Software Validation and Verification
Section II: Model Checking
Topic 1. Introduction

Pedro Cabalar

Department of Computer Science and IT
University of Corunna, SPAIN
cabalar@udc.es

2020/2021

Software as a product

- Software is a complex, conceptual product ⇒ errors are inherent
  - ☺ Good news: computers always do what we tell them to do!
  - ☹ Bad news: what we tell is not always what we want

When the outcome was not expected . . .

Why program outcome $\neq$ expected outcome?

- Example: "*I want to print my salary for the first six months*"

```
for (i=0;i<=5;i++)
  printf("%d\n",salary[j]);
```

this program is wrong because variable *j* should be *i*

- Someone else tells me to "*print the total salary for the first six months*"

```
for (i=0;i<=5;i++)
  printf("%d\n",salary[i]);
```

error with *j* fixed, but it is the wrong program!

```
total=0;
for (i=0; i<=5; i++)
  total+=salary[i];
printf("%d\n",total);
```

When the outcome was not expected . . .

So a program behaviour can be faulty or unexpected

- Faulty: is this program right? $\Rightarrow$ verification
- Unexpected: is this the right program? $\Rightarrow$ validation

In verification we assume we understood what the program must do but want to check that it is done correctly.

An example

- We want to compute the greatest common divisor of two integers $x > y > 0$, $gcd(x, y)$
- The following code is obviously correct:

```
gcd=1;
for (i=2; i<=y; i++)
  if (x % i == 0  &&  y % i == 0)
    gcd=i;
```

but rather inefficient. How many steps do we need for `gcd(10000000,1000000)` ?

An example

Euclid, by José de Ribera

```
a=x;
b=y;
while (a!=b)
  if (a>b)
    a=a-b;
  else
    b=b-a;
gcd=a;
```

Euclid's algorithm [$\sim$ 300 BC]

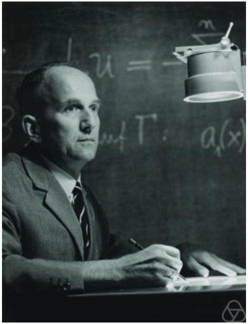
- Obviously faster but . . .

Exercise 1 (0,5 points T.G.R.)

Can you prove it is correct? (if not, a real shock after 2.300 years!)

Another example: Collatz conjecture

Things can be hard even for simple loops ...



Lothar Collatz
6/7/1910 - 26/9/1990

```
// int x, x>=0
while (x!=1)
  if (x%2==0)
    x=x/2;
  else
    x=3*x+1;
```

- Does this loop stop (i.e. reach $x = 1$) for any starting value $x \geq 0$? This is an unsolved problem!

Outline

Formal Verification

- Goal: prove or disprove the correctness of a given system like software (algorithms, protocols) or hardware (circuits).
- Formal methods
    1. Formal specification:
       formulas asserting what the system should do, not how.
    2. Formal verification: prove that the system satisfies the specification.
- formal = use mathematical objects to model the system.
  Examples: finite state machines, Petri nets, program semantics, process algebras, logics (classical, modal, temporal), etc.

Formal Specification

Goal: write a formula that describes the problem solution

Keypoint: how strong is the formula?

- Always correct: prove a necessary and sufficient condition
  The program is correct if and only if:

  $$x \bmod a = 0 \wedge y \bmod a = 0 \wedge \neg \exists z > a\,(x \bmod z = 0 \wedge y \bmod z = 0)$$

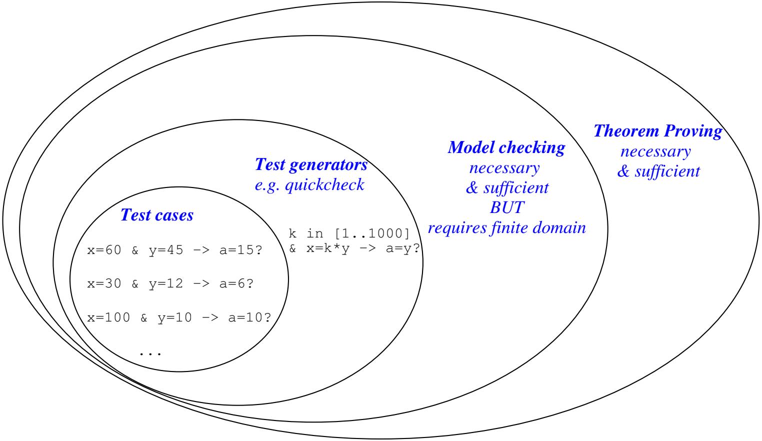
- Sometimes correct: prove a necessary condition.
  If the program is correct, it must satisfy:

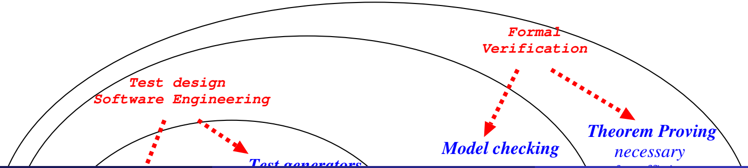
  $$x = k * y \quad \Rightarrow \quad a = y$$

- Correct for some test case (a stronger necessary condition).

  $$x = 60 \wedge y = 45 \quad \Rightarrow \quad a = 15$$

Verification





Formal Verification

	Trial-and-error testing	Formal verification
Confidence	100 % never reached	Mathematical methods
Keypoint	good design of test cases	good specification
Input object	We depend on a program	We work with an algorithm
Efficiency measurement	execution time, memory consumed	complexity

Warning: tests are still necessary for validation. We can prove that a property holds, but perhaps it's not the right property to be proved!

Formal Verification

- The ideal situation of Formal Verification
  Program + formulas $\longrightarrow$ Correct?
  - Yes : correctness proof
  - No : counterexample
  - ??: sometimes we may have no answer!

Alan Turing
(1912 – 1952)

Halting problem [Turing 1936] is undecidable:
no algorithm can decide in finite time
whether any arbitrary pair (program, input)
will eventually halt or run forever.

Approach 1. Theorem proving

- Theorem proving uses logical inference.
- Semi-automated: it usually requires user's supervision or selection of proof strategies.
- May deal with infinite domains . . . but the method is undecidable
- Best suited for proving correctness during the algorithm design.
- Examples of theorem provers: Isabelle, ACL2, Coq, PVS.

  In Coq, the proof is constructive: we can automatically derive a correct program in a functional language.

Approach 2. Model Checking

- Model checking: systematically exhaustive exploration of the states and transitions in the model.
- Decidable . . . but requires finite domain
(or infinite domain with a finite representation)
- Fully automated: no user supervision required
- Best suited for finding counterexamples on an already built system.

- Typically applied to reactive systems: they have inifinite execution, but must satisfy some properties expressed in temporal logics.
- Those properties are checked using tools that (intelligently) explore the state space. These tools are called model checkers.

Outline

A bit of history . . .

- But up to late 60's: GOTO statements, "spaghetti code"

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- During the 1960's algorithm design was born.
- Structured programming [Böhm & Jacopini 66]:
  any program = {sequential + conditional + iterative} instructions.

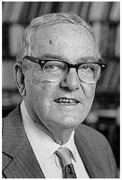
```
  for (i=1; i<=10; i++)
    printf("%d squared= %d\n",i,i*i);
  printf("Program completed.\n");
```

Formal reasoning in AI

John McCarthy
(1927 – 2011)
Turing Award 1971

- [McCarthy 1951] *"A basis for a Mathematical Theory of Computation"*
  First proposal of replacing trial-and-error by formal proof of correctness.
- [McCarthy 1960] *"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"* = LISP language.
  First example of program semantics (operational semantics) using lambda calculus for LISP.

Allen Newell
(1927 – 1992)
Turing Award 1975



Herbert Simon
(1916 – 2001)
Turing Award 1975
Nobel (Economics) 1978

- [Herbert & Simon 1955] Logic Theorist:
  First successful theorem prover
  (proved 38 theorems from Russell's Principia Mathematica)

Algorithm design and correctness

Sir C. Anthony R. Hoare (1934 –)
Turing Award 1980

- [H. 1962] designs the Quicksort algorithm. Was it correct? crucial point: defining a program semantics
- [H. 1969] Hoare Logic (axiomatic semantics).

$\{Q\}$ **prog** $\{R\}$ = If precondition Q initially true, then program **prog** terminates satisfying postcondition R.

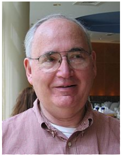
"There are two ways of constructing a piece of software: One is to make it so simple that there are obviously no errors, and the other is to make it so complicated that there are no obvious errors." Tony Hoare.

Algorithm design and correctness

Edsger W. Dijkstra (1930 – 2002)
Turing Award 1972

- [D. 1959] algorithm for shortest path tree,
- [D. 1965] introduces the idea of semaphore for controlling shared resources in a concurrent environment.
- [D. 1968] *Go To Statement Considered Harmful*.
- [D. 1976] *A Discipline of Programming*:
  formal verification, weakest precondition, program derivation, guarded commands programming language.

Denotational semantics



Dana S. Scott (1932 –)
Turing Award 1976

- [Scott & Rabin 1959] *“Finite Automata and Their Decision Problems”* (nondeterministic machines, automata theory)
- [Scott & Strachey 1971] *“Toward a mathematical semantics for computer languages”* denotational semantics.

  “Denotation” = function from input to output.
- A semantics is compositional when the meaning of a sentence is built on the meaning of its sub-sentences ⇒ basis of functional languages with concurrency (e.g. Haskell).

Model Checking main approaches

Amir Pnueli
(1941 – 2009)
Turing Award 1996

Edmund M. Clarke
(1945 –)
Turing Award 2007

Allen Emerson
(1954 –)
Turing Award 2007

Model checking: two main approaches

- Linear Temporal Logic (LTL) proposed by Pnueli in the 70's. It is used by the SPIN model checker. More oriented to (concurrent) software verification.
- Computation Tree Logic (CTL) proposed by Clarke and Emerson. It constitutes the basis of SMV, NuMV, nuXmv model checkers. More oriented to circuit verification.

Model Checking main approaches

See Clarke's invited talk on model checking
at Vienna Summer of Logic 2014

https://vimeo.com/103456257

Model Checking main approaches

See Vardi's invited talk on LTL synthesis at ECAI 2020

https://digital.ecai2020.eu/conference/the-siren-song-of-temporal-synthesis/

Master

- Máster en Métodos Formales en Ingeniería Informática
- Tres universidades: Complutense, Autónoma y Politécnica de Madrid
- 60 ECTS = 1 año
- https://informatica.ucm.es/estudios/master-mfingenieriainf
- Vídeo de presentación https://www.youtube.com/watch?v=yAm_VFEgk1I