COURSE ON PROGRAM SEMANTICS & VERIFICATION 2025/2026

URL: www.cs.uu.nl/docs/vakken/pv

by Wishnu Prasetya (s.w.b.Prasetya@uu.nl)

PROGRAM CORRECTNESS

- We do not want to deliver buggy software, not to mention that errors might have severe consequences.
- Yet thorough testing is very time consuming (expensive)
- So, we are looking for techniques to do software verification **automatically**.
- Is this possible?

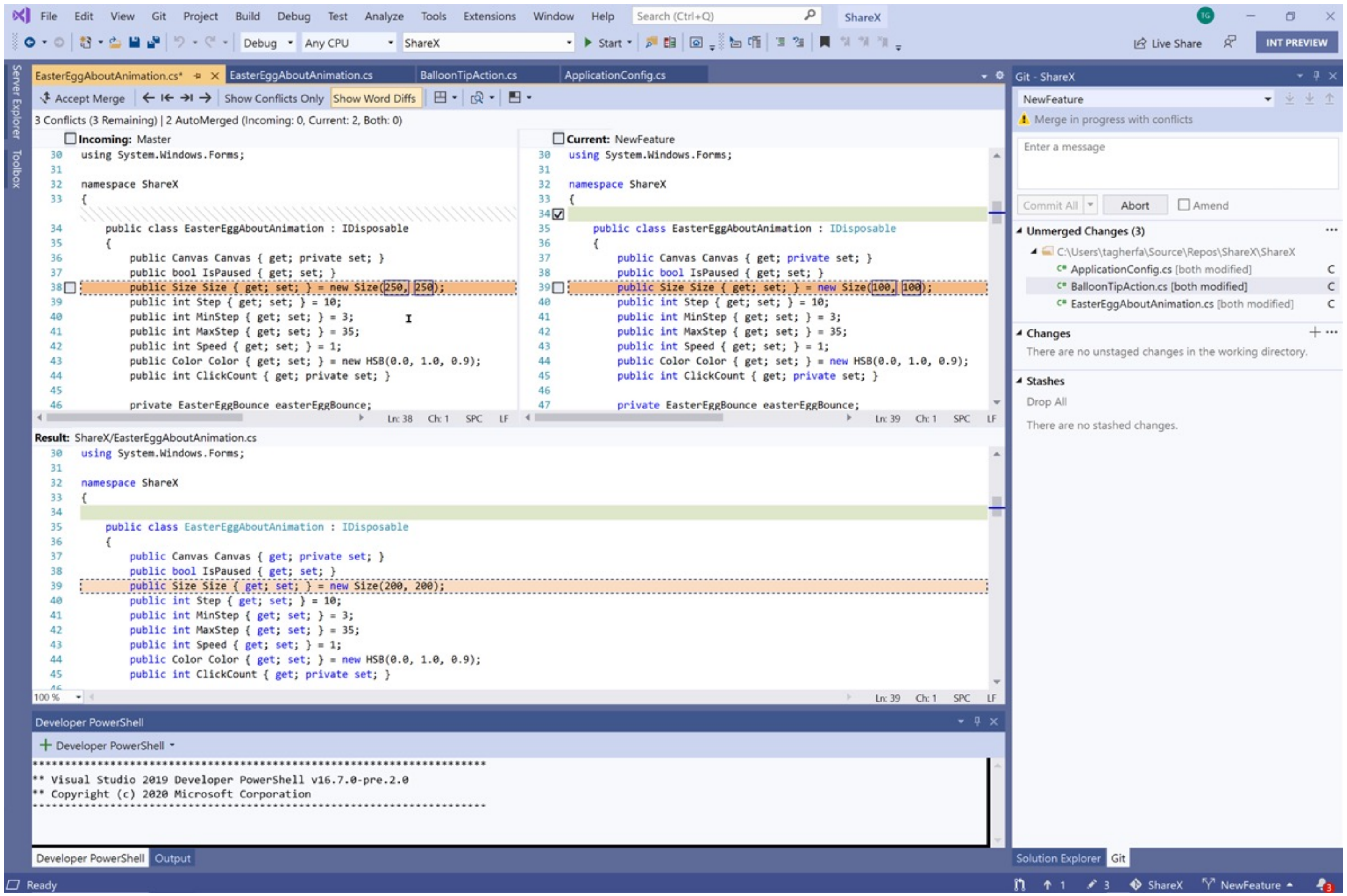
AUTOMATED PROGRAM VERIFICATION

- In general undecidable ☹
- We need to look for sub-cases that are: (1) decidable, and (2) have practical values.

  Example:
  - Type checking
  - Random testing with QuickCheck
  - More?

- To manage your expectation: it may take years or decades for theories to turn into a mature technology.

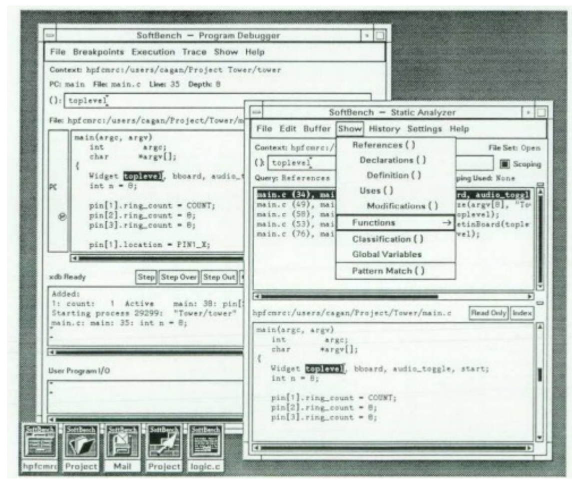
HOW LONG DOES IT TAKE?



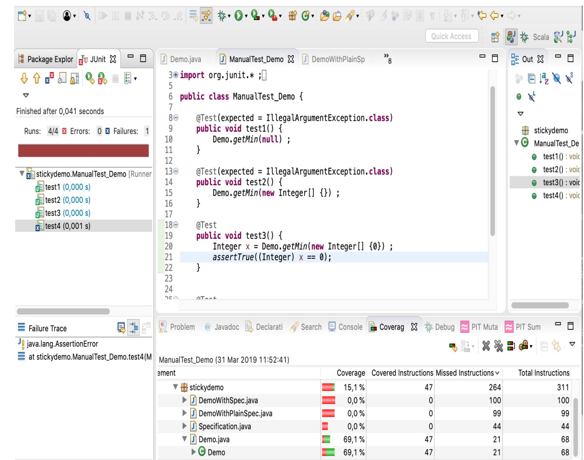
IDE 2022

HOW LONG DOES IT TAKE?

70’s



90’s



10’s

IDE

HOW LONG DOES IT TAKE?

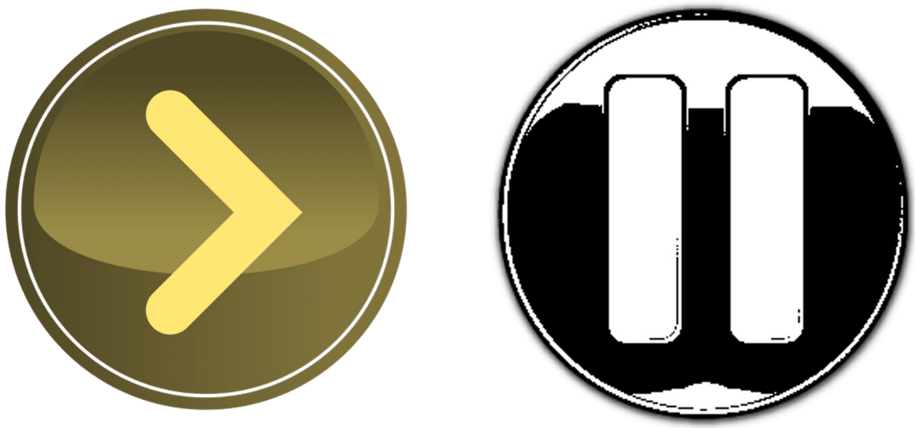
PROOF EC
[A1:] $1 \le k \le N$
[A2:] $ok = (\forall j : 0 < j < k : a[j-1] < a[j])$
[A3:] $k = N \lor \neg ok$ ✓
[G:] $ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
BEGIN
1. {see subproof CS1} $k = N \Rightarrow ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
SUBPROOF CS1
[A:] $k = N$
[G:] $ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
BEGIN
1. {Fill in k=N in EC.A2}
$ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
END
2. {see subproof CS2} $\neg ok \Rightarrow ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
SUBPROOF CS2
[A:] $\neg ok$
[G:] $ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
BEGIN
1. {def of ¬ok according to EC.A2}
$\neg(\forall j : 0 < j < k : a[j-1] < a[j])$
2. {Negate ∀} $(\exists j : 0 < j < k : a[j-1] \ge a[j])$
3. {∃-elimination on 2} [SOME j] $0 < j < k \land a[j-1] \ge a[j]$
4. {Conjunction on 3 and EC.A1}
$0 < j < k \land 1 \le k \le N \land a[j-1] \ge a[j]$
5. {Rewriting 4} $0 < j < N \land a[j-1] \ge a[j]$
6. {∃-introduction on 5} $(\exists j : 0 < j < N : a[j-1] \ge a[j])$
7. {Rewriting 6} $\neg(\forall j : 0 < j < N : a[j-1] < a[j])$
8. {follows from 1-7} $\neg ok = \neg(\forall j : 0 < j < N : a[j-1] < a[j])$
9. {Rewrite 8} $ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
END
3. {Case split on 1, 2 and A3}
$ok = (\forall j : 0 < j < N : a[j-1] < a[j])$
END

proving correctness in 80's

```
val HOARE_ifthenelse_thm = prove
  (``HOARE S1 (AND g p) q /\ HOARE S2 (AND (NOT g) p) q
     ==>
     HOARE (IFTHENELSE g S1 S2) p q``,
    REWRITE_TAC [HOARE_def]
    THEN REWRITE_TAC [exec_def, AND_def, NOT_def]
    THEN BETA_TAC
    THEN PROVE_TAC[]
  );
```

in 2000's

PROVING CORRECTNESS NOWADAYS



Though in terms of user-friendliness, performance, documentation, user support etc, the technologies are not maturing yet. Also: scalability remains a challenge.

(hopefully, you can contribute to this as well)

LEARNING GOALS

- Become familiar with, and acquire insight on the underlying concepts of:
  - **formalisms to express the correctness of programs**: a Hoare-style formalism, LTL, CTL.
  - **automated verification techniques**: predicate transformer, symbolic verification, model checking LTL/CTL, probabilistic model checking.
  - ~~**program semantics**: operational, denotational, axiomatic~~ (this is covered in MCPD).

LEARNING GOALS

- Acquire hands-on experience (towards your future research) with :
  - **implementing a verification technique** (one of previously mentioned).
  - **using a verification tool** to model a problem and conduct a verification of its solution.

SETUP

- **Lectures** : in the morning 9:00 – 11:00 every monday and wednesday.
- Discussing exercises, doing tutorial, doing/discussing/presenting project: 11:00 – 12:45, every monday and wednesday after lectures.

  We may use some of the exercise-hours for lectures if we get a bit short in time.

- **2 exams + 1 project** that also involves writing a paper presenting your results.
- Several other activities e.g. doing exercises together, running a tutorial.

EVALUATION

- Project + tutorial + assignment
  - All are **mandatory**
  - You can work in a team, up to size 3.
- Exams : 2x
- Grading:
  - **A** SPIN tutorial & assignments : 7.5%
  - **S** Probabilistic Model Checking tutorial : 7.5%
  - **E** Exams**:** 20% E1 + 25% E2, average should be ≧ 5.0
  - **P** Project: 40%
- Your grade = **P+E+A+S** rounded to the closest 0.1 pt, **however** if 5.0 ≤ P+E+A+S < 6.0 your grade is rounded to the closest integer.

EVALUATION

- Supplementary exam,
  - Note the Faculty’s regulation concerning this.

COVERAGE

	A	P	E1	E2
Pred. transformer		✓	✓	
LTL + model checking	✓		✓	
CTL + model checking				✓
Symbolic model checking		✓		✓
Probabilistic model checking	✓			✓
Experience with verification tool	✓	✓		
Can implement a verification technique		✓		

(may change if the actual progress during the course requires us to adapt)

COURSE MATERIALS

- Lecture Notes PV, free. Get it at the course website.
- Chapter 10 on probabilistic model checking of Principles of Model Checking, by Baier, Katoen, et. al. MIT Press, 2008.
- Slides.

SOFTWARE

- You need your own laptop/machine.
- Needed software:
  - Haskell (a functional programming language)
  - Z theorem prover and its Haskell-binding, Install them ASAP!!
  - Spin. Install them ASAP!! Model checker SPIN, also requires
    - C compiler + its standard libraries.
    - On Windows you probably also need Cygwin or Msys+Mingw to get the C compiler.
    - Tk/Tcl for its GUI
    - Dot for drawing state automata
  - PRISM probabilistic symbolic model checker, https://www.prismmodelchecker.org/
- Links to Spin can be found in PV website. Consult their install instructions.

OTHER NOTES

- www.cs.uu.nl/docs/vakken/pv

Slides, course plan, etc.

- Most communications through MS-Team