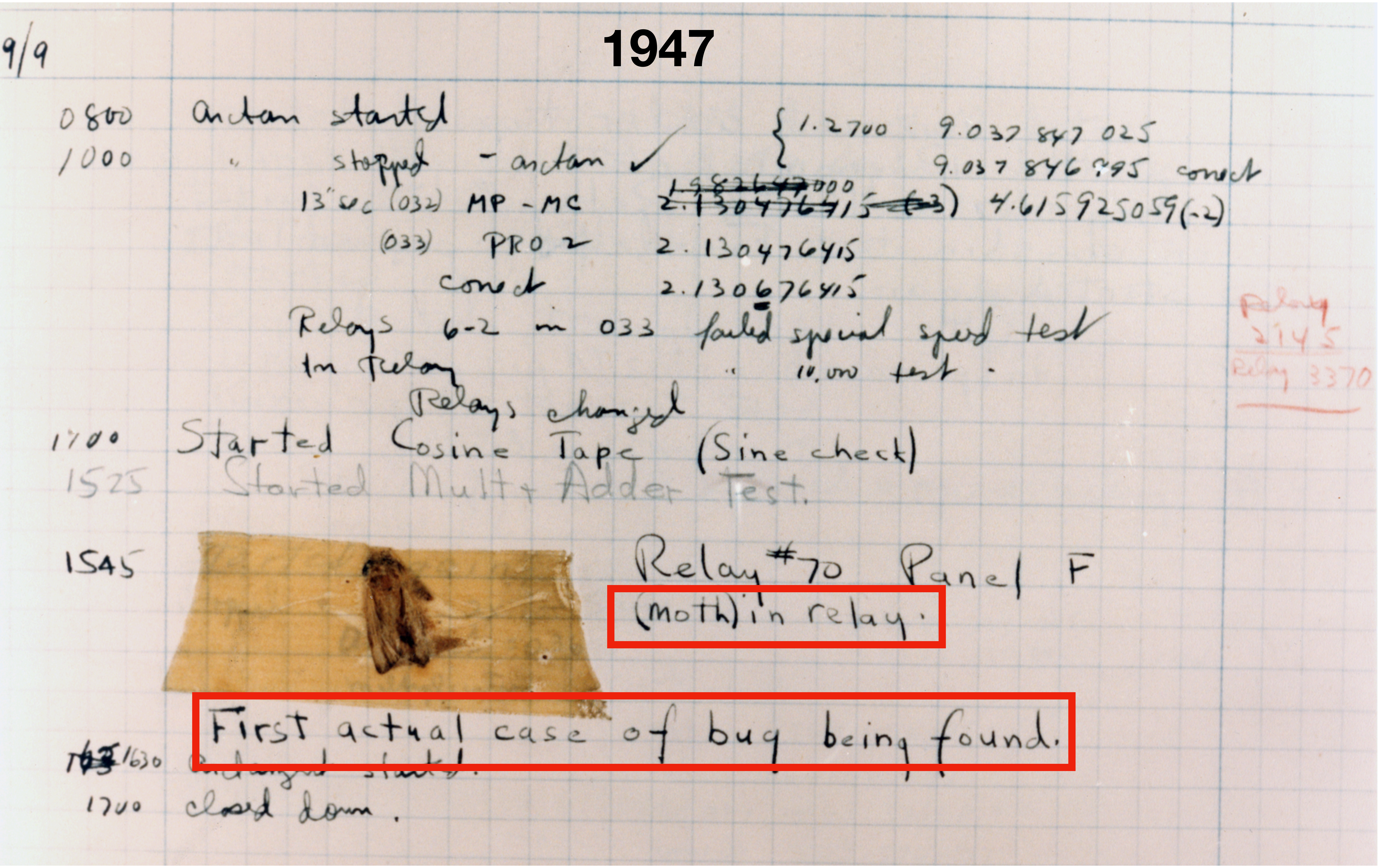
Program analysis: from proving correctness to proving incorrectness

Roberto Bruni, Roberta Gori
(University of Pisa)

Lecture #01

BISS 2024
March 11-15, 2024

Bugs



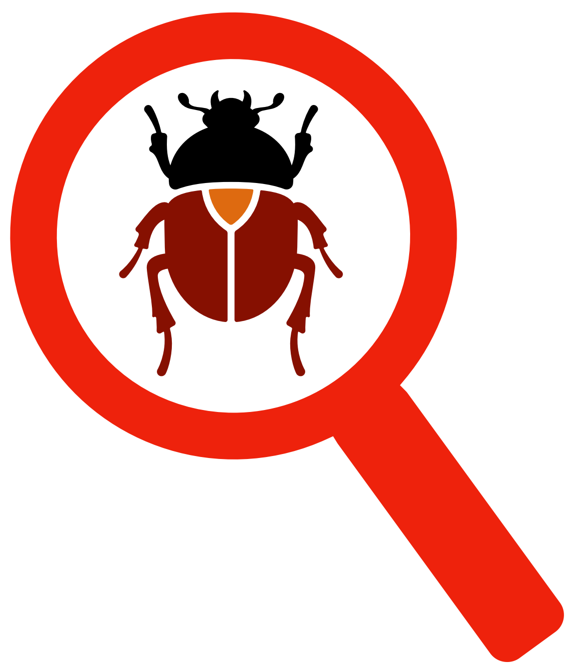
WIKIPEDIA
The Free Encyclopedia

A **software bug** is an error, flaw or fault in the design, development, or operation of computer software that causes it to produce an incorrect or unexpected result

Software Verification

Correctness — the aim is to prove the absence of bugs

Incorrectness — the aim is to prove the presence of bugs



The need for verification

Friday, 24th June [1949]

Checking a large routine by Dr A. Turing.

How can one check a routine in the sense of making sure that it is right?

Ariane 5 Rocket Explosion (1996)

Caused due to numeric overflow error

Attempt to fit 64-bit format data into 16-bit space

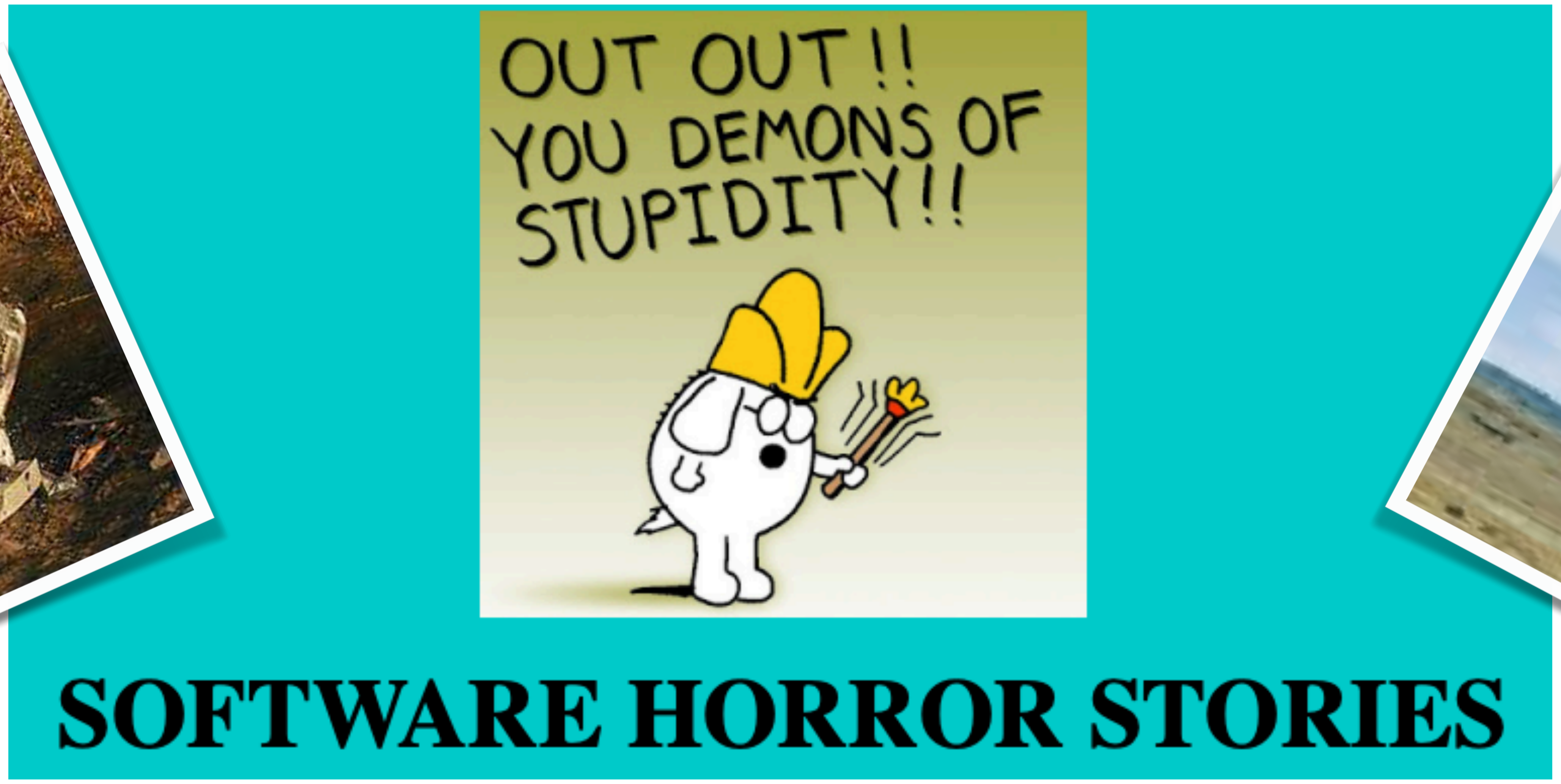
Cost: $100M for loss of mission

Multi-year set back to the Ariane program

Read more at:
https://www.bugsnag.com/blog/bug-day-ariane-5-disaster/

Unfortunately

It was one of the most serious but not the only one….



https://www.cs.tau.ac.il/~nachumd/horror.html

Toyota unintended acceleration
4 people died

Boeing 747 Max Crashes
350 people died

Costs of SW bugs

Knight Capital Trading Glitch (2012)
$ 440 M

Nissan Airbag Malfunction (2014)
1 Million Vehicles Recalled

CISION PR Newswire (2020): SW bugs cost $ 61 Billion loss in productivity annually.

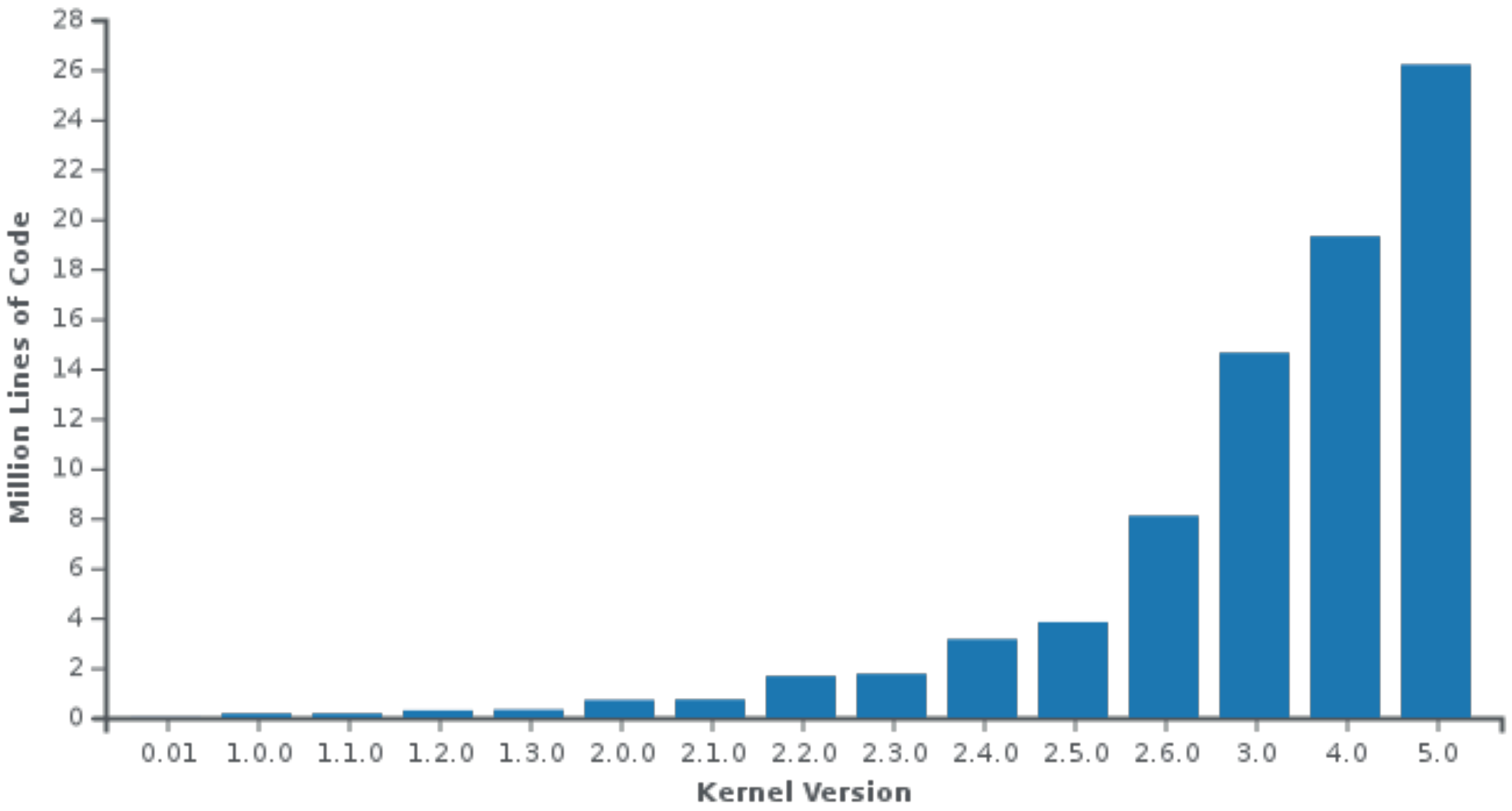
Software Fails Watch (Tricentis, 2017): SW bugs lead to $ 1.7 Trillion revenue lost.

https://www.prnewswire.com/news-releases/study-software-failures-cost-the-enterprise-software-market-61b-annually-301066579.html

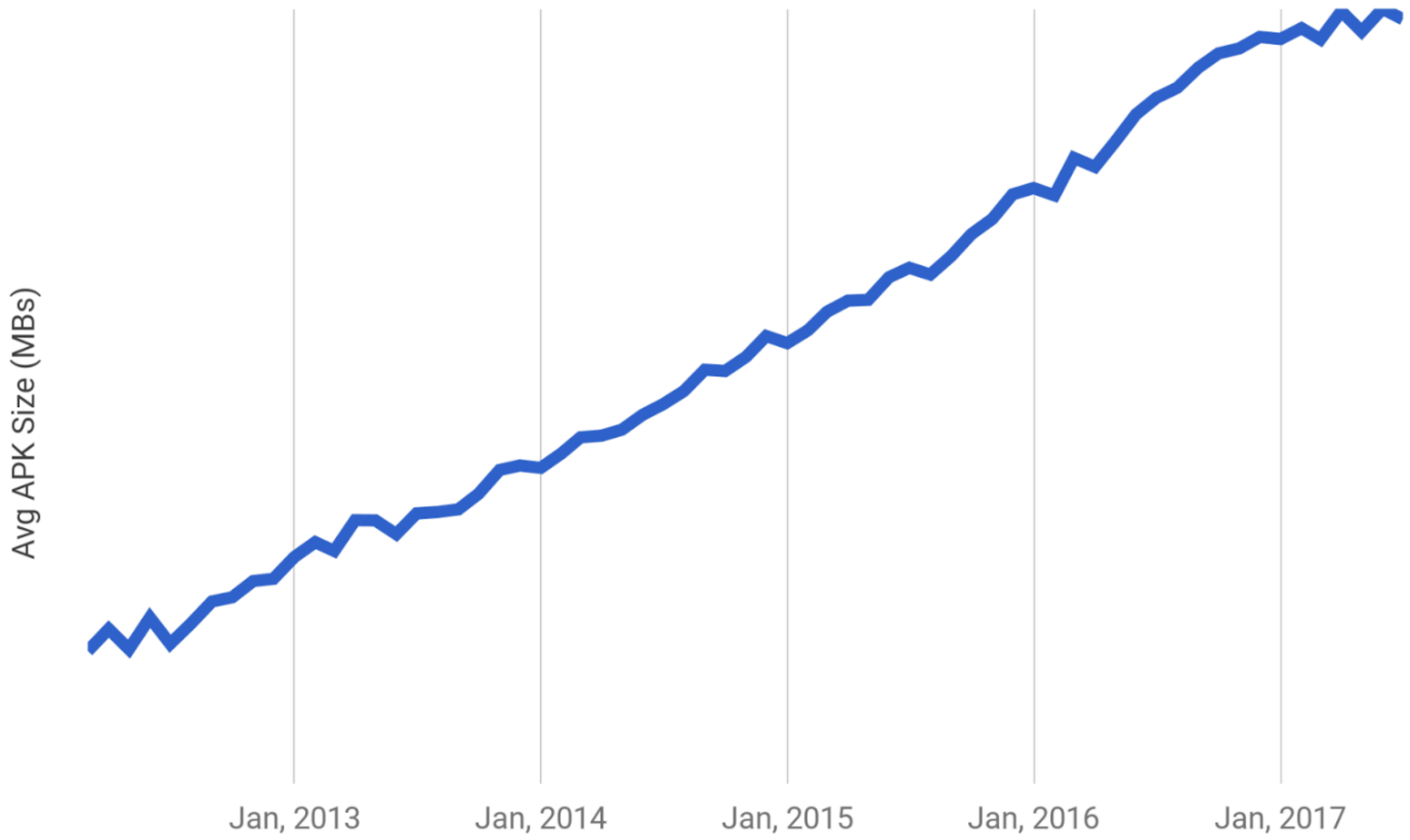
https://www.tricentis.com/news/tricentis-software-fail-watch-finds-3-6-billion-people-affected-and-1-7-trillion-revenue-lost-by-software-failures-last-year/

Complexity of programs

Size of Linux Kernel



Avg. Size of Android Apps



always increasing!

The main question

Will our program behave as we intended?

We need to analyse all executions of the program

The semantics of a program is a description of its run-time behaviors

Checking if a software will run as intended is equivalent to checking if the code satisfies a (semantic) property of interest

Success stories

A long time before success

Computer-assisted verification is an old idea

- Turing, 1948
- Floyd-Hoare logic, 1969

Success in practice: only from the mid-1990s

- Importance of the *increase of performance of computers*

A first success story:

- Paris metro line 14, using *Atelier B* (1998, refinement approach)

Other Famous Success Stories

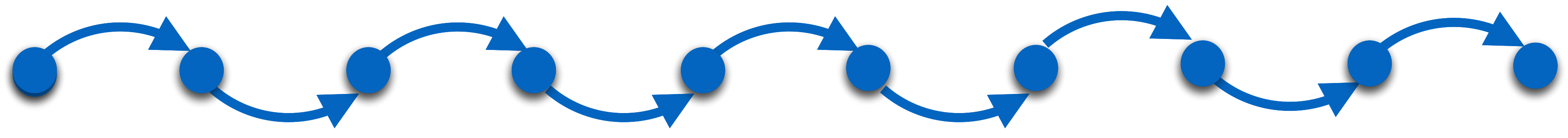
- Flight control software of A380: *Astree* verifies absence of run-time errors (2005, abstract interpretation)
  `http://www.astree.ens.fr/`
- Microsoft's hypervisor: using Microsoft's *VCC* and the *Z3* automated prover (2008, deductive verification)
  `http://research.microsoft.com/en-us/projects/vcc/`
  More recently: verification of PikeOS
- Certified C compiler, developed using the *Coq* proof assistant (2009, correct-by-construction code generated by a proof assistant)
  `http://compcert.inria.fr/`
- L4.verified micro-kernel, using tools on top of *Isabelle/HOL* proof assistant (2010, Haskell prototype, C code, proof assistant)
  `http://www.ertos.nicta.com.au/research/l4.verified/`

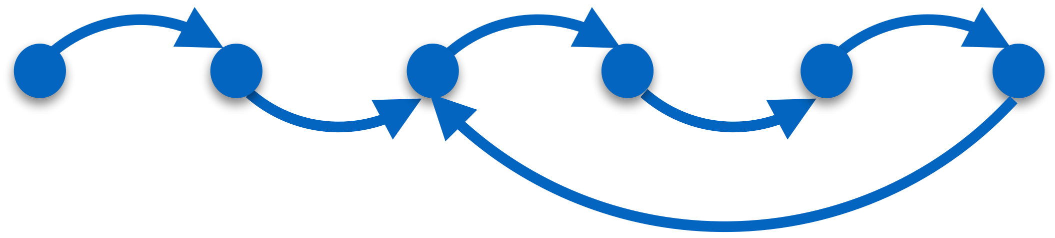
Forward semantics for deterministic programs

We start from input state σ and we want to characterise the reachable output states

A program

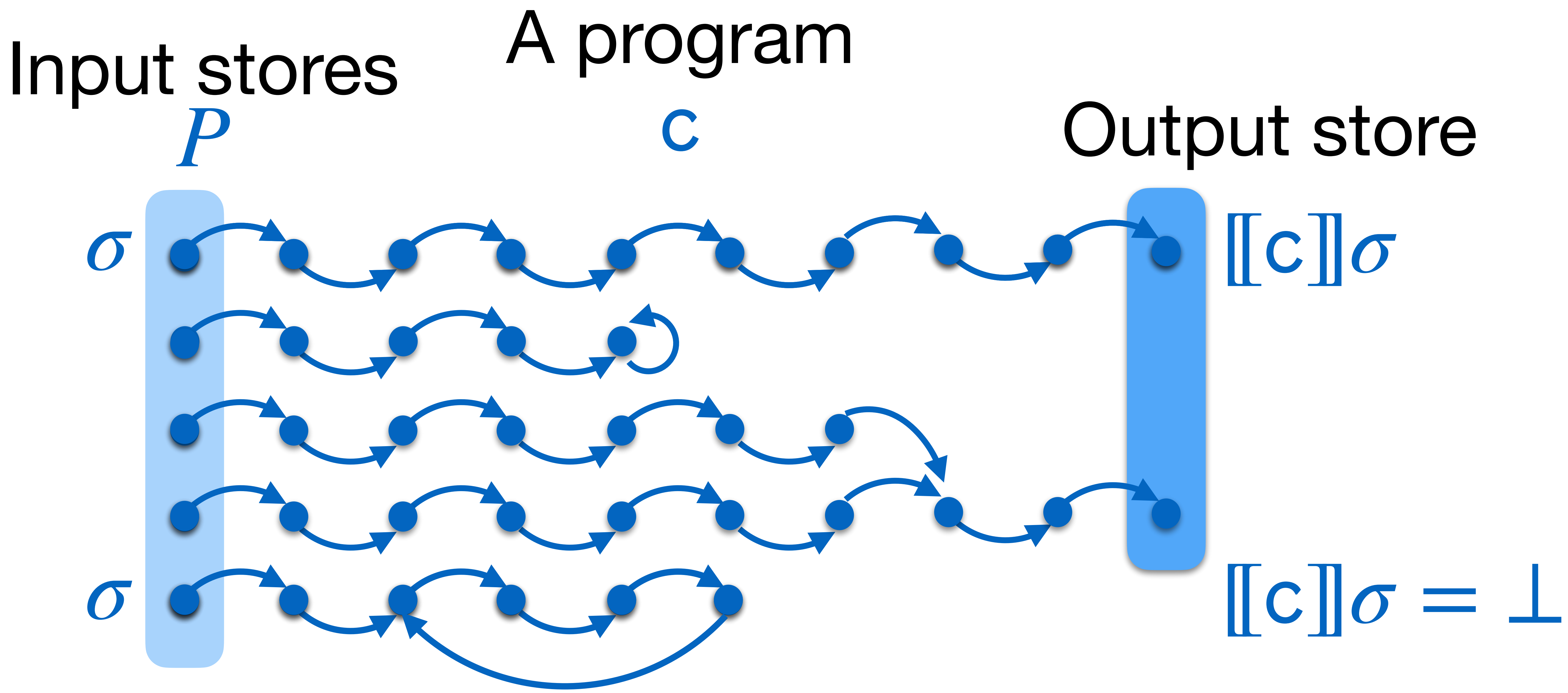
Input store c Output store

σ  $[\![c]\!]\sigma$

σ  $[\![c]\!]\sigma = \bot$ Non terminating execution

Denotational semantics $[\![c]\!] : \Sigma \to \Sigma_\bot$

Collecting semantics for deterministic programs



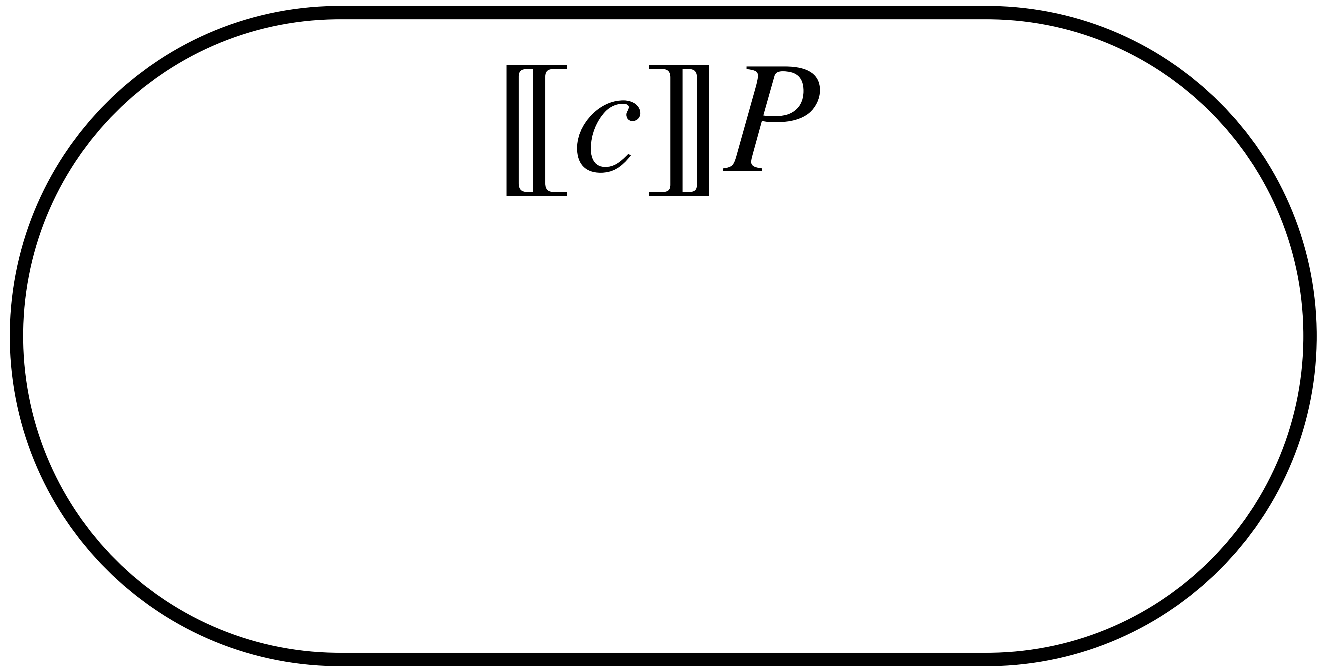
$$[\![\mathsf{c}]\!]P = \bigcup_{\sigma \in P} [\![\mathsf{c}]\!]\sigma$$

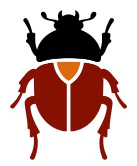
Denotational semantics $[\![\mathsf{c}]\!] : \Sigma \to \Sigma_{\bot}$

Collecting semantics $[\![\mathsf{c}]\!] : \wp(\Sigma) \to \wp(\Sigma)$

Ideal exact analysis

$$[\![c]\!] : \wp(\Sigma) \to \wp(\Sigma)$$



 $\in^{?} [\![c]\!]P$

semantic property of a program: a property about $[\![c]\!]$

$$\mathscr{P}(c) \equiv \forall P \,.\, \forall \sigma \in [\![c]\!]P \,.\, \sigma(x) \neq 0$$

Undecidability in the way

non trivial property:

- there exists a program c such that $\mathscr{P}(c)$ holds true
- and there exists also some program c such that $\mathscr{P}(c)$ is false

Rice theorem.

Let $\mathscr{P}(c)$ be a **non trivial** semantic property of programs c.
There exists no **algorithm** such that, **for every program** c,
it returns true **if and only if** $\mathscr{P}(c)$ holds true

no analysis method that is automatic, universal, exact !

For some program…

$$\mathscr{P}(c) \equiv \forall P \neq \varnothing \,.\, \exists \sigma \in [\![c]\!]P \,.\, \sigma(x) \neq 0$$

$c \triangleq$

```
x := 1;
```

and for some other program...

$$\mathscr{P}(c) \equiv \forall P \neq \emptyset \,.\, \exists \sigma \in [\![c]\!]P \,.\, \sigma(x) \neq 0$$

$c \triangleq$

```
while (n>1) {
    n := n+1;
    x := 0;
}
x := 1;
```

❌

but for Collatz's conjecture?

$$\mathscr{P}(c) \equiv \forall P \neq \emptyset \,.\, \exists \sigma \in [\![c]\!]P \,.\, \sigma(x) \neq 0$$

$c \triangleq$

```
while (n>1) {
    if (even(n)) { n := n/2; }
    else { n:= 3n+1; }
} % does it terminate for any value of n?
x := 1;
```

As of 2020, the conjecture has been checked by computer for all starting values up to $2^{68} \approx 10^{20}$.

Limitations of the analysis

no analysis method that is automatic, universal, exact !

We need to give something up:

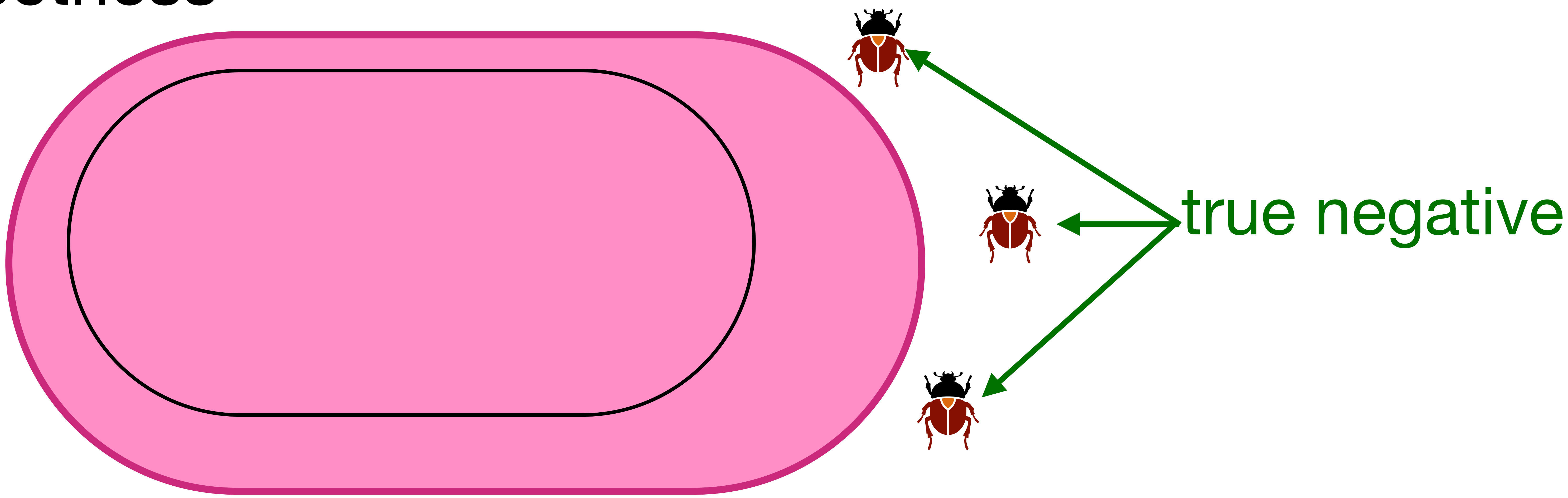
automation: machine-assisted techniques

the **universality** “for all programs”:
targeting only a restricted class of programs

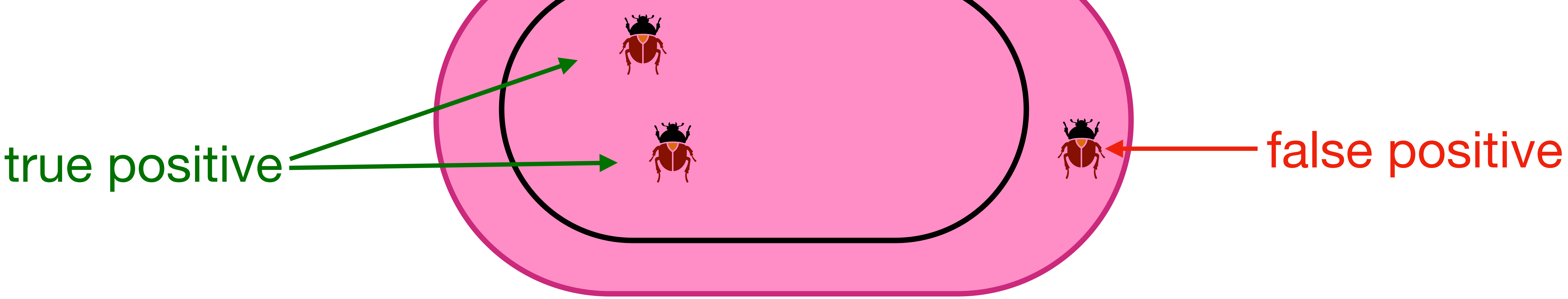
claim to find **exact** answers: introduce approximations

Over approximations

Good for proving correctness



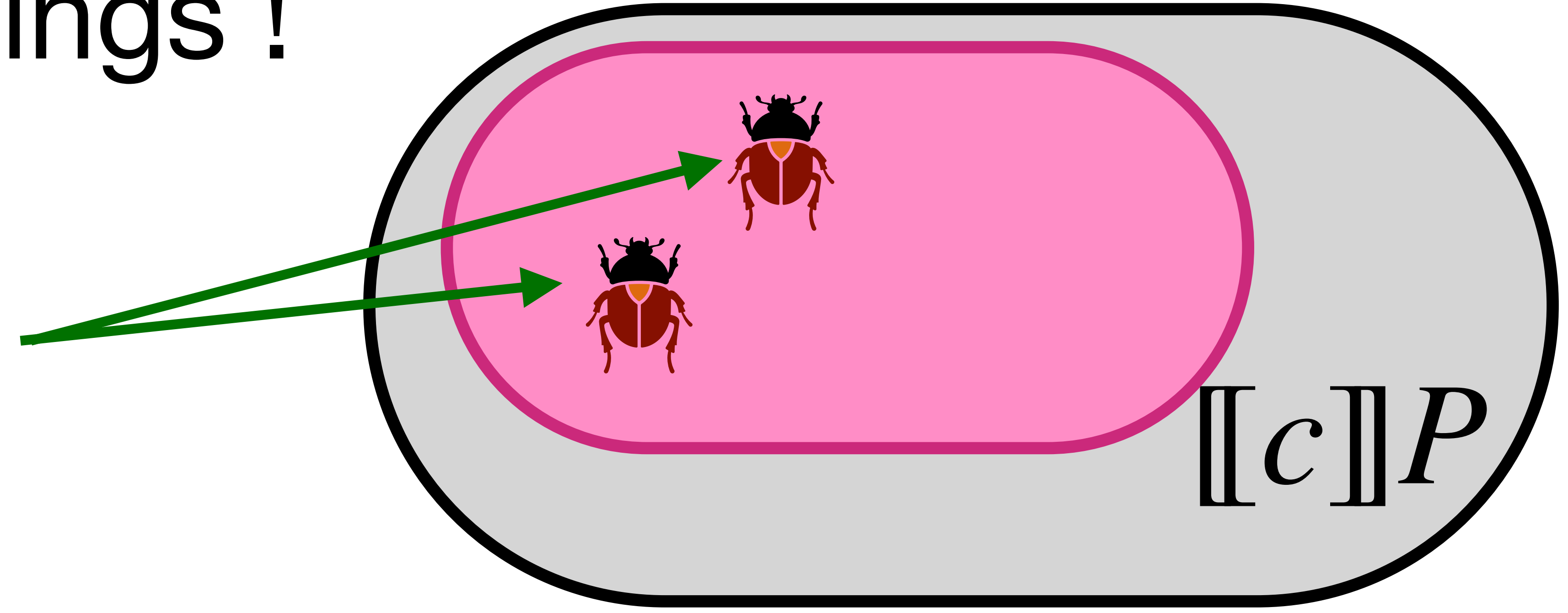
Bad for bug-findings !



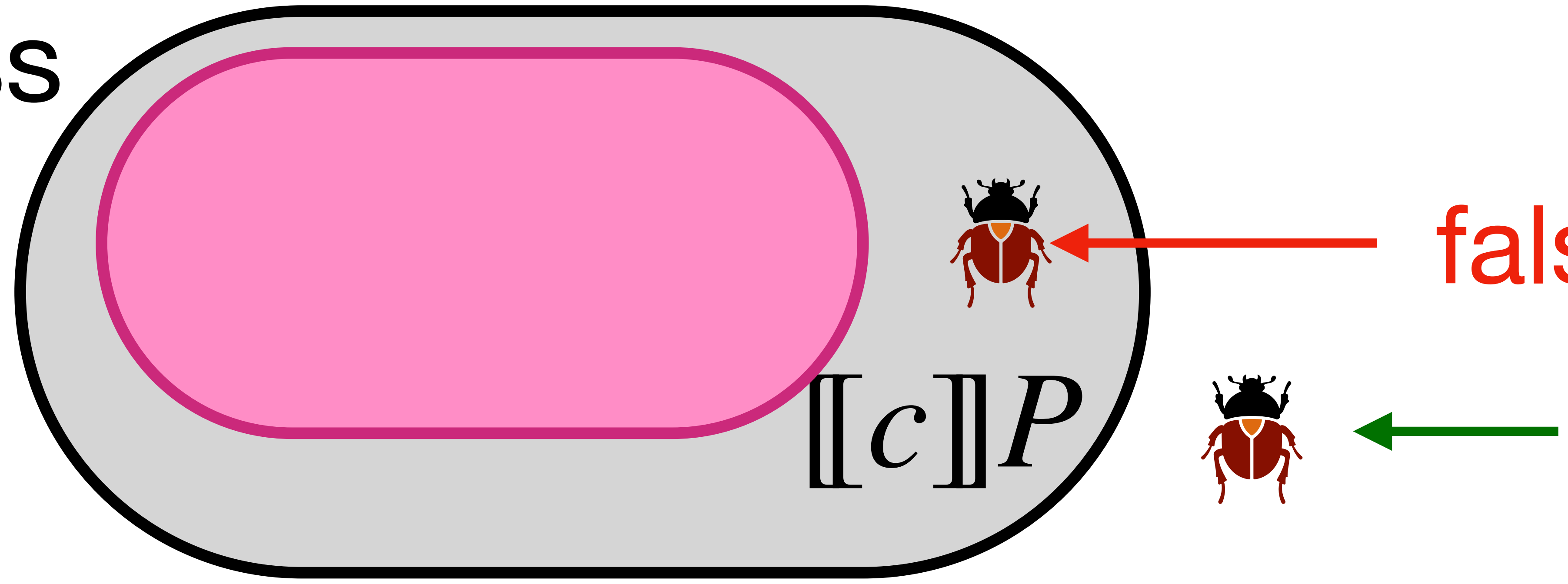
Under approximations

Good for bug-findings !

true positive



Bad for proving correctness



false negative

true negative

Comparison

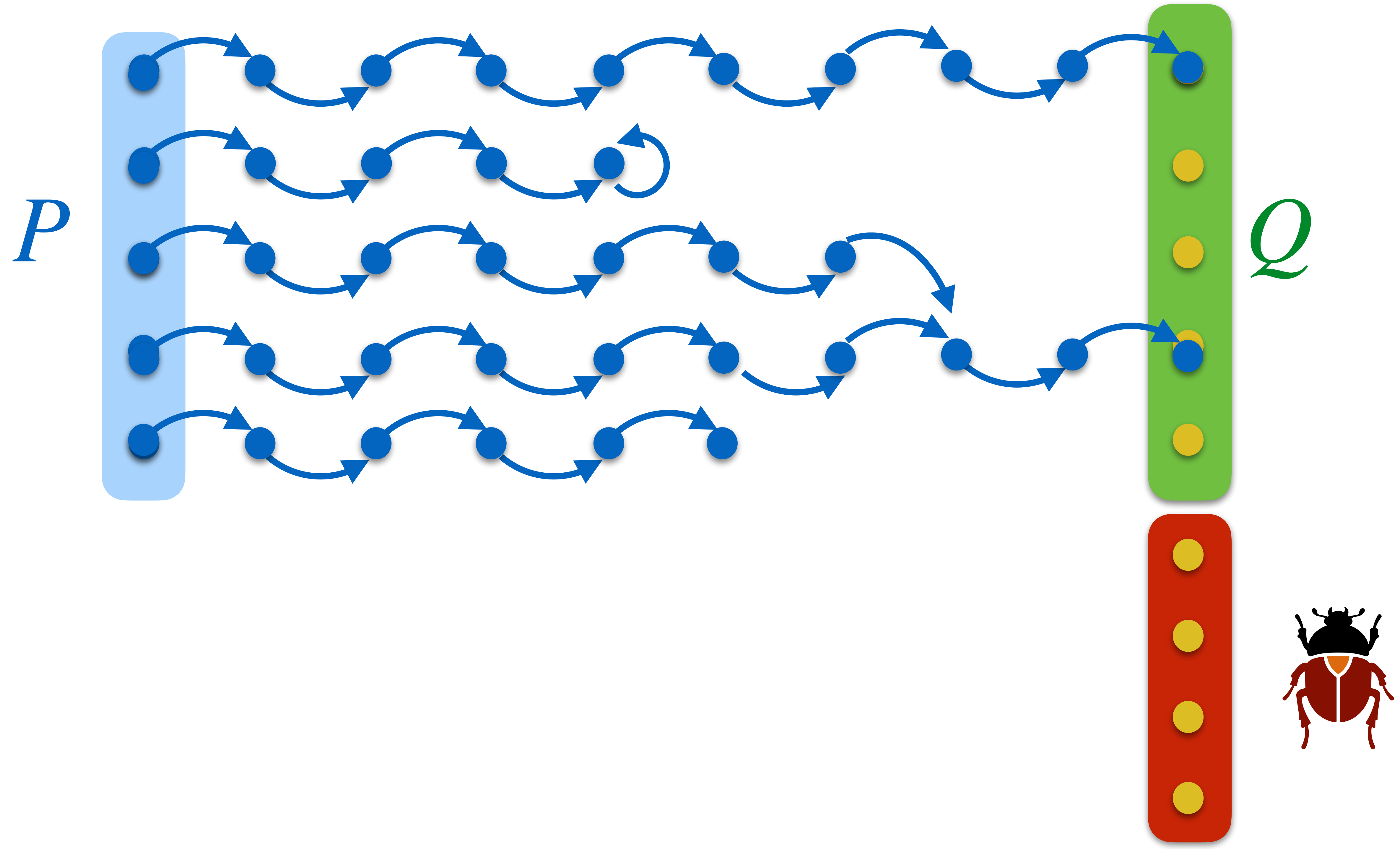
	Automatic	Over-approximation	Under-approximation
Testing	Yes	No	Yes
Machine-assisted Verification	Yes/No	Yes/No	Yes/No
Bounded model checking	Yes	No	Yes
Abstract Interpretation	Yes	Yes	No

Correctness: forward approach

A program

c

$[\![c]\!]P \subseteq Q$



$\forall \sigma \in P \,.\, [\![c]\!]\sigma$ either does not terminate or terminates in Q

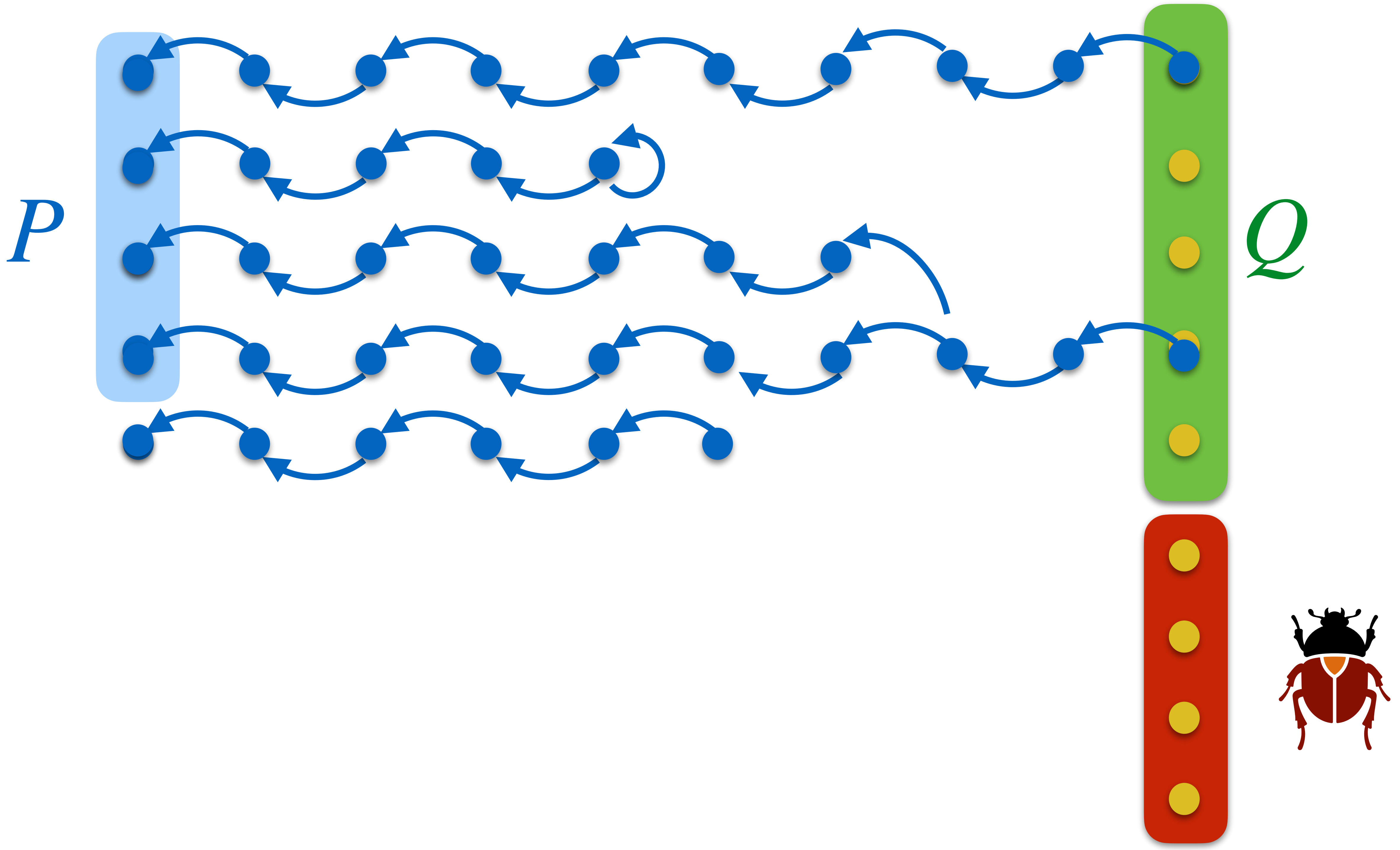
Correctness: backward approach

A program

c

$P \subseteq wlp(c, Q)$

$[\![c]\!]P \subseteq Q$



Dijkstra's weakest liberal precondition

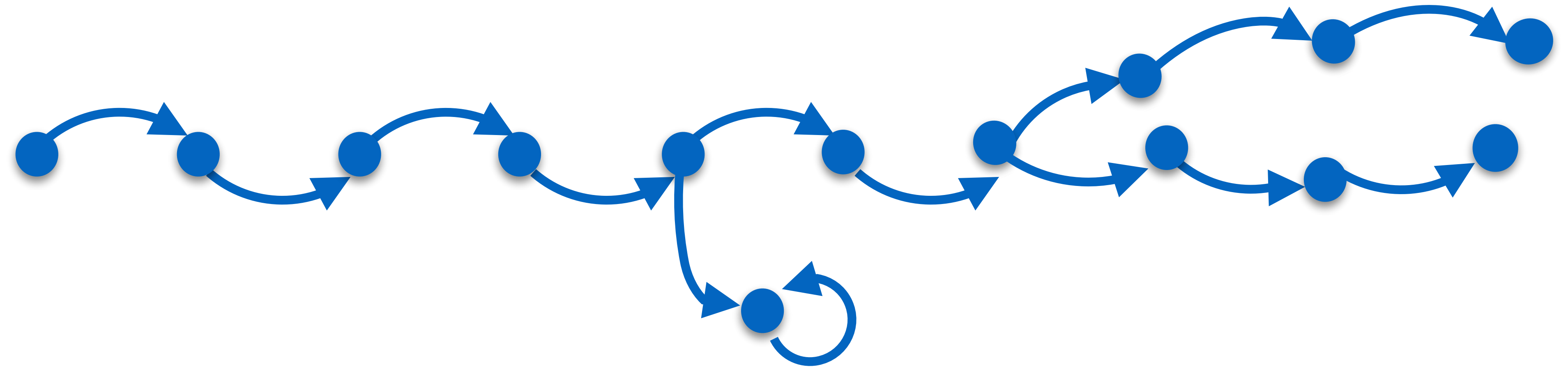
$wlp(c, Q) = \{\sigma \mid [\![c]\!]\{\sigma\} \subseteq Q\}$

Nondeterministic programs

Some programs may exhibit nondeterministic behaviour
(lack of information, approximation, programming constructs $c_1 + c_2$)

A program c

σ



$[\![c]\!] : \Sigma \to \wp(\Sigma)$

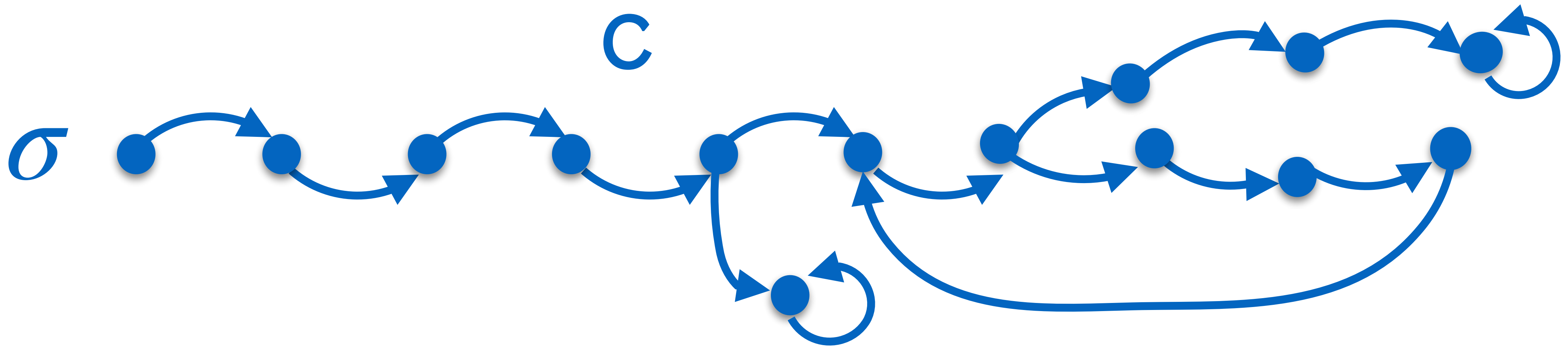
$[\![c]\!]P \subseteq Q$

$P \subseteq wlp(c, Q)$

all the outputs starting from $\sigma \in P$ either do non terminate or terminate in Q

An example: non-termination analysis

Given a program c and an input store σ does $[\![c]\!]\sigma = \emptyset$?



Using over-approximation: we try to prove $[\![c]\!]\sigma \subseteq \emptyset$

Non termination

Using under-approximation: we try to prove $[\![c]\!]\sigma \supseteq Q$ for some $Q \neq \emptyset$

Termination

What we will see

	Forward	Backward	Over-approximation	Under-approximation
Hoare Logic (HL)	X		X	
Incorrectness Logic (IL)	X			X
Locally Complete Logic (LCL)	X		X	X
Necessary Condition (NC)		X	X	
Sufficient Incorrectness Logic (SIL)		X		X
Separation Logic (SL)	X		X	
Incorrectness SL	X			X
Separation SIL		X		X
UNTer	X	X		X

Questions

Question 1

Let $c \triangleq (z := x) + (z := y)$

and let $P \triangleq (x = y = 0)$

What is $[\![c]\!]P$?

$$(x = y = z = 0)$$

Question 2

Let $c \triangleq \text{if } x < y \text{ then } x := y \text{ else (while true do skip)}$

and let $Q \triangleq (x = y = 0)$

What is $wlp(c, Q)$?

$$(x \geq y \vee y = 0)$$