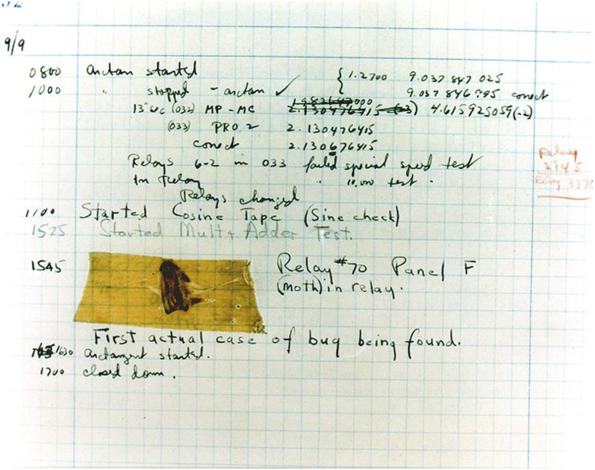
Introduction

Pedro Cabalar

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

2015/2016



The first real bug detected in Harvard Mark II

Introduction

- Software is a complex, conceptual product $\Rightarrow$ Errors are inherent
- Software may be faulty or work unexpectedly.

Facing a program, a pair of natural questions are:

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

Let us focus on verification ...

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 to y do
  if ((x mod i)=0) and ((y mod i)=0) then
    gcd:=i;
```

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

An example

- Euclid's algorithm [Euclid, 300 BC]:

```
a:=x;
b:=y;
while a<>b do begin
  if a>b then
    a:=a-b
  else
    b:=b-a;
end;
gcd:=a;
```

- Obviously faster but ...

Exercise 1 (0,6 points T.G.R.)

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

A bit of history . . .

- During the 1960's algorithm design was born.
- But up to late 60's: unreadable programs, `GOTO` statements.
  So-called "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
```

- Structured programming [Böhm & Jacopini 66]:
  any program = {sequential + conditional + iterative} instructions.

```
  for i:=1 to 10 do
    writeln(i,' squared= ',i*i);
  writeln('Program completed');
```

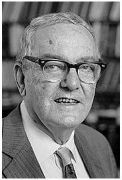
Formal reasoning in AI

John McCarthy
(1927 – 2011)
Turing Award 1971

- [McCarthy 1951] *"A basis for a Mathematical Theory of Computation"* actually the 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.
- McCarthy introduced the first example of program semantics (operational semantics) using lambda calculus for LISP.

Formal reasoning in AI

Allen Newell
(1927 – 1992)
Turing Award 1975



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

- [Herbert & Simon 1955] Logic Theorist: probably the 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.

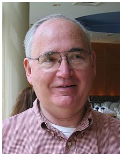
"There are two ways of constructing a piece of software: One is to make it so simple that there are obviously no errors, and the other is to make it so complicated that there are no obvious errors." Tony Hoare.

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).

Verification

- Verification: prove or disprove the correctness of a given system: software (algorithms, protocols) or hardware (circuits).
- Checking that the program is correct . . . but when? always, sometimes, in a given case?
  - Always correct: prove a necessary and sufficient condition
    The program is correct if and only if:

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

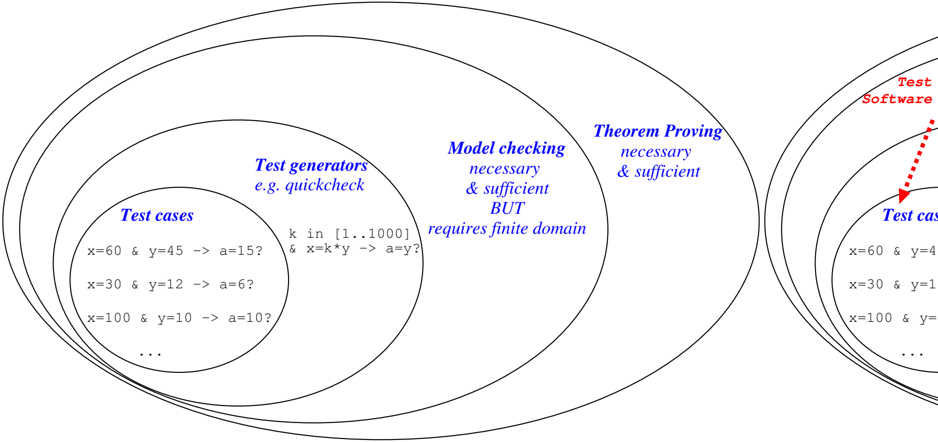
  - Sometimes correct: prove a necessary condition.
    If the program is correct, it must satisfy:

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

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

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

Verification



Formal Verification

- Formal methods
  1. Formal specification: formulas asserting what the system should do, not how.
  2. Formal verification: obtain a formal proof for the specification.

- The word formal means we use mathematical objects to model the system, both for specifying properties and for obtaining proofs.

- Some examples of used mathematical models: finite state machines, Petri nets, program semantics, process algebras, logics (classical, modal, temporal), etc.

Formal Verification

Trial-and-error verification	Formal verification
100 % confidence never reached	Mathematical methods
Keypoint: good design of test cases	Keypoint: good formulation of properties to prove
We always depend on a program	We can work with an algorithm
Efficiency = execution time	Efficiency = 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!

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

Alan Turing
(1912 – 1952)
No Turing Award
but he's Turing!

Halting problem [Turing 1936] is undecidable:
there exists no algorithm to decide
if any arbitrary pair program+input
will eventually halt or run forever.

Formal verification approaches

There are two main approaches to formal verification

1. Theorem proving: uses logical inference to prove the verification conditions.
   - Semi-automated: it usually requires user supervision or selection of proof strategies.
   - 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.

Formal verification approaches

There are two main approaches to formal verification

2. Model checking: systematically exhaustive exploration of the states and transitions in the model.
   - When models are infinite, only possible if we can deal with a finite representation. With finite models, verification becomes decidable.
   - Best suited for finding counterexamples on an already built system.
   - Typically applied to reactive systems: they have inifinite execution, but must satisfy some properties:
     - liveness: something good eventually happens;
     - safety: nothing bad ever happens.
   - Properties are expressed using temporal logics.
   - Those properties are checked using tools that (intelligently) explore the state space. These tools are called model checkers.

Model Checking main approaches

Amir Pnueli
(1941 – 2009)
Turing Award 1996

Edmund M. Clarke
(1945 –)
Turing Award 2007

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

Edmund M. Clarke
(1945 –)
Turing Award 2007

Exercise: watch Clarke's invited talk on model checking at Vienna Summer of Logic 2014
`https://vimeo.com/103456257`