# Introduction to Formal Verification of Programs: Theory and Practice

Karoliine Holter

University of Tartu

20 August 2026

# Why Care About Program Correctness?

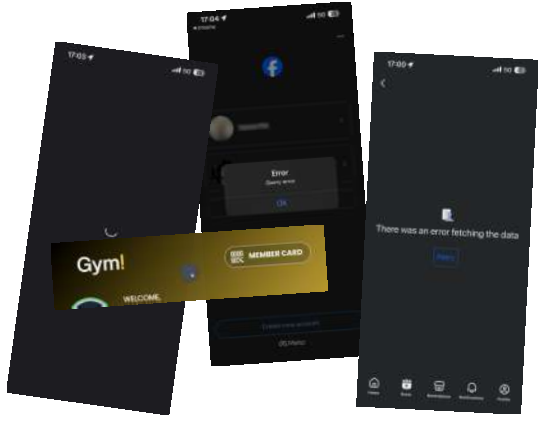
- Typical examples include rockets, aircraft, and nuclear power plants.
- Therac-25 radiation therapy machine: overdoses caused by a race condition.
- But also the smaller things in our everyday lives ...
- How much has „bad programming“ hurt you?

# Why Care About Program Correctness?

- Typical examples include rockets, aircraft, and nuclear power plants.
- Therac-25 radiation therapy machine: overdoses caused by a race condition.
- But also the smaller things in our everyday lives ...
- How much has „bad programming“ hurt you?

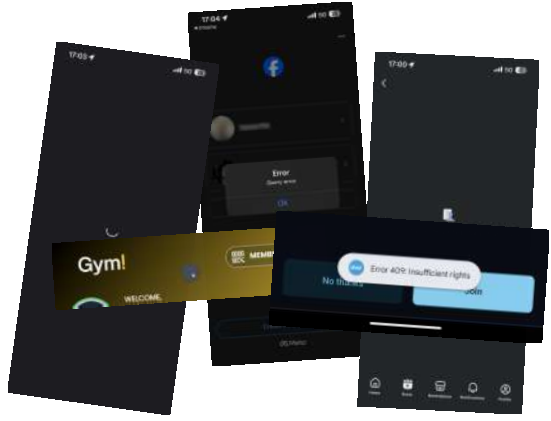
# Why Care About Program Correctness?

- Typical examples include rockets, aircraft, and nuclear power plants.
- Therac-25 radiation therapy machine: overdoses caused by a race condition.
- But also the smaller things in our everyday lives ...
- How much has „bad programming“ hurt you?



# Why Care About Program Correctness?

- Typical examples include rockets, aircraft, and nuclear power plants.
- Therac-25 radiation therapy machine: overdoses caused by a race condition.
- But also the smaller things in our everyday lives ...
- How much has „bad programming“ hurt you?

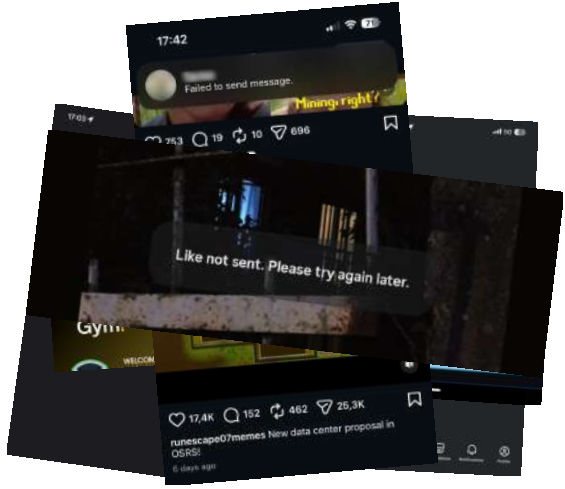


# Why Care About Program Correctness?

- Typical examples include rockets, aircraft, and nuclear power plants.
- Therac-25 radiation therapy machine: overdoses caused by a race condition.
- But also the smaller things in our everyday lives ...
- How much has „bad programming“ hurt you?

# Why Care About Program Correctness?

- Typical examples include rockets, aircraft, and nuclear power plants.
- Therac-25 radiation therapy machine: overdoses caused by a race condition.
- But also the smaller things in our everyday lives ...
- How much has „bad programming“ hurt you?



# Goal and Angle

CHESS

## Goal

- spark interest in programming languages and verification
- build intuition for how verification methods work
- make the mechanics of verification less mysterious

## Angle

✗ not a heavy theory lecture

✗ not a tool-usage tutorial

✓ an example-driven overview of verification mechanics and tradeoffs

# PL Researchers Want to Make Programs Better in Different Ways

1. Language specification

**Language specification**

- syntax: what forms are allowed?

```
if 5 > 2:
  print("Five is greater than two")
```

- semantics: what does a program *actually mean*?

```
x = x + 1
```

*„A program is not just a string of symbols to copy from ~~Stack Overflow~~ a language model.“*

# PL Researchers Want to Make Programs Better in Different Ways

1. Language specification
2. Formal semantics

**Formal semantics**

We give mathematically precise definitions of program meaning

- operational semantics: execution as transition/evaluation rules
- denotational semantics: meaning as mathematical objects

useful for proving equivalence, optimizations, and effects

# PL Researchers Want to Make Programs Better in Different Ways

1. Language specification
2. Formal semantics
3. Static reasoning

**Static reasoning**

We reason about a program *without running it*

- types
  ```
  x = ''abc'' + 1
  ```
- static analysis
  ```
  x = [1,2,3]
  x[4]
  ```

# PL Researchers Want to Make Programs Better in Different Ways

CHESS

1. Language specification
2. Formal semantics
3. Static reasoning
4. Dynamic reasoning

**Dynamic reasoning**

We study program behavior *during execution*

- testing
  **assert** sum(2,3) == 5
- error messages

```
>>> x = [1,2,3]
>>> x[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- monitors, profiles

# PL Researchers Want to Make Programs Better in Different Ways

1. Language specification
2. Formal semantics
3. Static reasoning
4. Dynamic reasoning
5. Implementation

**Implementation**

How do we make a program run on a machine?

- interpretation
- compilation

# Program Verification Methods

# Model Checking

$$\{\{x \mapsto 4\}, \{x \mapsto 5\}, \{x \mapsto 6\}, \dots\} \xrightarrow{x=x+1} \{\{x \mapsto 5\}, \{x \mapsto 6\}, \{x \mapsto 7\}, \dots\}$$

$$\{x \mapsto [4, \infty]\} \xrightarrow{x=x+1} \{x \mapsto [5, \infty]\}$$

Cousot & Cousot (POPL 1977)

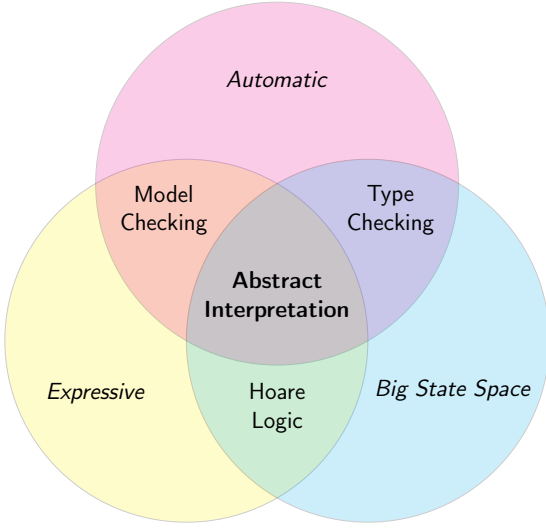
# Hoare Logic

$$\frac{x \geq 4 \rightarrow x + 1 \geq 5 \qquad \{x + 1 \geq 5\}x = x + 1\{x \geq 5\} \qquad x \geq 5 \rightarrow x \geq 5}{\{x \geq 4\}x = x + 1\{x \geq 5\}}$$

# Weakest Precondition

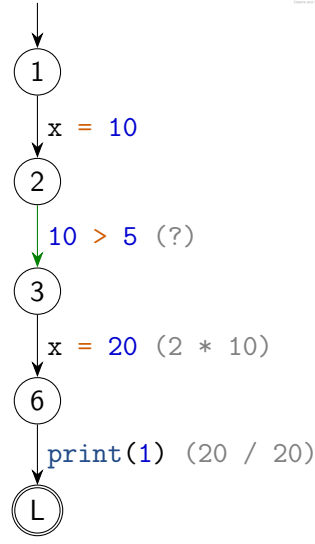
$$x + 1 \geq 5 \xrightarrow{x=x+1} x \geq 5$$

# Comparison of Methods



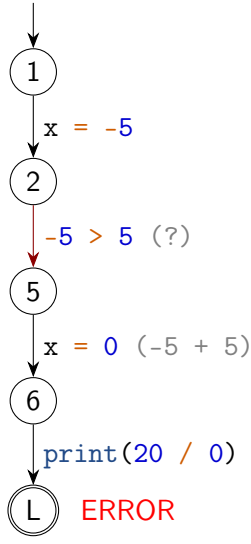
# Interpretation: what happens with input 10?

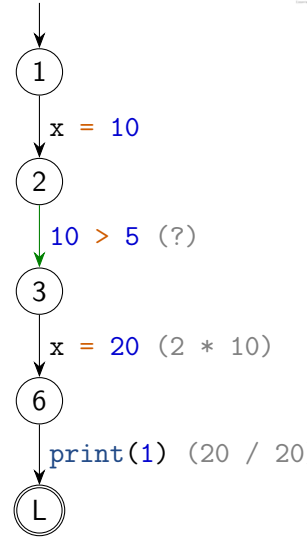
```
1 def fun(x):
2     if x > 5:
3         x = 2 * x
4     else:
5         x = x + 5
6     print(20 / x)
```



# Interpretation: what happens with input -5?

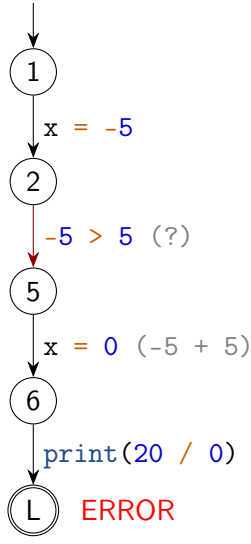
```
1 def fun(x):
2     if x > 5:
3         x = 2 * x
4     else:
5         x = x + 5
6     print(20 / x)
```



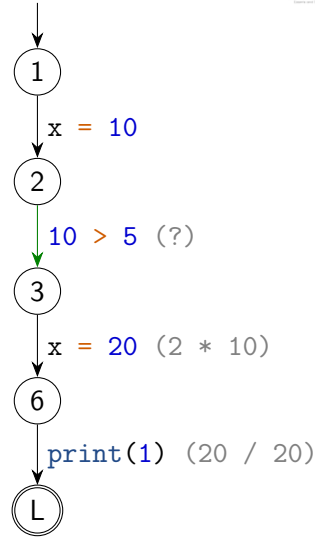



# Interpretation: what happens with other inputs?

```
1 def fun(x):
2     if x > 5:
3         x = 2 * x
4     else:
5         x = x + 5
6     print(20 / x)
```

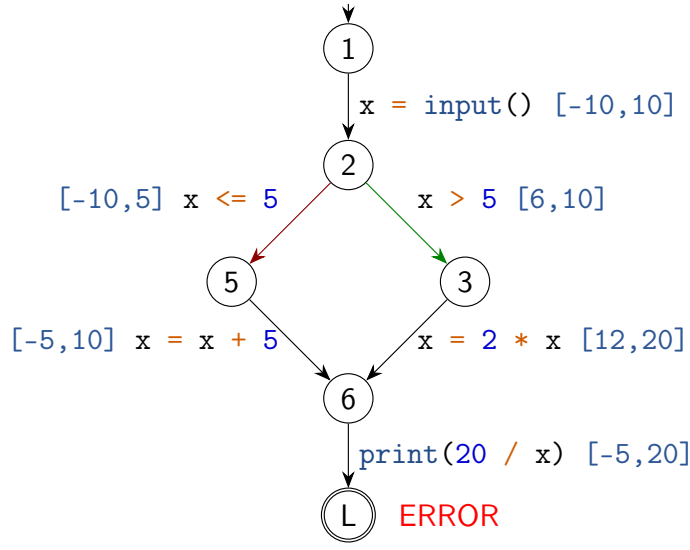



...



# Abstract Interpretation, Input [-10, 10]

```
1 def fun(x):
2     if x > 5:
3         x = 2 * x
4     else:
5         x = x + 5
6     print(20 / x)
```



Model checking: concrete-state exploration

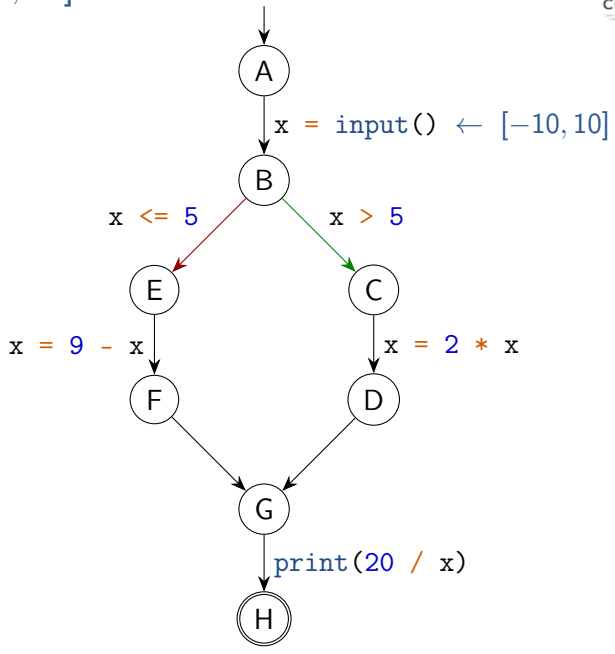
$$\{\{x \mapsto 4\}, \{x \mapsto 5\}, \{x \mapsto 6\}, \dots\} \xrightarrow{x=x+1} \{\{x \mapsto 5\}, \{x \mapsto 6\}, \{x \mapsto 7\}, \dots\}$$

Abstract interpretation: abstract domain computation

$$\{x \mapsto [4, \infty]\} \xrightarrow{x=x+1} \{x \mapsto [5, \infty]\}$$

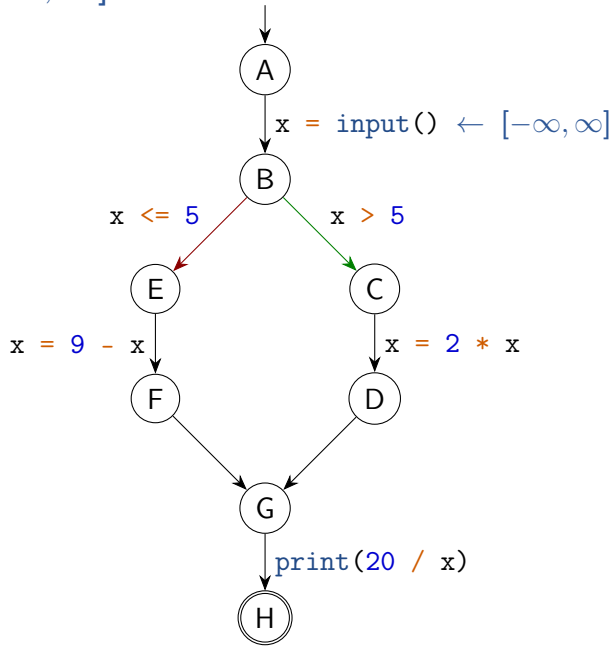
# Second Program with Input $[-10, 10]$

```
1 x = input()
2 if x > 5:
3     x = 2 * x
4 else:
5     x = 9 - x
6 print(20 / x)
```



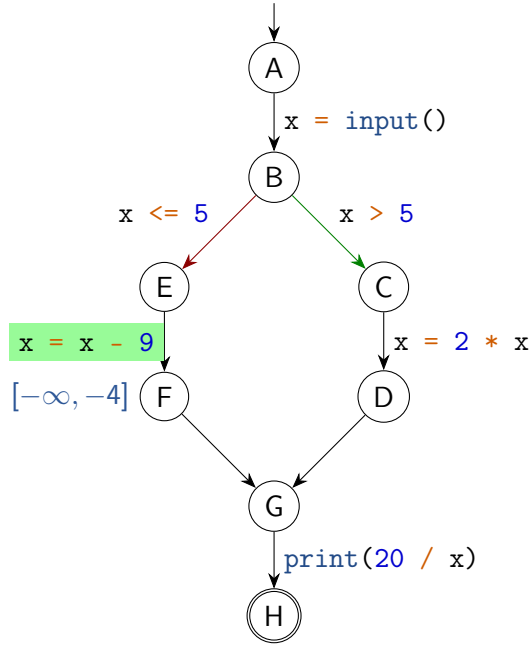
# Second Program with Input $[-\infty, \infty]$

```
1 x = input()
2 if x > 5:
3     x = 2 * x
4 else:
5     x = 9 - x
6 print(20 / x)
```



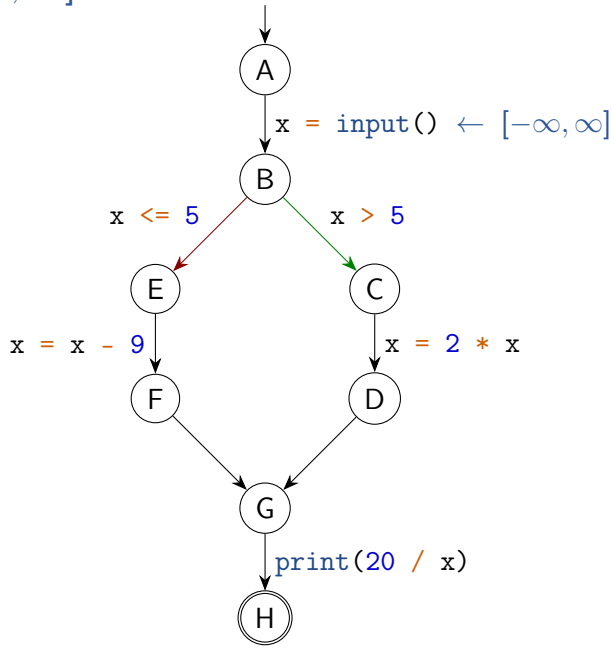
# Third Program

```
1 x = input()
2 if x > 5:
3     x = 2 * x
4 else:
5     x = x - 9
6 print(20 / x)
```



# Third Program with Input $[-\infty, \infty]$

```
1 x = input()
2 if x > 5:
3     x = 2 * x
4 else:
5     x = x - 9
6 print(20 / x)
```



# Abstract Interpretation

CHESS

## Sound (no false negatives)

*If a real execution can fail, the abstract execution reports a failure.*

equivalently

*If no failure appears abstractly, then no real failure can occur.*

This proves program correctness.

## Imprecise

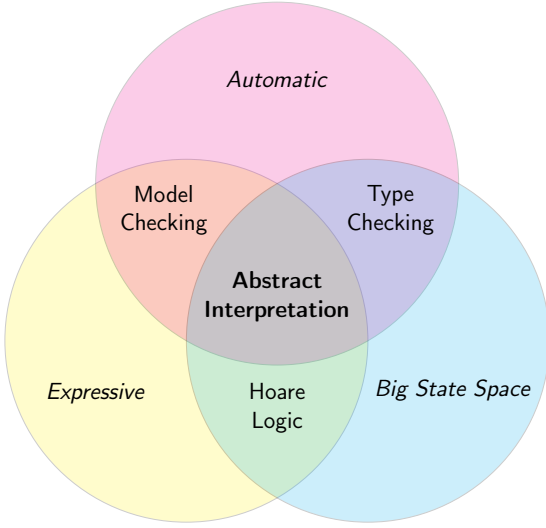
*If a failure appears abstractly, a real failure may still be impossible.*

It can be wrong, but only in one direction: it may report false positives.

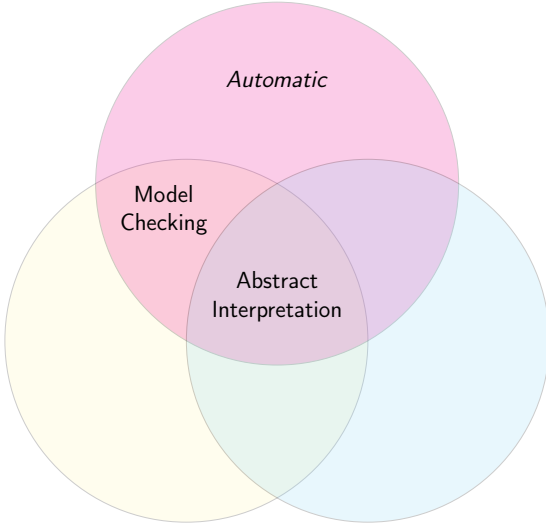
## Tradeoff

Abstract interpretation is typically **sound but incomplete**: it scales well and can prove the absence of errors automatically, but it may also report spurious alarms.

# Comparison of Methods



# Comparison of Methods



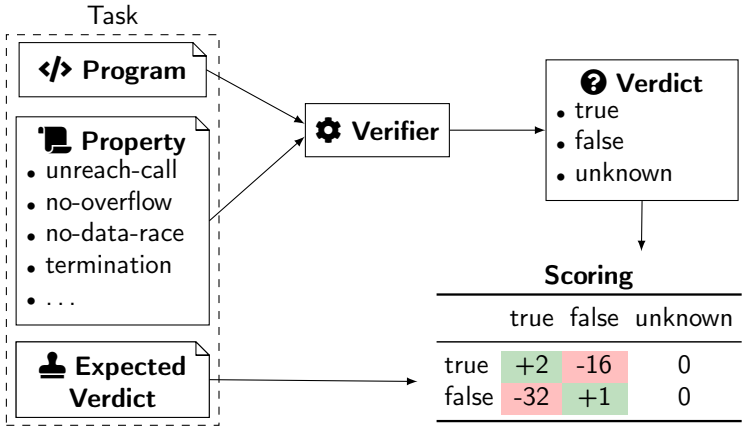
# *Competition on Software Verification (SV-COMP)*

CHESS

An annual international competition in which software verification tools are compared and evaluated: the de facto reference point for comparison of state-of-the-art automated software verifiers.

- A common benchmark suite for evaluating verification tools
- Each benchmark comes with a fixed specification/property to check
- Participants submit tools that check program correctness using different methods
- Tools are run under comparable conditions
- Results are summarized with standardized scoring, making tradeoffs visible

# SV-COMP Task and Scoring Model (Simplified)



# Goblint

- An analyzer for concurrent C programs
- Based on abstract interpretation
- Special focus on data races
  - Therac-25 kind of incidents are preventable
- A collaboration between University of Tartu and Technical University of Munich
- Multiple-time winner of the *NoDataRace* category in SV-COMP
- *Wettlaufweltmeister*

# From Verification Competition to Reality

*Competition on Software Verification (SV-COMP)*, Dirk Beyer: SV-COMP = F1



Matti Blume: Mercedes-Benz, Techno-Classica 2018, Essen

# Real-World Uses of Formal Verification

- **Astrée + Airbus**: abstract interpretation was used to prove the absence of run-time errors in Airbus A340 fly-by-wire control software, and later extended to the A380 flight-control code.
- **NASA IKOS**: NASA released IKOS, an abstract-interpretation-based static-analysis framework for verifying critical software properties.
- **NASA + model checking**: NASA Langley's DO-333 case studies for a flight-guidance system explicitly include model checking alongside theorem proving and abstract interpretation.

# Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law

Raphaël Monat[1⋆(✉)], Aymeric Fromherz[2⋆], and Denis Merigoux[2]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Inria Paris, Paris, France
{raphael.monat,aymeric.fromherz,denis.merigoux}@inria.fr

**Abstract.** Legal expert systems routinely rely on date computations to determine the eligibility of a citizen to social benefits or whether an application has been filed on time. Unfortunately, date arithmetic exhibits many corner cases, which are handled differently from one library to the other, making faithfully transcribing the law into code error-prone, and possibly leading to heavy financial and legal consequences for users.
In this work, we aim to provide a solid foundation for date arithmetic working on days, months and years. We first present a novel, formal semantics for date computations. and formally establish several semantic properties through a mechanization in the F⋆ proof assistant. Building upon this semantics, we then propose a static analysis by abstract interpretation to automatically detect ambiguities in date computations. We finally integrate our approach in the Catala language, a recent domain-specific language for formalizing computational law, and use it to analyze the Catala implementation of the French housing benefits, leading to the discovery of several date-related ambiguities.

# Takeaways

- Testing checks some executions; formal verification can reason about all executions.
- Verification methods trade off automation, precision, and scalability.
- These techniques matter in real systems, from avionics to even legislation.
- What about non-safety critical systems?

# Acknowledgements


- Slide 3 terminology and framing adapted from Michael Hicks, „What is Programming Languages Research?“
- Slides 4–8 adapted and translated from Simmo Saan's AKTSP course slides: `https://courses.cs.ut.ee/2026/AKT/spring`
- Slides 10–17 adapted and translated from Simmo Saan's AKTSP course slides: `https://courses.cs.ut.ee/2026/AKT/spring`
- Real-world examples slide based on Astrée and NASA source pages: `https://www.astree.ens.fr/`, `https://software.nasa.gov/software/ARC-16789-1`, `https://shemesh.larc.nasa.gov/fm/DO-333-case-studies.html`


# Thank you!