

Pros

- Assuming independence is an overapproximation
- Can discard information about states
- Compositional / modular

Cons

- When you reach $\top$...
- "Yes but why?"
- Widen is *hard*

Process Considerations

Key Trade-Off

No false alarms / No missed bugs / Automatic

Two for free, will a reasonable amount of computation and human effort give you enough of the third?

Considerations:

- What happens if the system doesn't meet the spec?
- Who uses the evidence? For what?
- Can the the system or spec be changed?
- Can the tools be changed?
- Are partial results useful?

Understanding Tool Evaluation

Skip To The Results and ...

Over-Approximate Number of Alarms (proxy for false alarms)

Under-Approximate Benchmarks Solved (proxy for coverage rate)

Human-Approximate LoC of Annotation (proxy for human effort)

Limits of Verification

Biased Personal Opinion

For verification to be "useful",
the specification must be "simpler" than the system.

(What is the spec for *cosine*? What is the spec for `printf`? What is the spec for `getopt`? What is the spec for `strtod`? What is the spec for time & date?)

Conclusion : How to Pick Verification Tools

1. What is your system? What is your specification?
   Are they formal? Could they be?
2. How finished / fixed are they?
   "Writing verified software" vs. "Verifying written software"
3. What kinds of evidence support your goal?
4. Automatic, no missed bugs, no false alarms – pick two!
5. Fit the tool to the process or vica versa or both.

Conclusion : How to Pick Verification Tools

1. What is your system? What is your specification?
   Are they formal? Could they be?
2. How finished / fixed are they?
   "Writing verified software" vs. "Verifying written software"
3. What kinds of evidence support your goal?
4. Automatic, no missed bugs, no false alarms – pick two!
5. Fit the tool to the process or vica versa or both.

Thank you for your time and attention.

Resources

- These slides:
  http://polgreen.github.io/pdfs/pyramid-slides.pdf
- Tutorial paper:
  http://polgreen.github.io/pdfs/pyramid-paper.pdf
- Tutorial on youtube:
  https://youtu.be/BlGZuQIESRU?si=qEzNGt6wvqtq91m1

Coverity scan: Static analysis. `https://scan.coverity.com/`, accessed: 2024-04-10

CPPCheck: A tool for static c/c++ code analysis. `https://cppcheck.sourceforge.io/`, accessed: 2024-04-10

CREST: Concolic test generation tool for c. `https://www.burn.im/crest/`, accessed: 2020-22-07

FindBugs. `http://findbugs.sourceforge.net/`, accessed: 2020-22-07

Fortify static code analyzer. `https://www.opentext.com/products/fortify-static-code-analyzer`, accessed: 2024-04-10

MALPAS software static analysis toolset. `http://malpas-global.com/`, accessed: 2024-04-10

- PolySpace Code Prover. https://www.mathworks.com/products/polyspace-code-prover.html, accessed: 2020-22-07
- SPARK. https://www.adacore.com/about-spark, accessed: 2024-04-10
- Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
- Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011)
- Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex

systems programs. In: OSDI. pp. 209–224. USENIX Association (2008)

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: ESOP. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005)

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c - A software analysis perspective. In: SEFM. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012)

- Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI. pp. 234–245. ACM (2002)
- Havelund, K.: Java pathfinder, A translator from java to promela. In: SPIN. Lecture Notes in Computer Science, vol. 1680, p. 152. Springer (1999)
- Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: SPIN. Lecture Notes in Computer Science, vol. 3639, pp. 25–26. Springer (2005)
- Johnson, S.C.: Lint, a c program checker. In: COMP. SCI. TECH. REP. pp. 78–1273 (1978)
- Kettl, M., Lemberger, T.: The static analyzer infer in SV-COMP (competition contribution). In: TACAS (2). Lecture Notes in Computer Science, vol. 13244, pp. 451–456. Springer (2022)

- Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer (2014)
- Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (Dakar). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
- Logozzo, F.: Practical specification and verification with code contracts. In: HILT. pp. 7–8. ACM (2013)
- Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)

- Norell, U.: Dependently typed programming in agda. In: Advanced Functional Programming. Lecture Notes in Computer Science, vol. 5832, pp. 230–266. Springer (2008)
- Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: CADE. Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer (1992)
- Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: TAP. Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer (2008)