COMPUTER AIDED VERIFICATION

LECTURE 1:

Overview of formal verification

Sławomir Lasota
University of Warsaw

PLAN

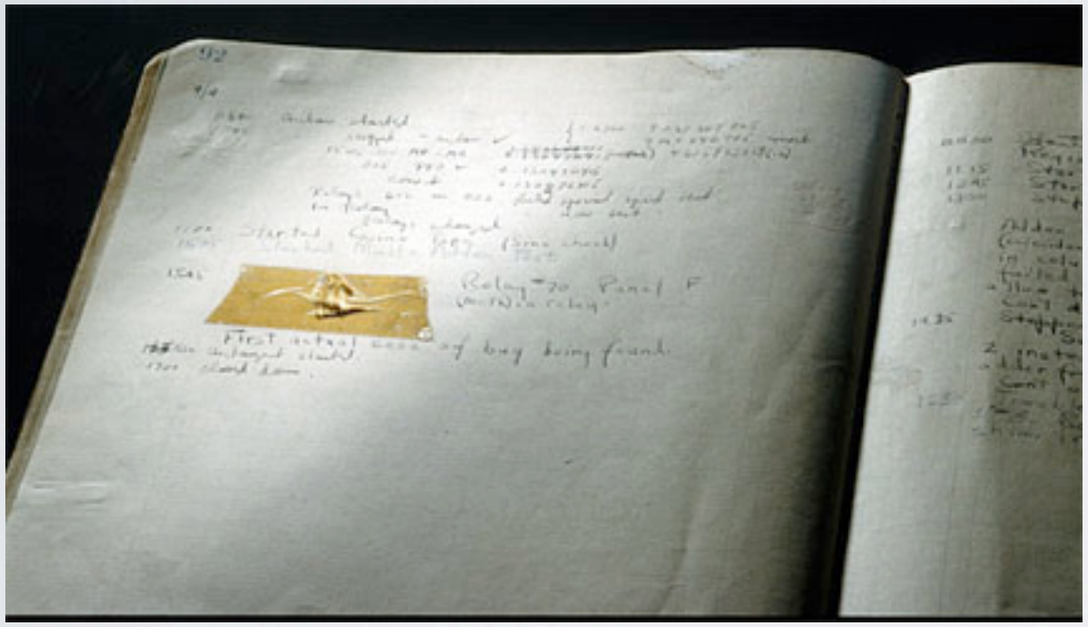
- Motivation (famous bugs)
- Motivation (success stories)
- Formal verification:
  - interactive (proving correctness)
  - approximation (static analysis)
  - abstraction (model checking)
- Brief history of formal verification

Famous bugs

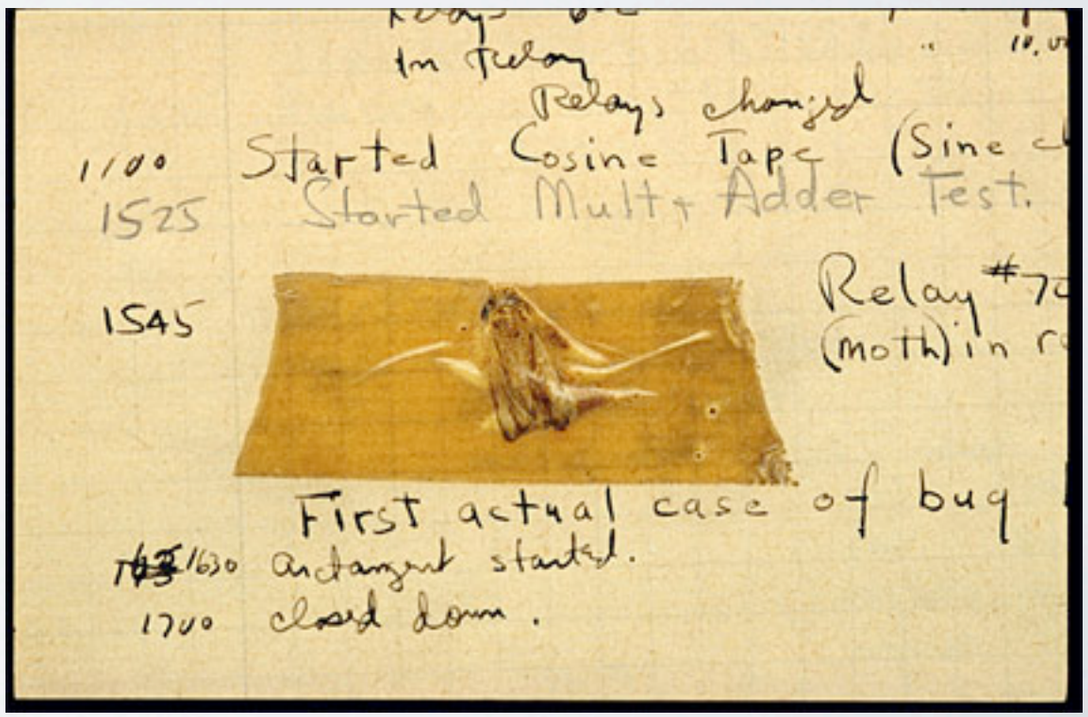
THE FIRST BUG...

THE FIRST BUG...

...was a moth:)



Mark II computer logbook



1947 Harward

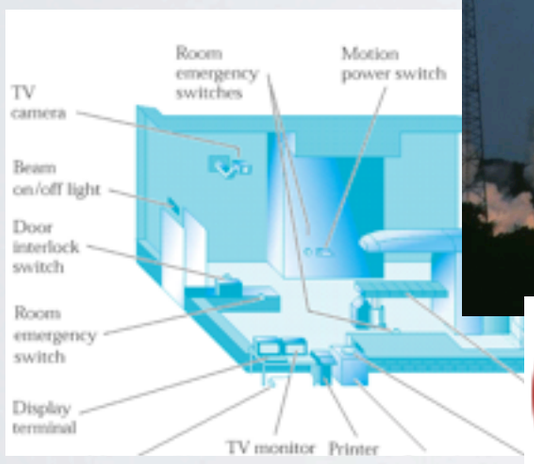
MARINER I

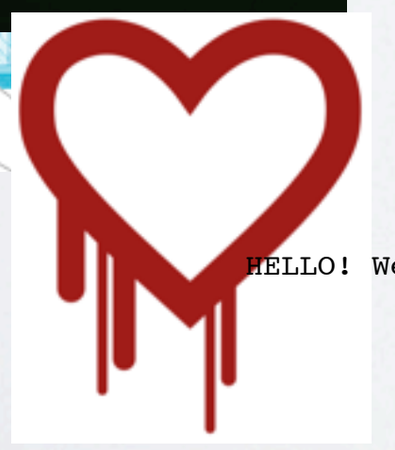
- period instead of comma in Fortran source code
- estimated cost: 18.5 mln $

(hypothesis)

```
    ...
    IF (TVAL .LT. 0.2E-2) GOTO 40
    DO 40 M = 1, 3
    W0 = (M-1)*0.5
    X = H*1.74533E-2*W0
    DO 20 N0 = 1, 8
    EPS = 5.0*10.0**(N0-7)
    CALL BESJ(X, 0, B0, EPS, IER)
    IF (IER .EQ. 0) GOTO 10
20  CONTINUE
    DO 5 K = 1. 3
    T(K) = W0
    Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
    D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4*
   *(B0**2-X*B0*B1))/Z
    E(K) = H**2*93.2943*W0/SIN(W0)*Z
    H = D(K)-E(K)
 5  CONTINUE
10  CONTINUE
    Y = H/W0-1
40  CONTINUE
    ...
```

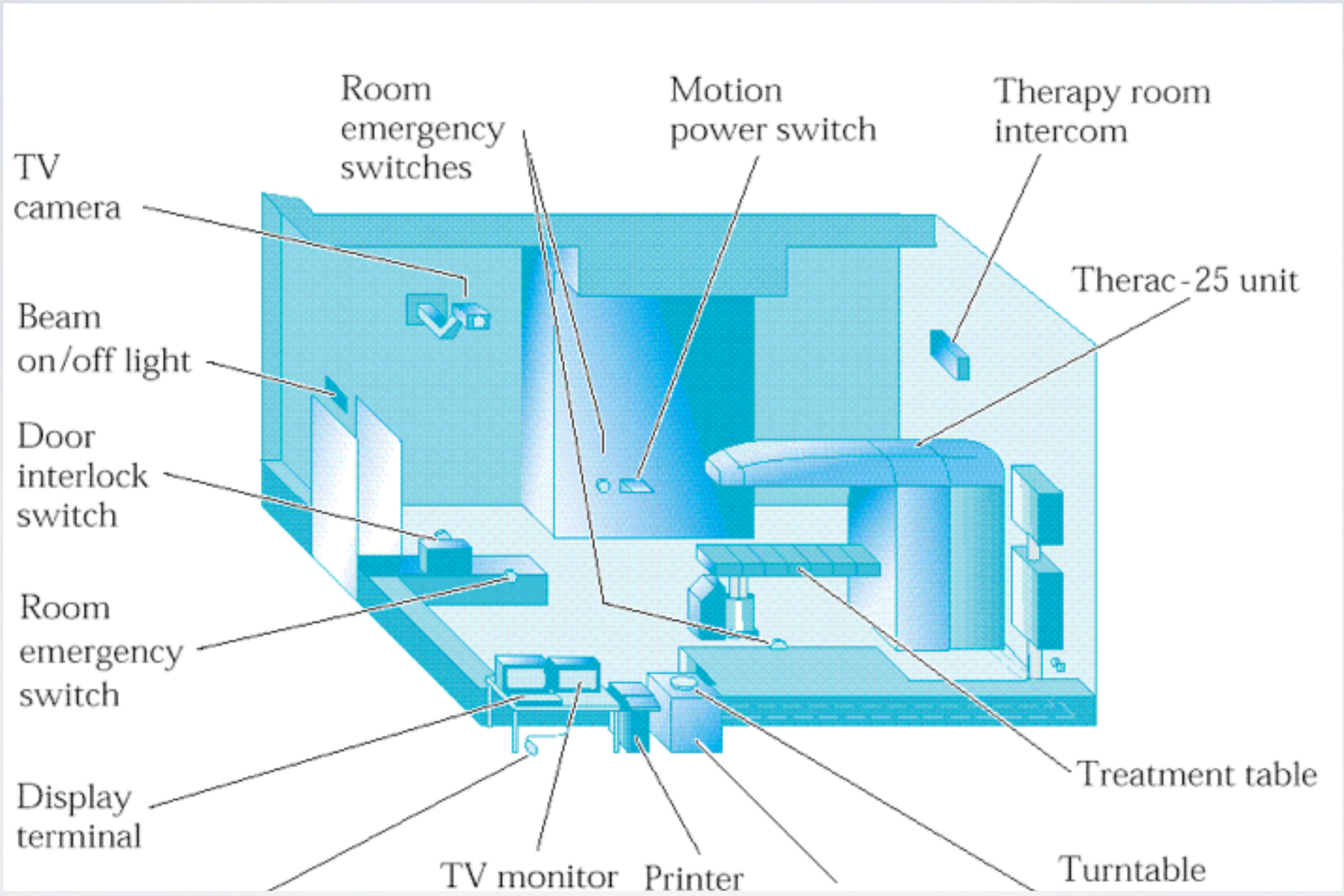
July 1962





HELLO! Welcome to http://www.worm.com! Hacked By Chinese!

THERAC-25



1985-87

- race condition
- at least 6 victims

PATRIOT MISSILE

- inaccurate calculation of time due to arithmetic rounding (drift by one third of a second over a period of one hundred hours)

- failed to track and intercept an incoming enemy's Scud missile

- 28 soldiers killed, around 100 injured

February 1991

PENTIUM FDIV BUG

- floating point division operation occasionally yields incorrect result

October 1994

ARIANE 5

FLIGHT 501

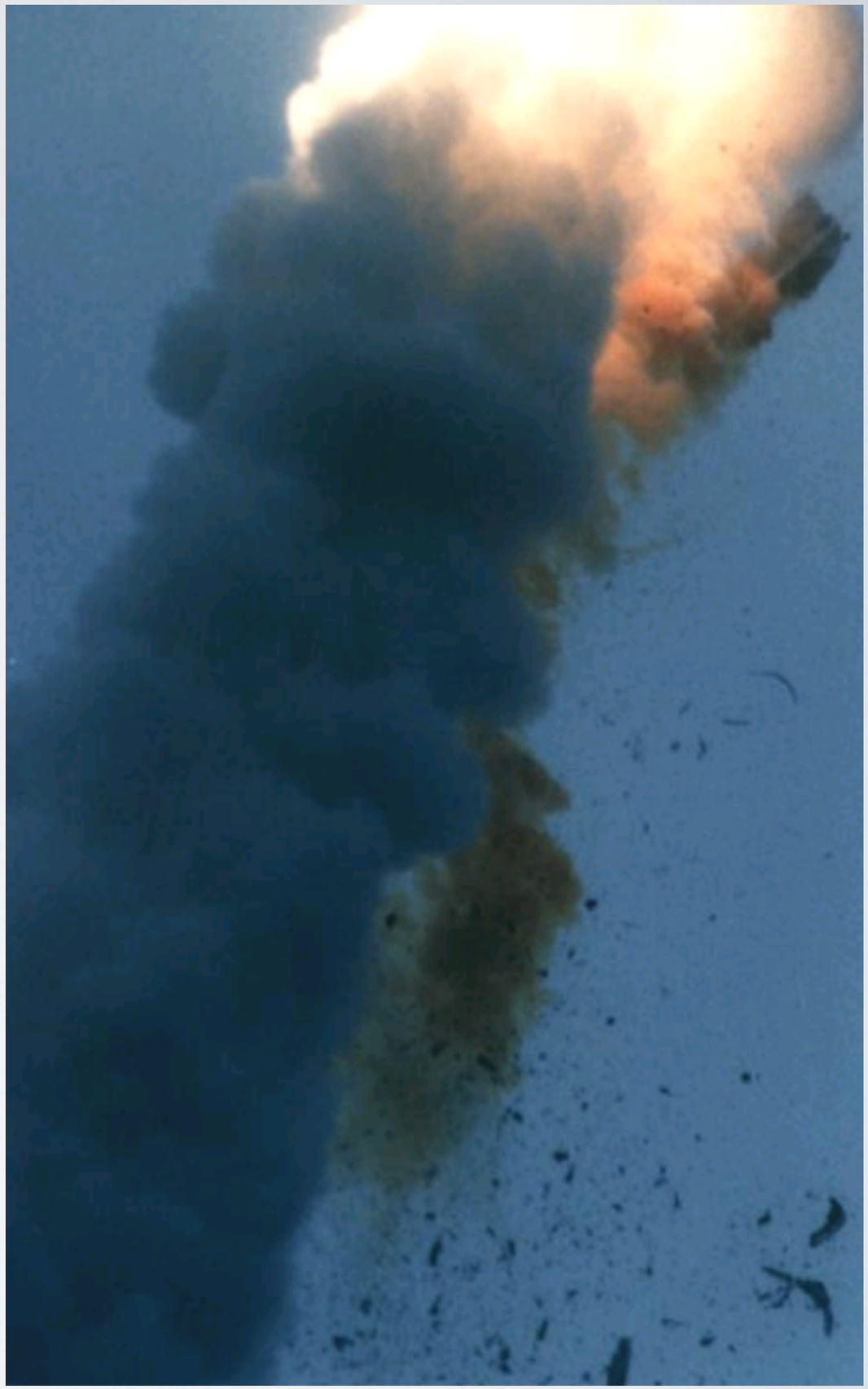
June 1996

ARIANE 5

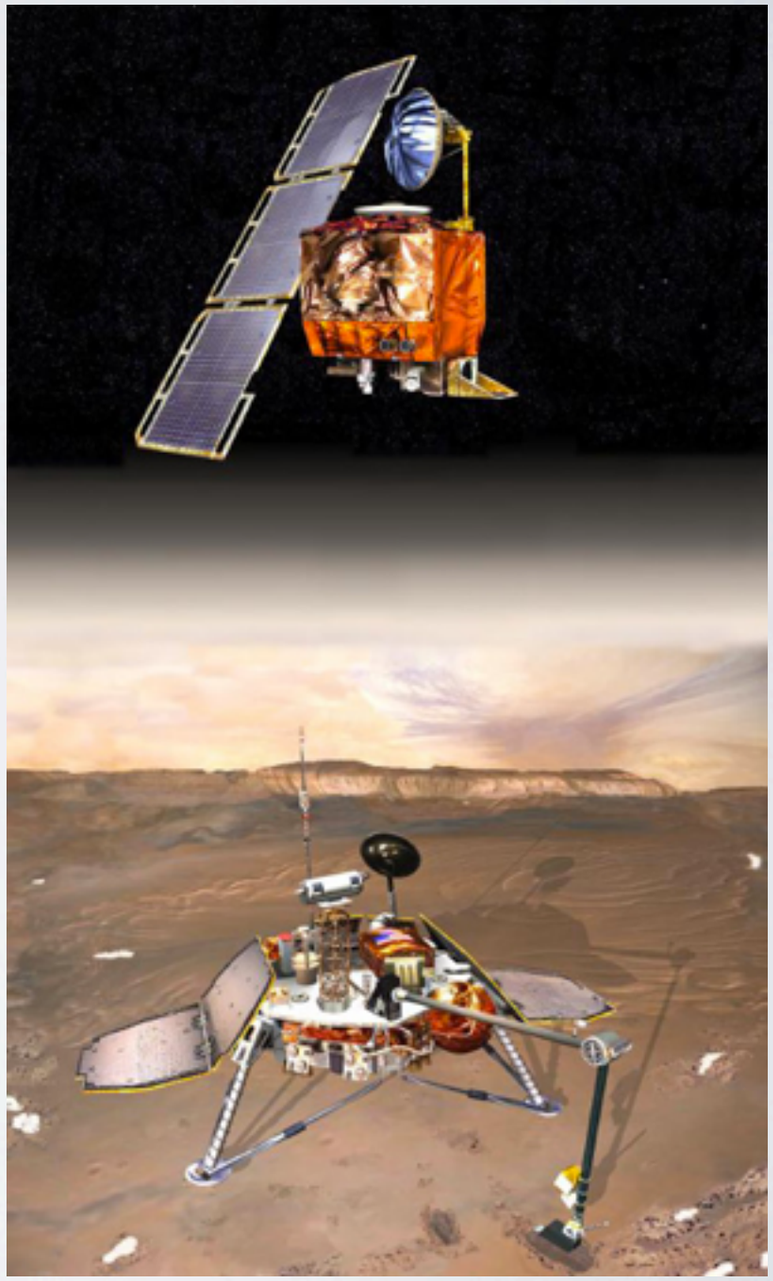
FLIGHT 501

- conversion from 64-bit to 16-bit format, at less than one minute after launch

- estimated cost: 600 mln euro



MARS CLIMATE ORBITER
AND
MARS POLAR LANDER

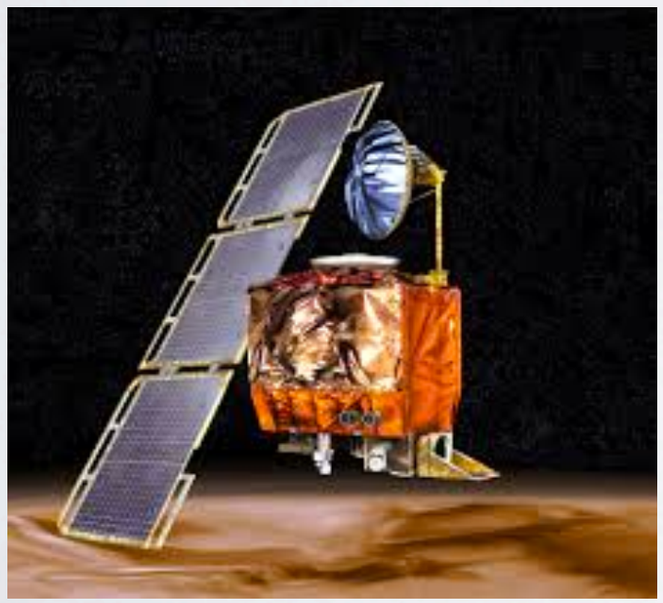


- launched on December 1998 and January 1999
- estimated cost: 327 mln $

MARS CLIMATE ORBITER
AND
MARS POLAR LANDER

- different units (pound, kg) used in different software components

- discrepancy between a planned trajectory and the actual one



September 1999

- software incorrectly interpreted vibrations as surface touchdown

(hypothesis)

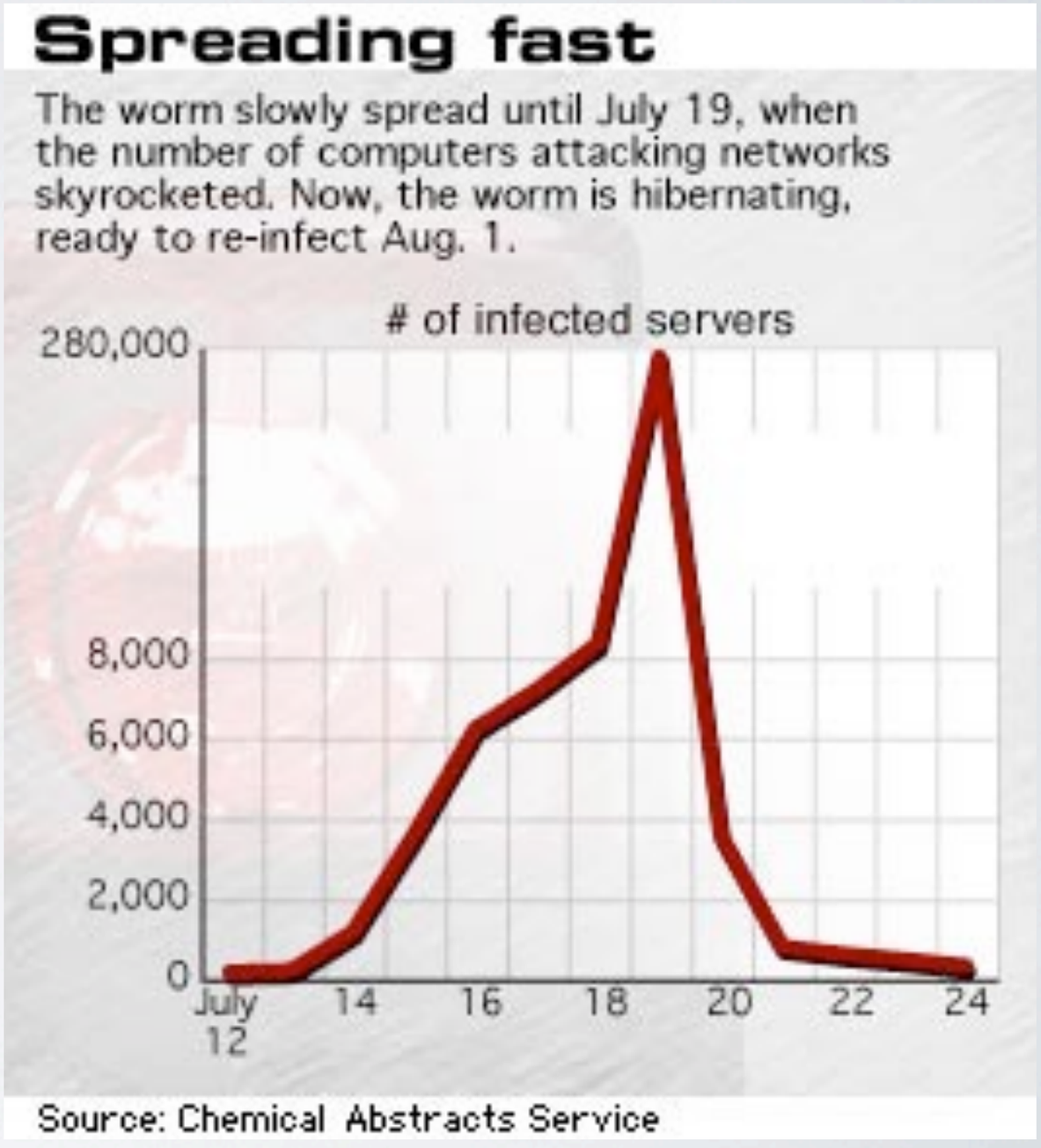
December 1999

CODE RED

`HELLO! Welcome to http://www.worm.com! Hacked By Chinese!`

- buffer overflow in Microsoft Internet Information Server
- estimated cost: 2.5 billion $

July 2001



NORTHEAST BLACKOUT

August 2003

- bug in the alarm system
- operators unaware of overload
- race condition in the controlling software
- local blackout cascaded to massive global one
- 50 mln people affected

HEARTBLEED

- buffer over-read in Open SSL cryptography library

- leakage of keys

- violation of confidentiality

April 2014

HEARTBLEED

- buffer over-read in Open SSL cryptography library

- leakage of keys

- violation of confidentiality

April 2014

SUMMARY

- bugs are costly ...
- ... and often unacceptable (safety critical systems)
- formal verification may help to decrease the number of bugs
- testing proves presence of bugs, while formal verification (sometimes) proves their absence

Success stories

SOFTWARE SUCCESS STORY

SOFTWARE SUCCESS STORY

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)

SOFTWARE SUCCESS STORY

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)
- one of reasons is the complexity of the Windows drivers API

SOFTWARE SUCCESS STORY

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)
- one of reasons is the complexity of the Windows drivers API
- SLAM: automatically checks device drivers for certain correctness properties with respect to the Windows device drivers API

SOFTWARE SUCCESS STORY

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)
- one of reasons is the complexity of the Windows drivers API
- SLAM: automatically checks device drivers for certain correctness properties with respect to the Windows device drivers API
- now part of Windows Driver Development Kit, a toolset for drivers developers

verification of coders ;)

verification of coders ;)

verification of coders ;)

we model-check coders

Assignment: compute equilibrium point

$$k \text{ jest punktem równowagi} \iff \sum_{i=0}^{k-1} a_i = \sum_{i=k+1}^{n-1} a_i$$

Solution:

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= 2 * A[i];
  }

  return -1;
}
```

Solution:

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= 2 * A[i];
  }

  return -1;
}
```

Solution:

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= 2 * A[i];
  }

  return -1;
}
```

kontrprzykład : $\{2^{30}, 0, 2^{30}\}$

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= 2 * A[i];
  }

  return -1;
}
```

kontrprzykład : $\{2^{30}, 0, 2^{30}\}$

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= A[i];
    d -= A[i];
  }

  return -1;
}
```

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= 2 * A[i];
  }

  return -1;
}
```

kontrprzykład : $\{2^{30}, 0, 2^{30}\}$

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= A[i];
    d -= A[i];
  }

  return -1;
}
```

How is it possible?

```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= 2 * A[i];
  }

  return -1;
}
```

kontrprzykład : $\{2^{30}, 0, 2^{30}\}$

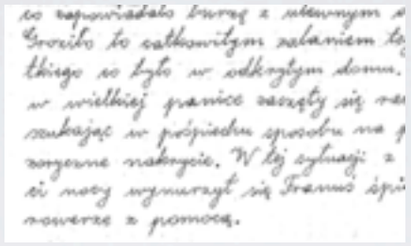
```
int equi(int *A, int n) {
  int i;
  long long d = 0;

  // Obliczenie wartosci d_0
  for (i = 0; i < n; ++i)
    d += A[i];

  // Poszukiwanie punktu równowagi
  for (i = 0; i < n; ++i) {
    if (d == A[i])
      return i;
    d -= A[i];
    d -= A[i];
  }

  return -1;
}
```

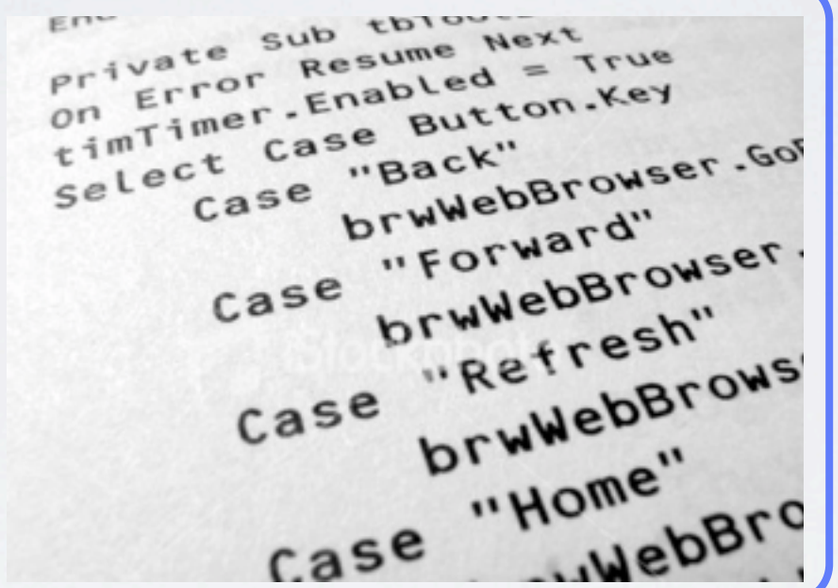
How is it possible?

Due to symbolic approach!

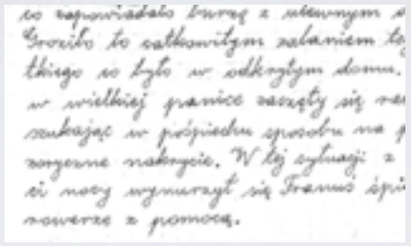
Formal verification

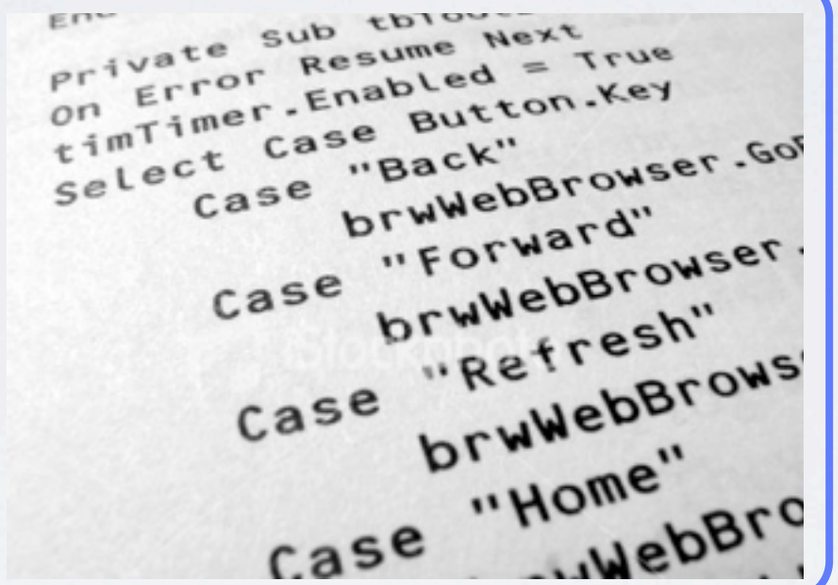
A POSTERIORI VERIFICATION





A POSTERIORI VERIFICATION



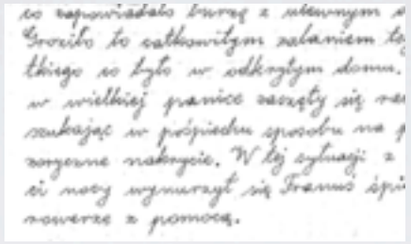


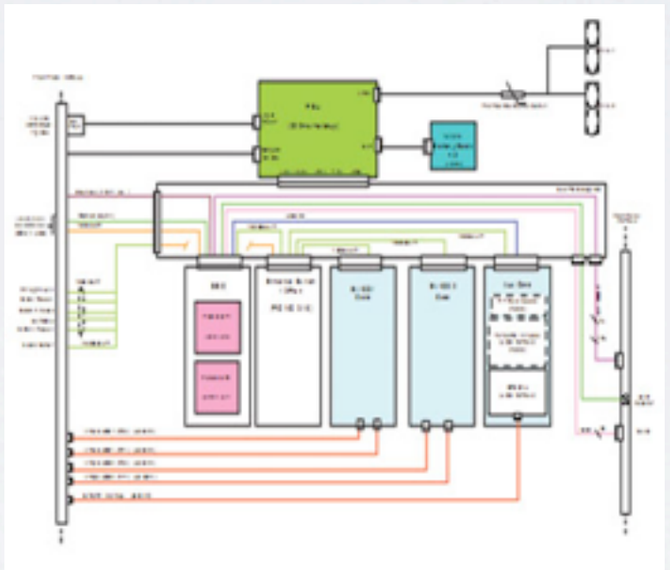
automatically!

✔

✘

A POSTERIORI VERIFICATION





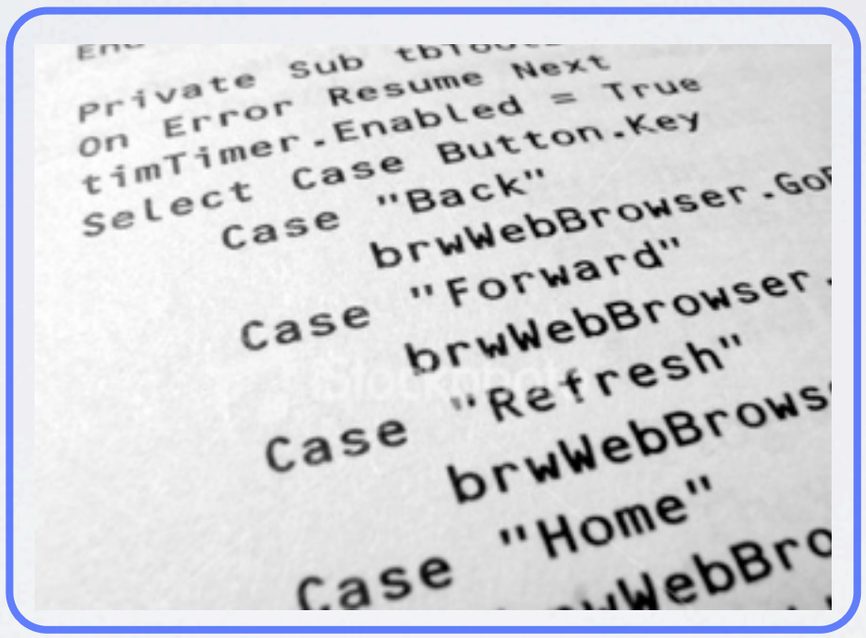
automatically!

✔

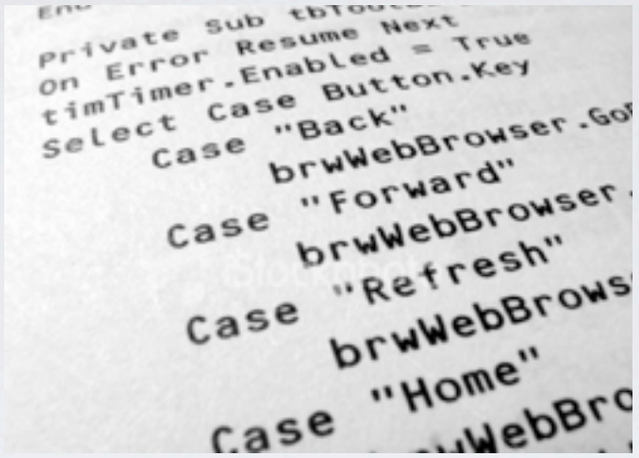
✘

RESTRICTION

every non-trivial question is undecidable !

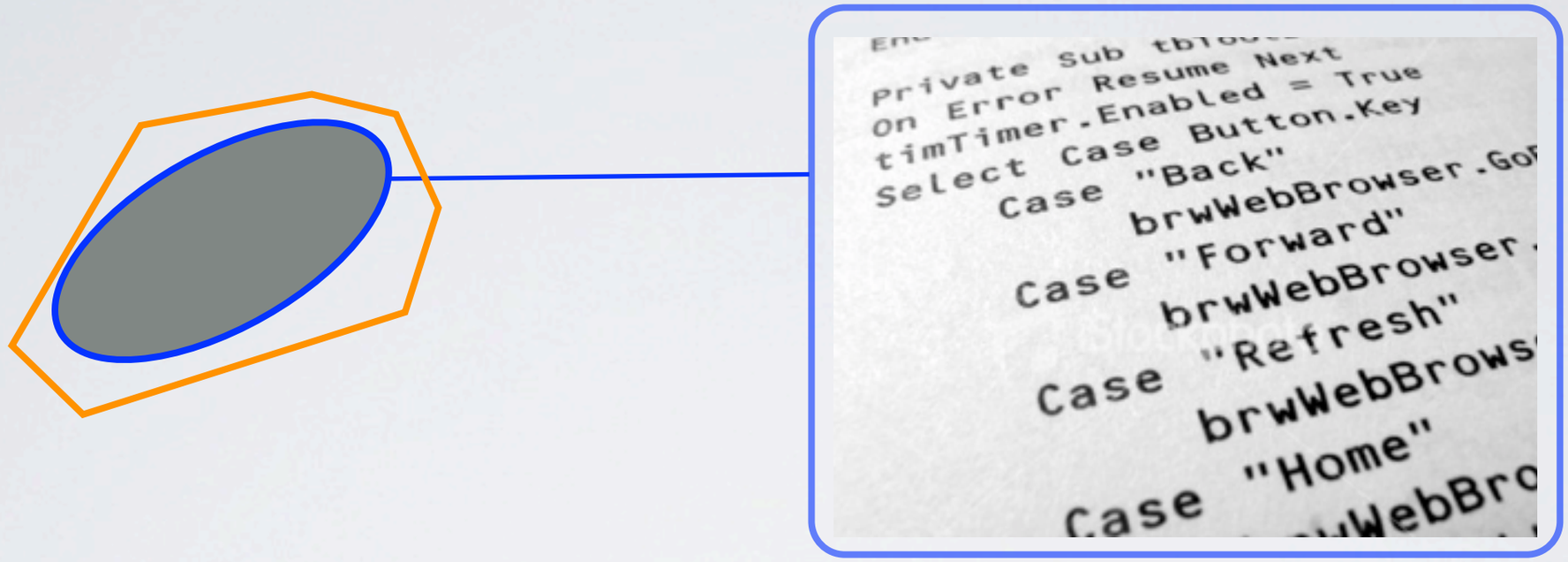


METHOD 1: INTERACTIVE



(proving correctness)

METHOD 2: APPROXIMATION

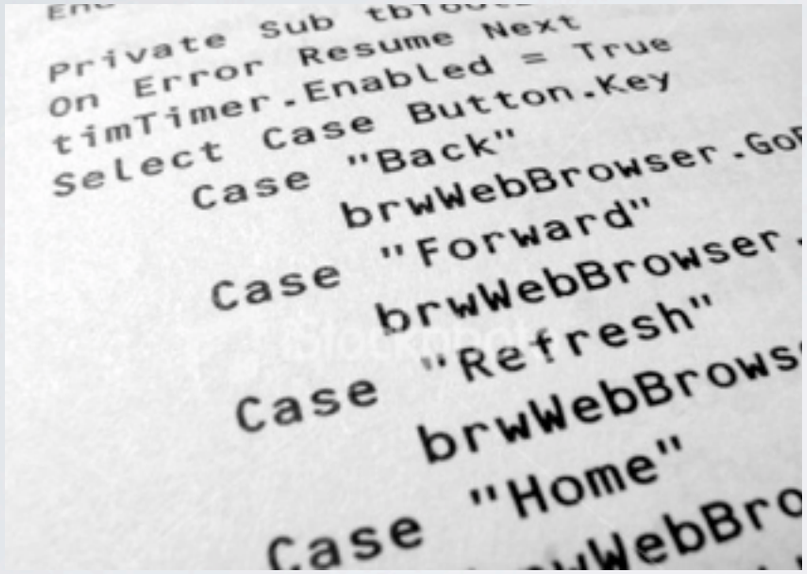


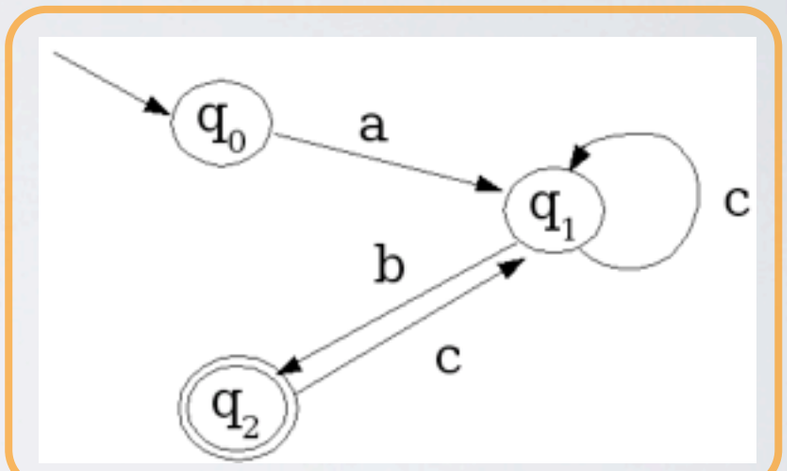
surely ✔

possibly ✘

(static analysis)

METHOD 3: ABSTRACTION





(model checking)

RESTRICTIONS

- Method 1 (interactive): substantial human effort needed
- Method 2 (approximation): false alarms
- Method 3 (abstraction): model is verified, not the system itself

MOTTO

Formal verification aims not at developing correct computer systems ...

MOTTO

Formal verification aims not at developing correct computer systems ...

... but at providing more rigorous methodologies yielding better reliability of designed systems.

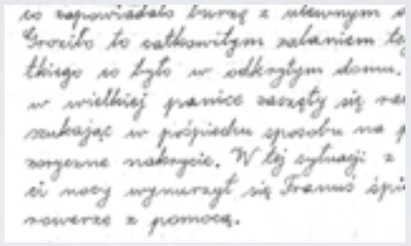
MOTTO

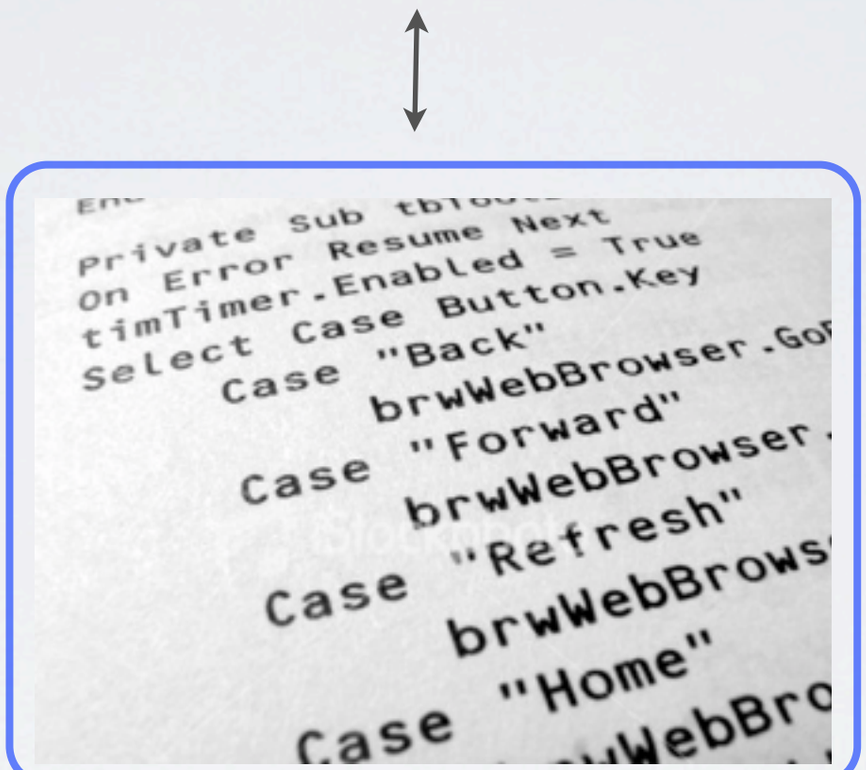
Formal verification aims not at developing correct computer systems ...

... but at providing more rigorous methodologies yielding better reliability of designed systems.

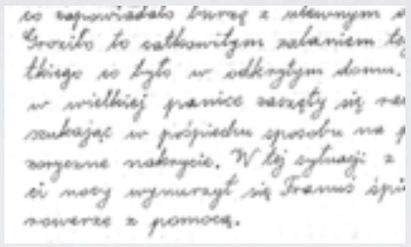
-standard software: 25 bugs per 1000 loc
-good software: 2 bugs per 1000 loc
-spacecraft software: <1 bugs per 10000 loc

VERIFICATION VS VALIDATION

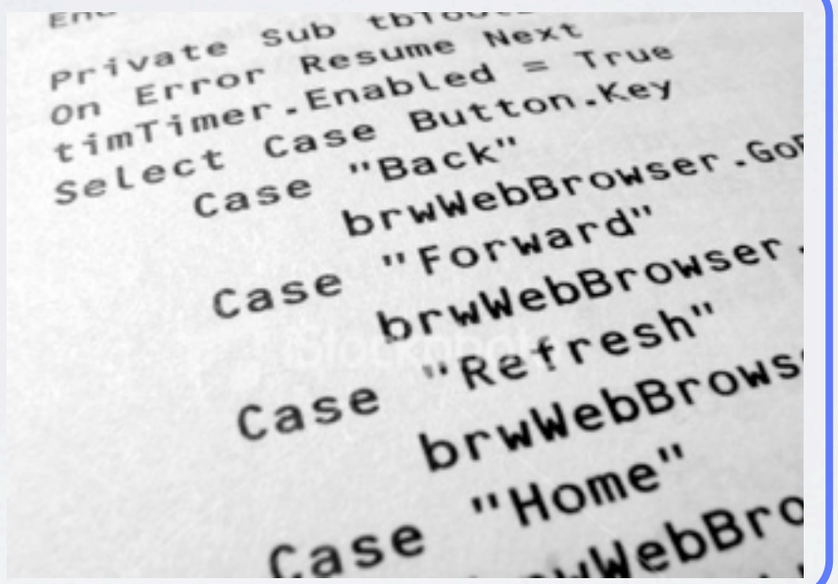




VERIFICATION VS VALIDATION



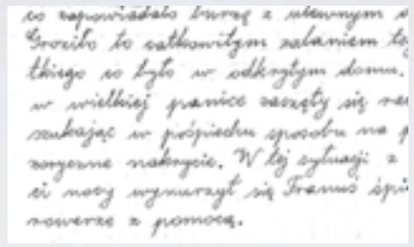
do we build the right thing?

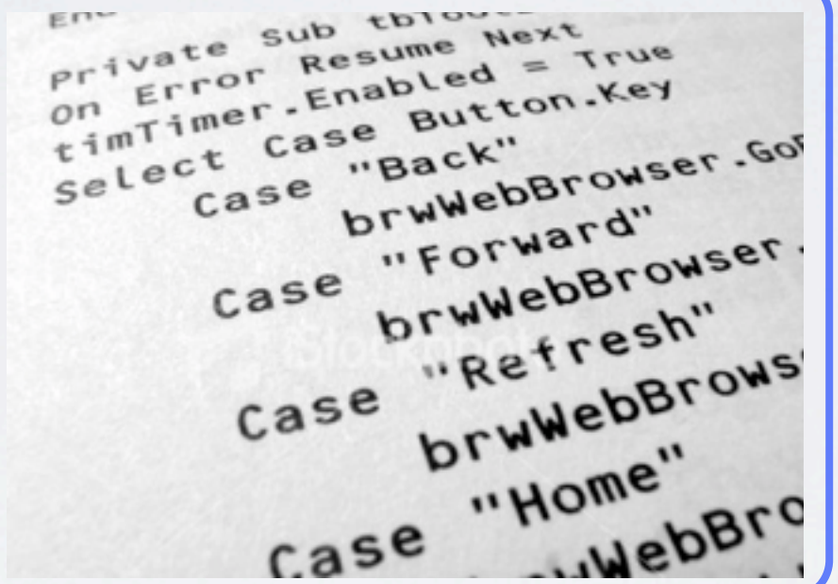


VERIFICATION VS VALIDATION

do we build the thing right?

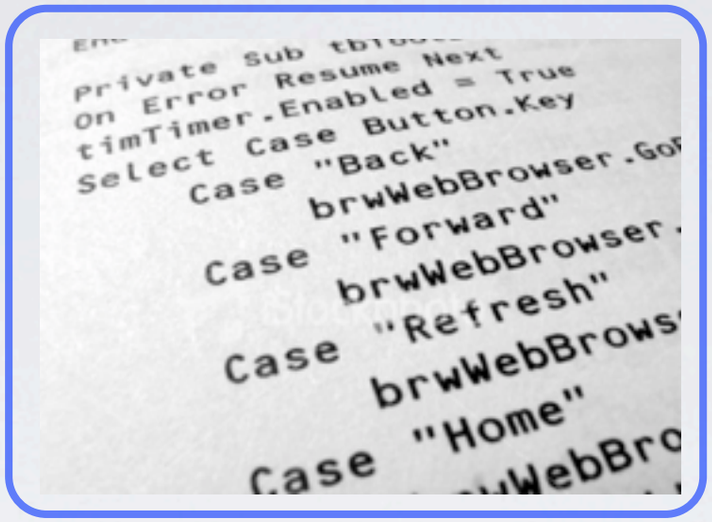
do we build the right thing?





Method 1: Interactive

PROVING CORRECTNESS



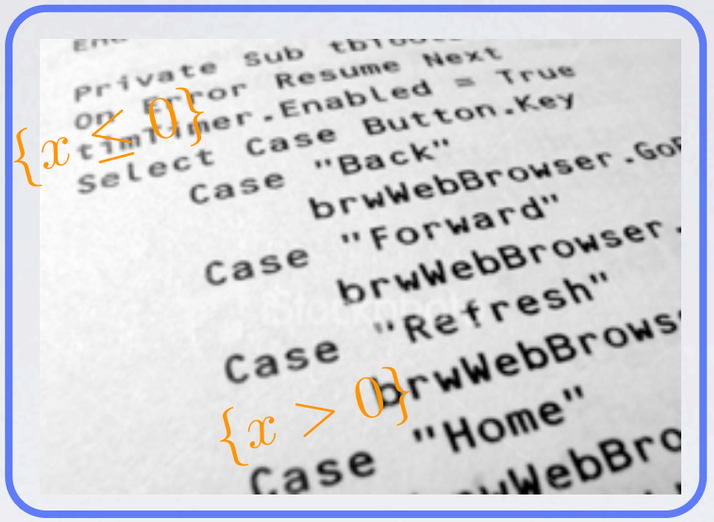
proof obligations

proof assistant tool

proof

?

PROVING CORRECTNESS



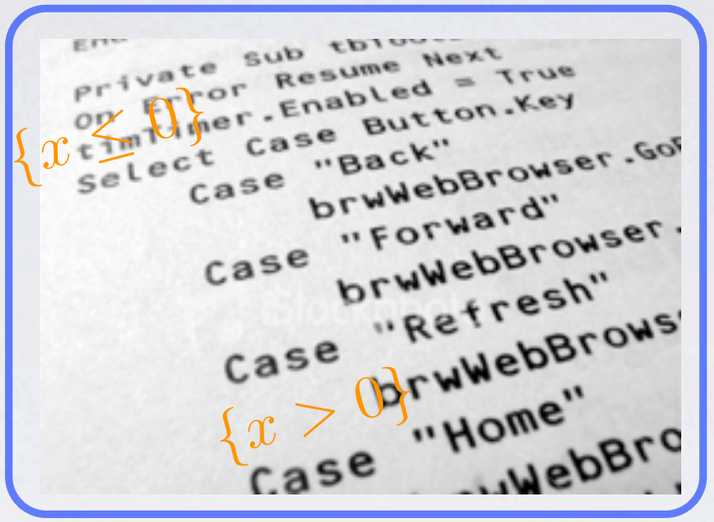
proof obligations

proof assistant tool

proof

?

PROVING CORRECTNESS



proof obligations

proof assistant tool

automatically
or
interactively

proof

?

EXAMPLE - HOARE LOGIC

```
{ a = m  ∧  b = n }
c = 0;
while( b > 0 )
        while( even(b) )
                a := a+a;
                b := b>>1;
        b := b-1 ;
        c := c+a;
{ c = m*n }
```

EXAMPLE - HOARE LOGIC

invariant:
c + a*b = m*n

```
{ a = m  ∧  b = n }
c = 0;
while( b > 0 )
        while( even(b) )
                a := a+a;
                b := b>>1;
        b := b-1 ;
        c := c+a;
{ c = m*n }
```

EXAMPLE - HOARE LOGIC

```
{ a = m  ∧  b = n }
c = 0;
while( b > 0 )
        while( even(b) )
                a := a+a;
                b := b>>1;
        b := b-1 ;
        c := c+a;
{ c = m*n }
```

invariant:
c + a*b = m*n

proof obligations, eg:
c + a*b = m*n ∧ not even(b) ⇒ c+a + a*(b-1) = m*n

PROVING CORRECTNESS - CHARACTERISTIC PROPERTIES

- we analyze decorated source code
- typically only partial automatization is possible
- typically a substantial human expert engagement is necessary
- applicable to small-scale systems
- parametrization/generalization

PIONEERS

PIONEERS

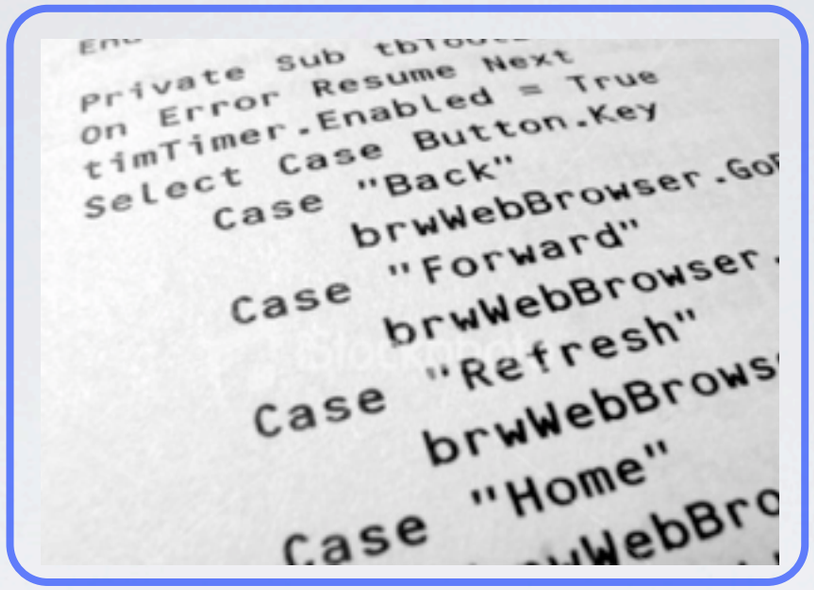
Edsger Dijkstra

Robert Floyd

C.A.R. Hoare

Method II:
Approximation

STATIC ANALYSIS



static analyzer

surely ✔

possibly ✘

STATIC ANALYSIS -
CHARACTERISTIC PROPERTIES

- we analyze source code (control flow diagram)
- approximate analysis - false alarms (false positives)
- typically oriented towards specific properties
- fully automatic
- applicable to large-scale systems

STATIC ANALYSIS - APPLICATIONS

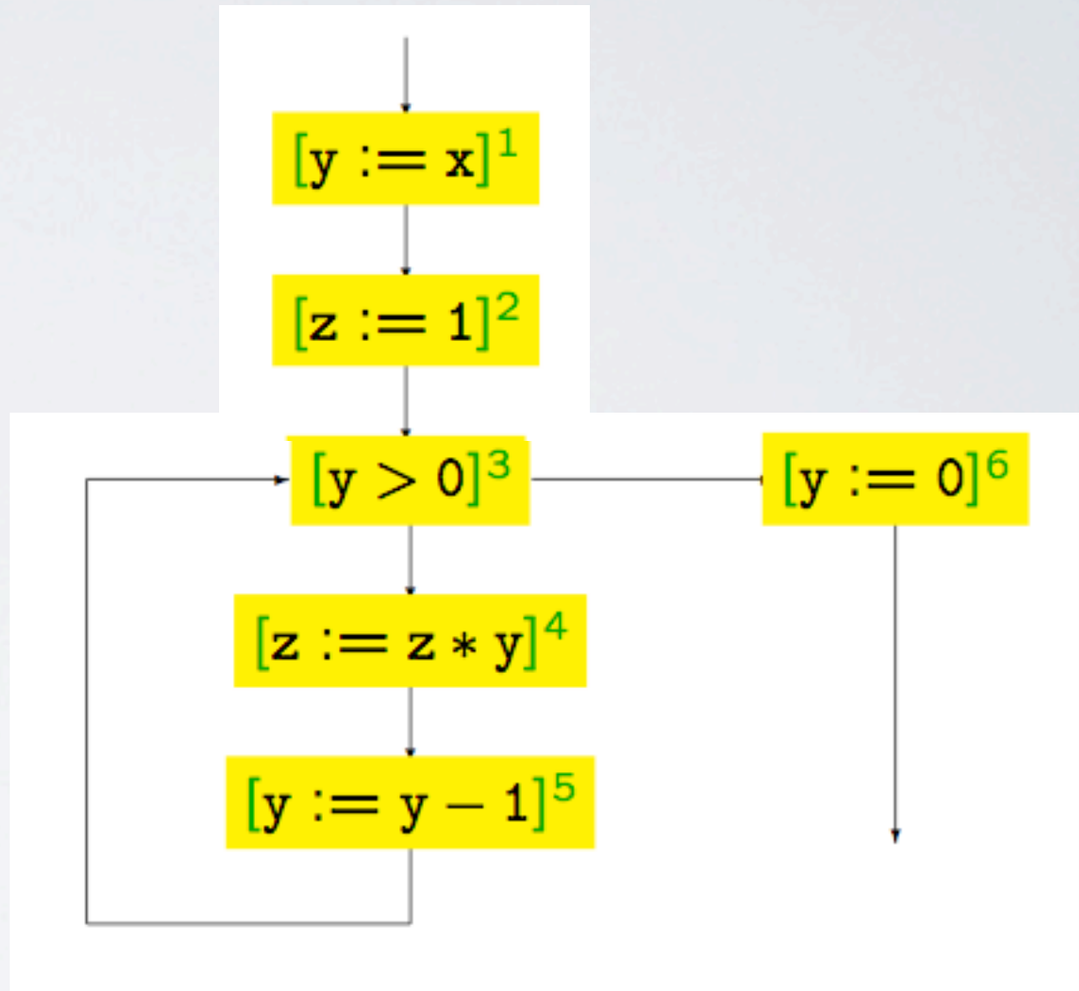
- compiler optimization
- source code quality estimation
- program verification

STATIC ANALYSIS - METHODS

- data flow analysis
- control flow analysis
- type analysis
- shape analysis
- ...
- abstract interpretation

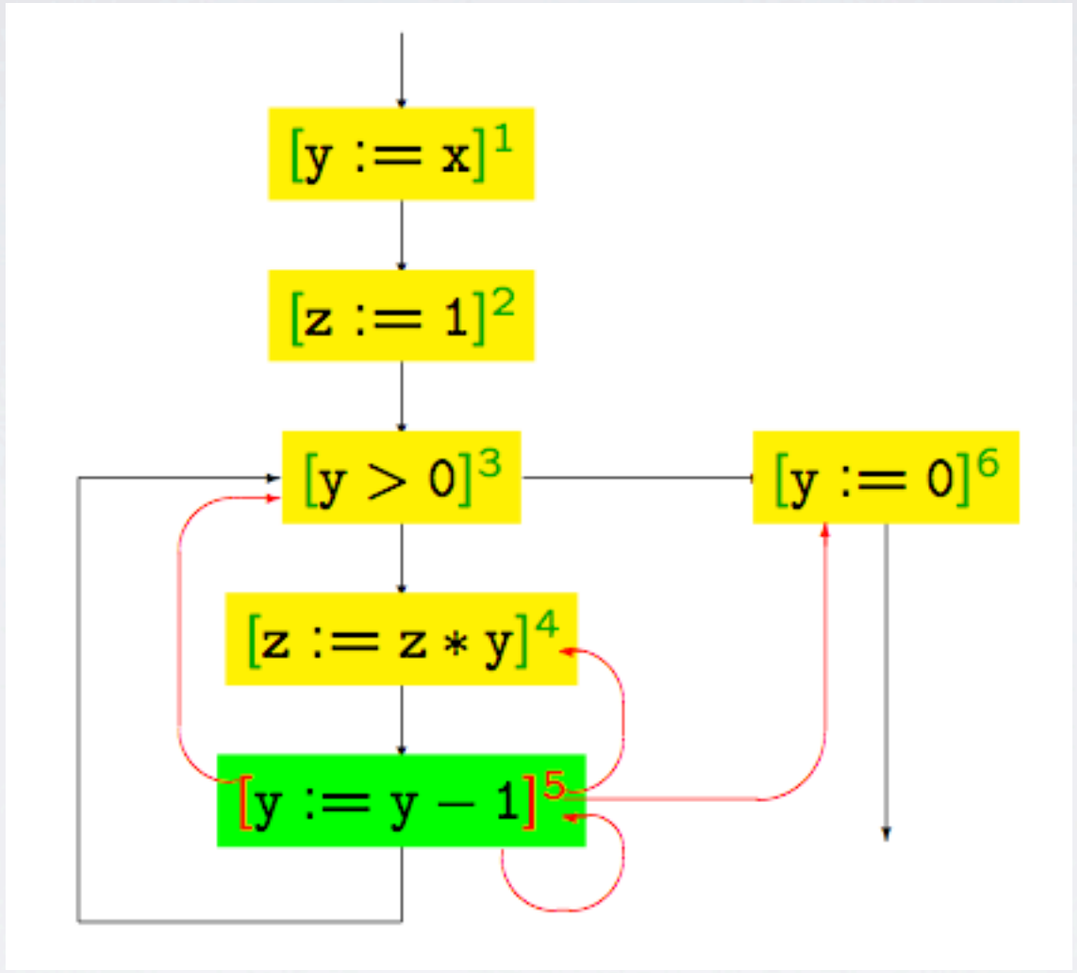
STATIC ANALYSIS - EXAMPLE

$[y := x]^1;$
$[z := 1]^2;$
while $[y > 0]^3$ **do**
  $[z := z * y]^4;$
  $[y := y - 1]^5$
od;
$[y := 0]^6$



[Nielson, Nielson, Hankin 2005]

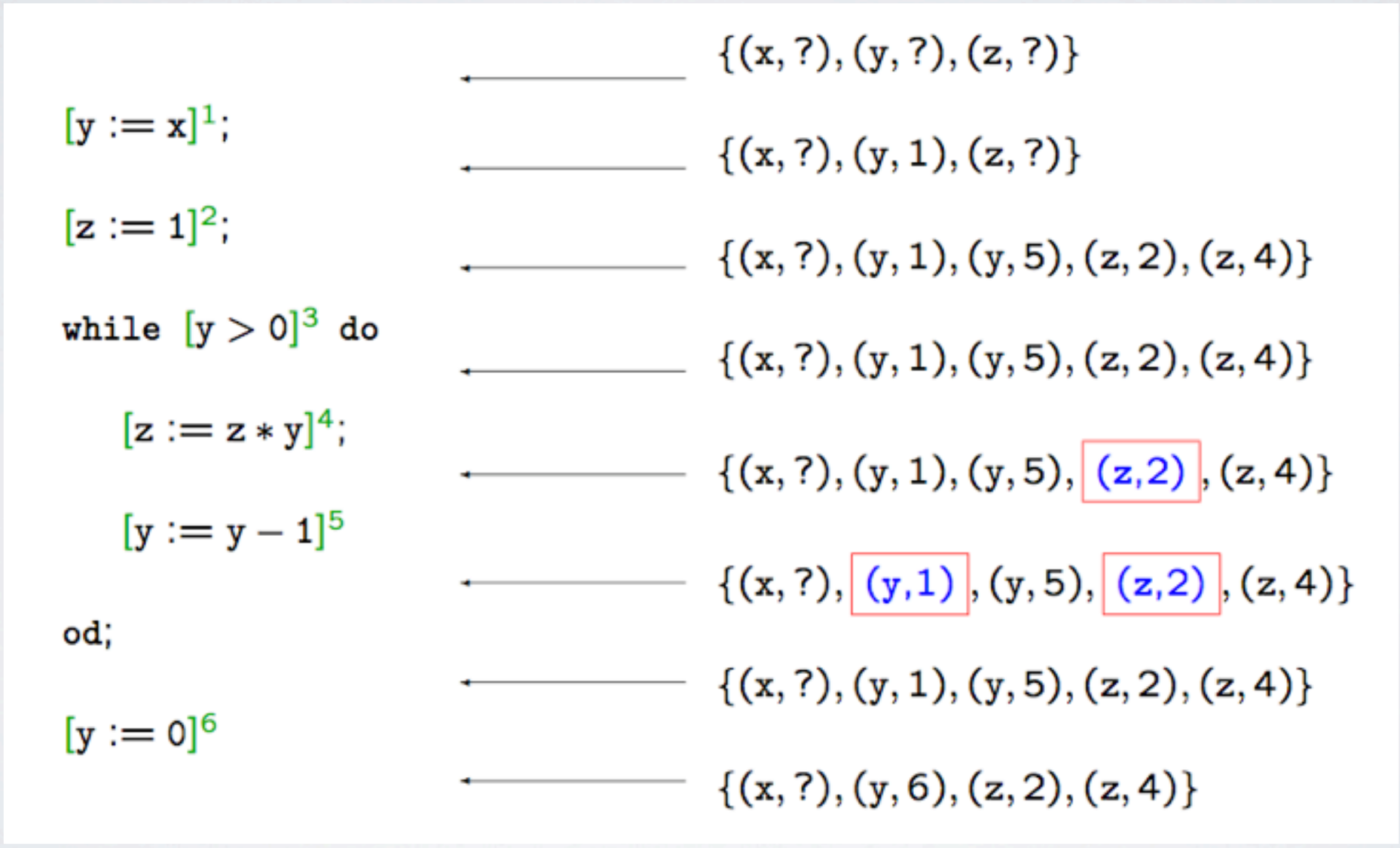
"REACHING" ASSIGNMENTS



[Nielson, Nielson, Hankin 2005]

"REACHING" ASSIGNMENTS

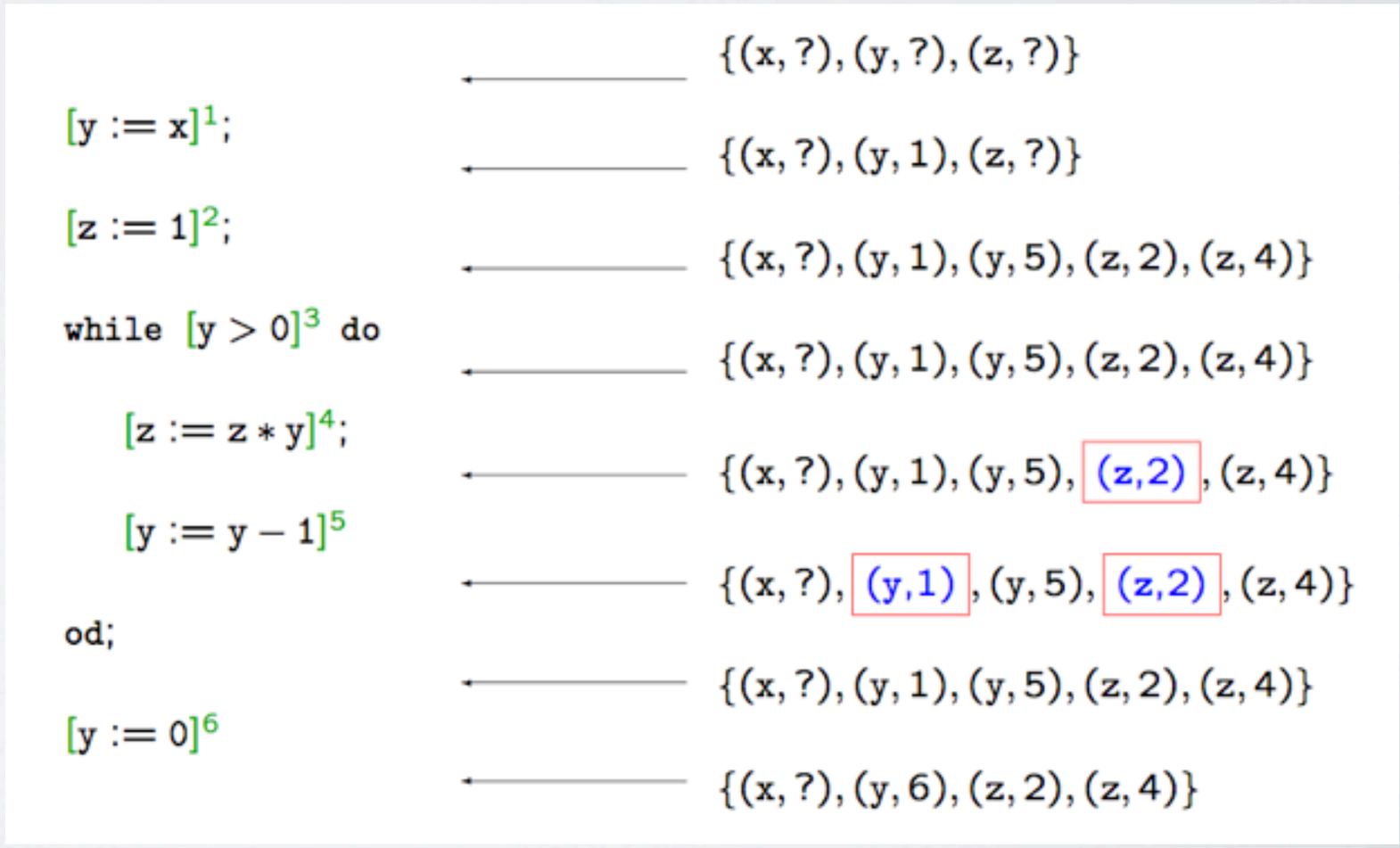
- execution in an abstract domain



[Nielson, Nielson, Hankin 2005]

"REACHING" ASSIGNMENTS

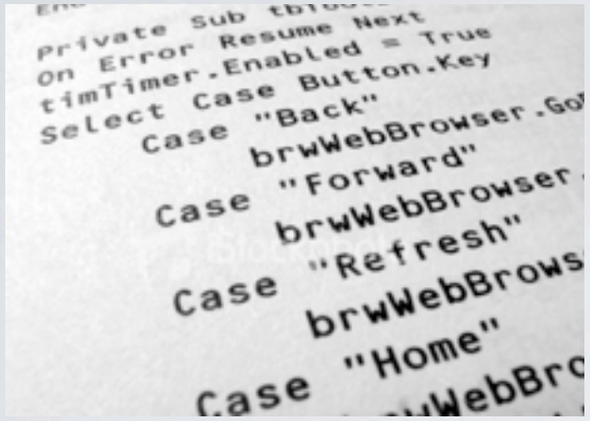
- we formalize the problem as a set of equations
- the least solution
- iterative algorithm

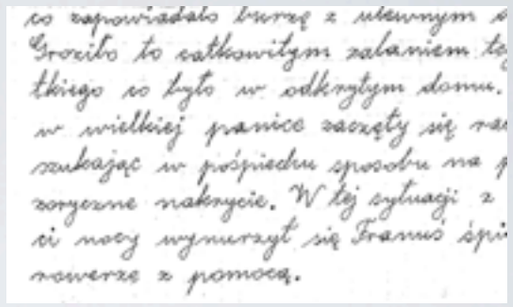


Method III:

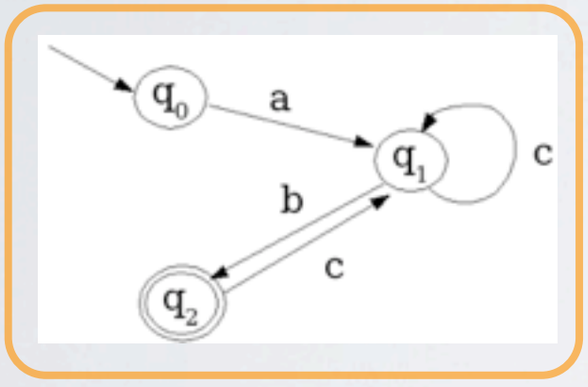
Model checking

MODEL CHECKING





$\exists_X((\forall_x \exists_{y \leq x} y \in X)$



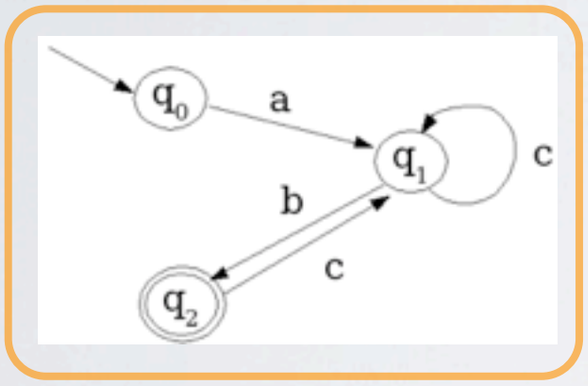
model checker

✔

counterexample

error

MODEL CHECKING



$\exists_X((\forall_x \exists_{y \leq x} y \in X)$

model checker

✔

counterexample

error

MODEL CHECKING

- finite-state model M - possible system's behavior

- property Φ - admissible system's behavior expressed in a temporal logic

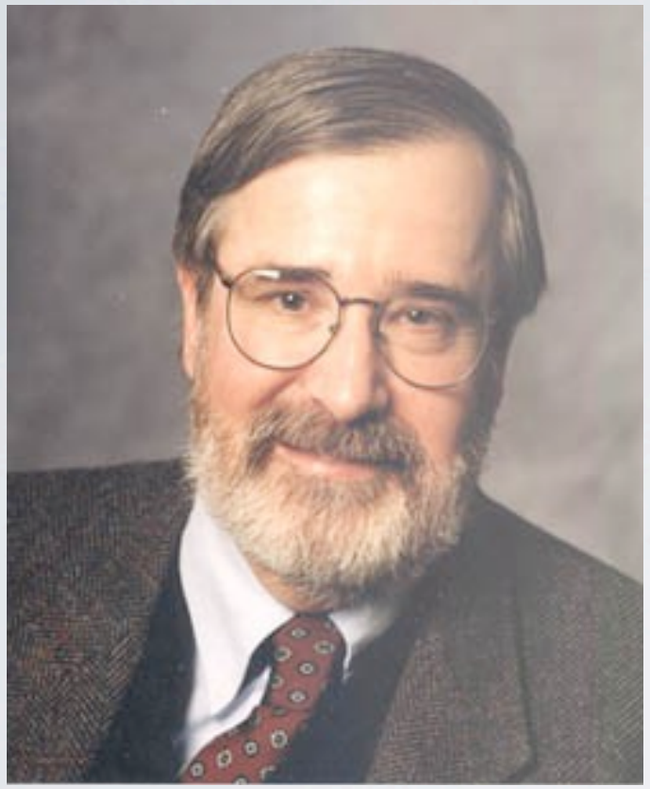
- automatically check

M satisfies Φ

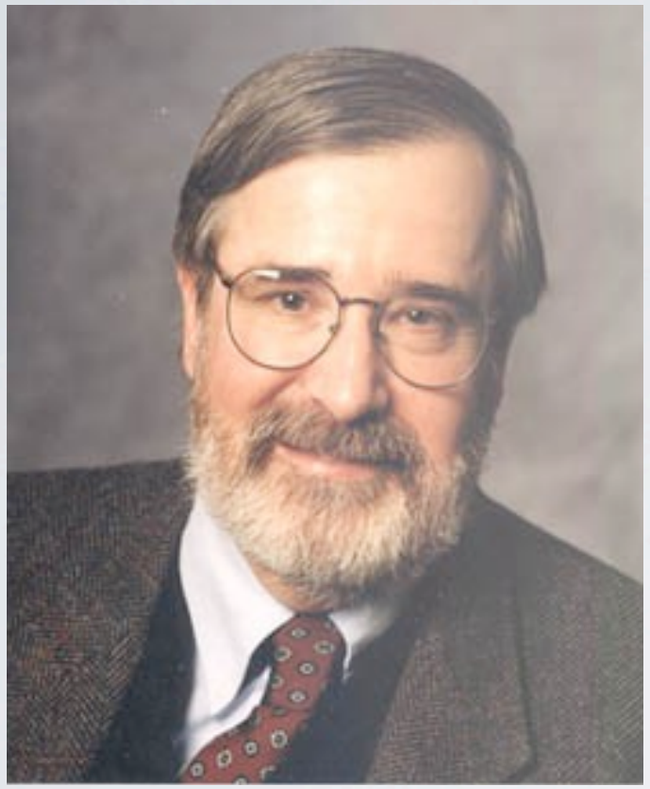
TYPICAL TEMPORAL PROPERTIES

- safety: all reachable states satisfy ϕ
- liveness: eventually ϕ is satisfies
- fairness: ϕ is satisfies infinitely often

TURING AWARD 2007



TURING AWARD 2007

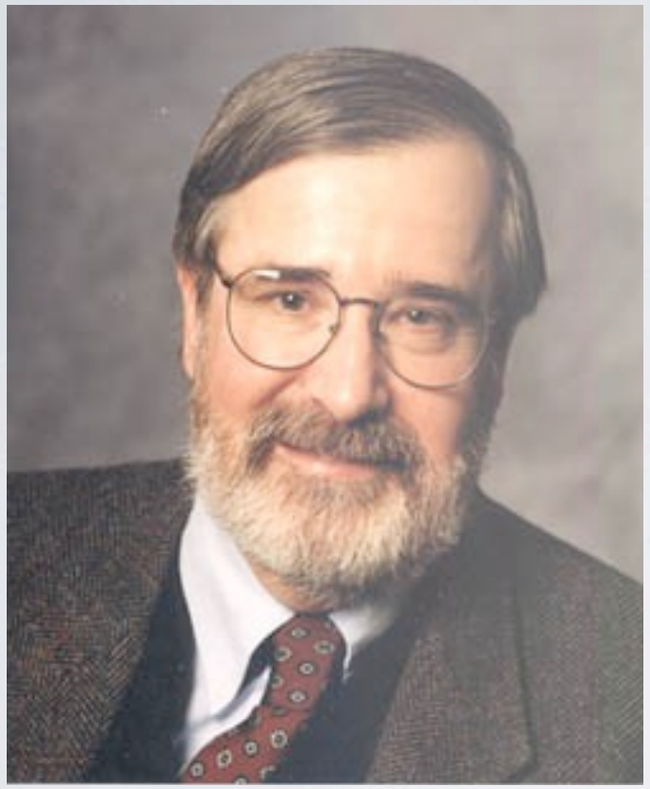


Ed Clarke

Allen Emerson

Joseph Sifakis

TURING AWARD 2007



Ed Clarke

Allen Emerson

Joseph Sifakis

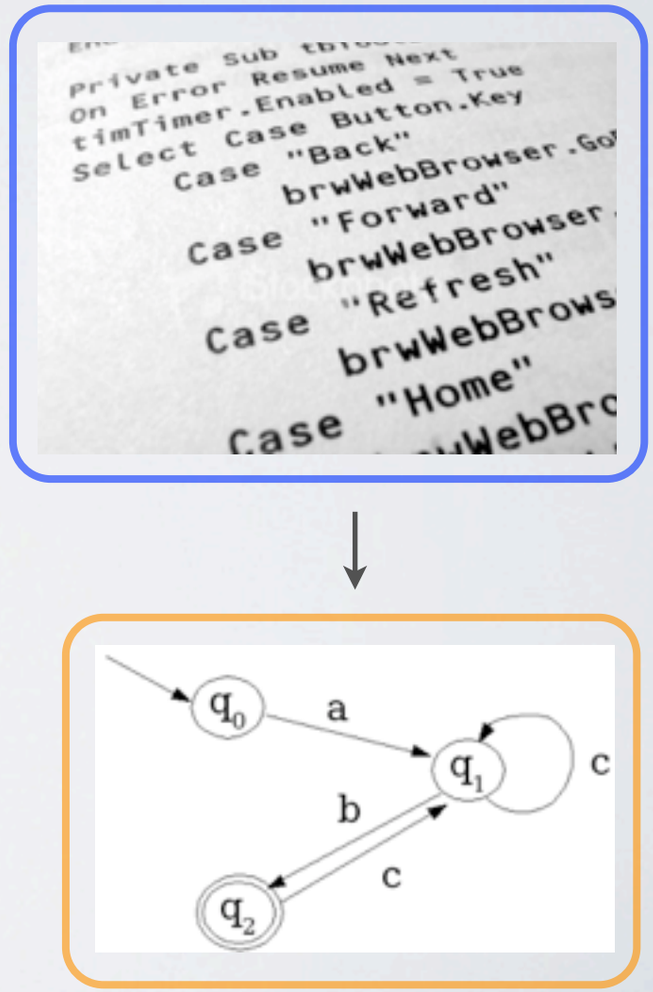
Turing awards
1972, 1978, 1980

MODEL CHECKING - CHARACTERISTIC PROPERTIES

- model of a system (graph of states and transitions)
- analysis of a model via exhaustive state-space exploration
- requirement specification = temporal formula
- (almost) fully automatic
- applicable to small-size models
- in case of negative answer, diagnostic information - counterexample

FROM SYSTEM TO MODEL

- not always fully automatic
- appropriate choice of abstraction level is crucial

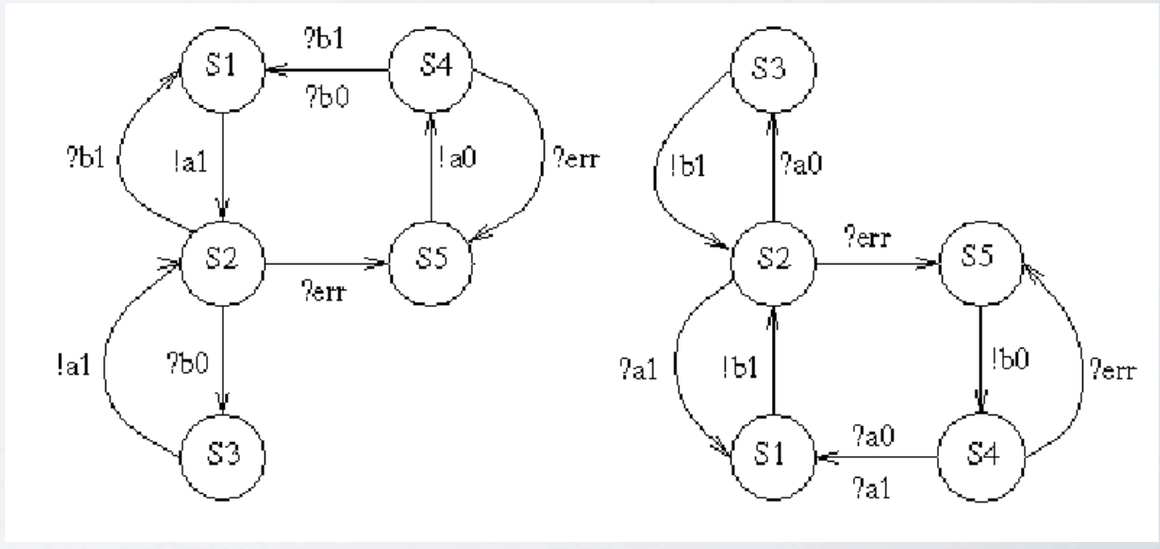


WHAT KIND OF MODEL?

- functional (relational): input/output
- reactive:
  - interaction with environment
  - maybe non-terminating

MODEL = CONTROL + INTERACTION

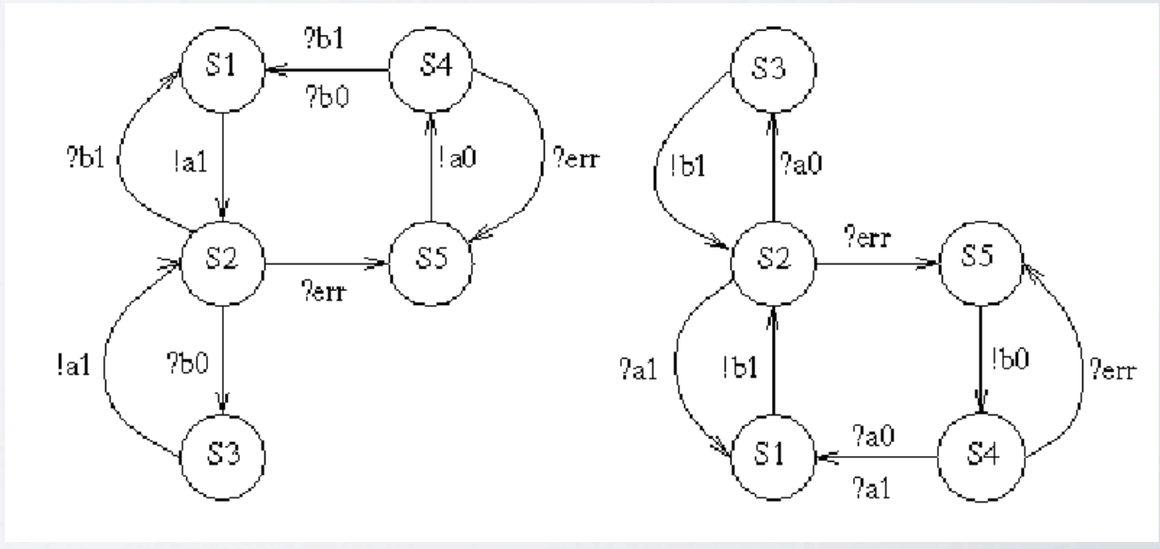
- no complex data structures and computations on them
- abstract (nondeterminism)
- compositional
- concurrency, internal interaction among components (nondeterminism)



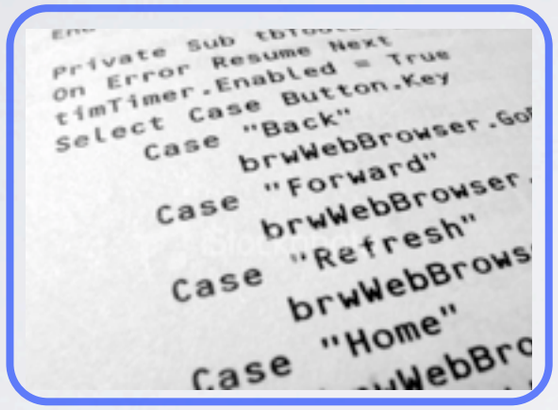
STATE SPACE

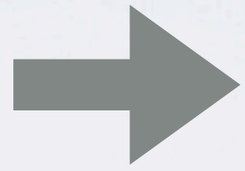
- local state =
  - control point +
  - valuation of variables +
  - content of communication channels +
  - ...

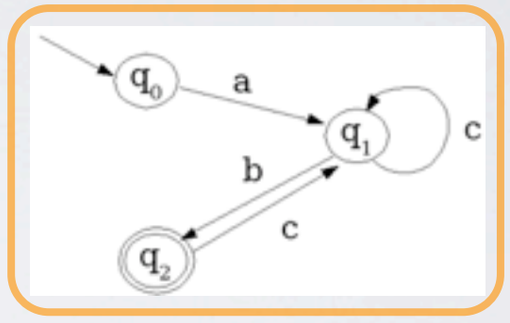
- global state = local states of components + ...



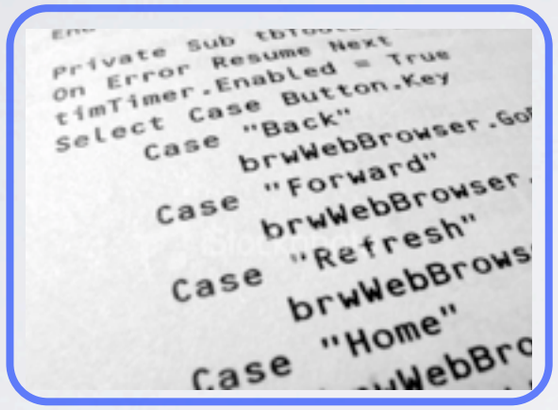
STATE-SPACE EXPLOSION

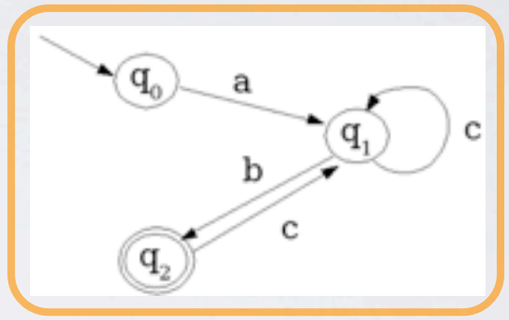




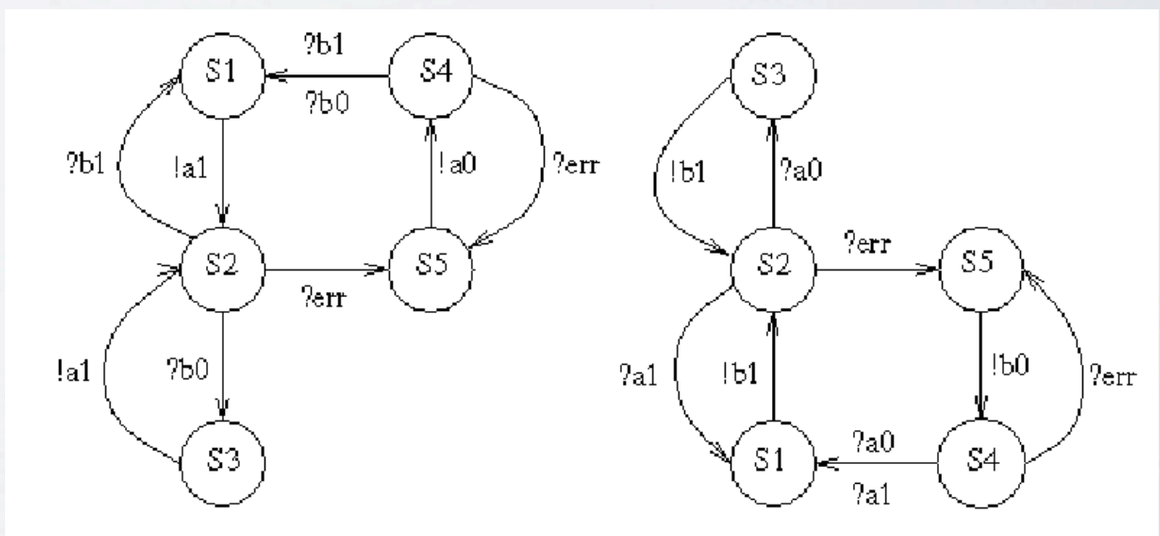


STATE-SPACE EXPLOSION

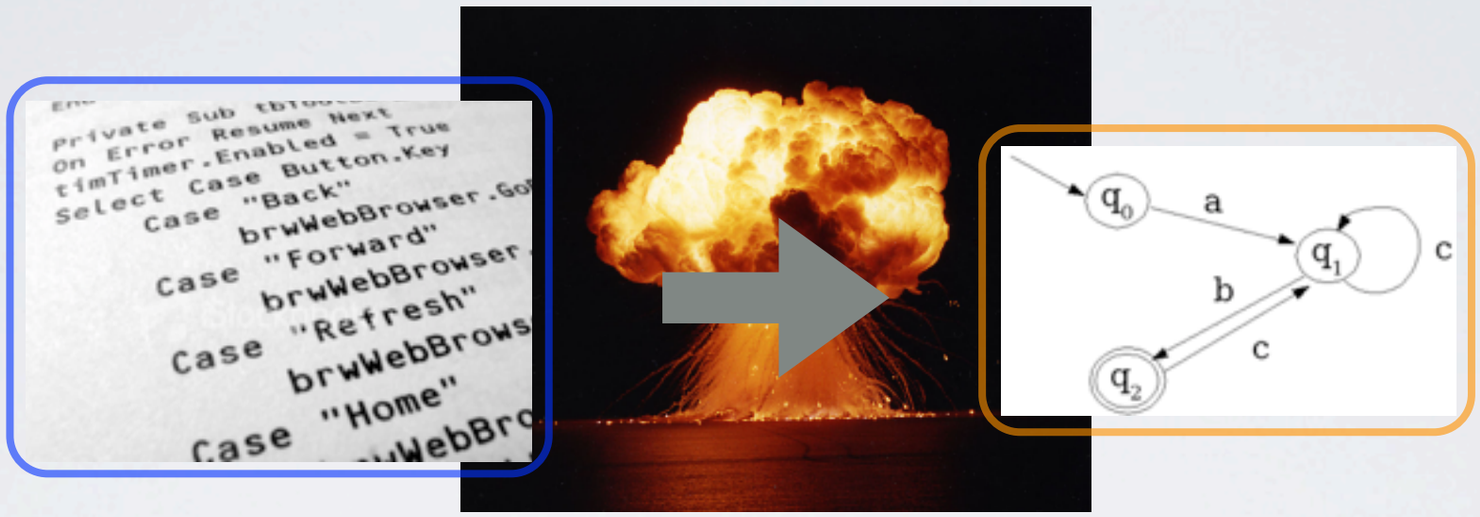




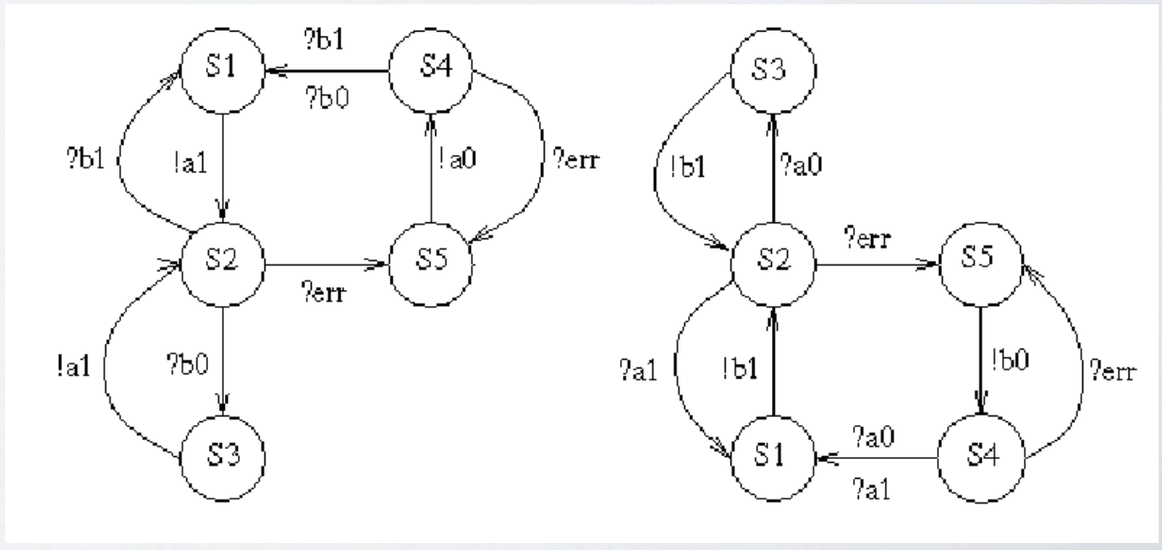
$$G = L_1 \times \ldots \times L_n$$



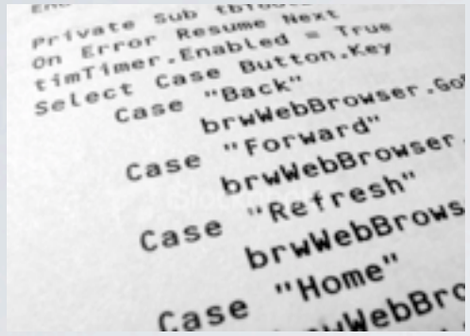
STATE-SPACE EXPLOSION

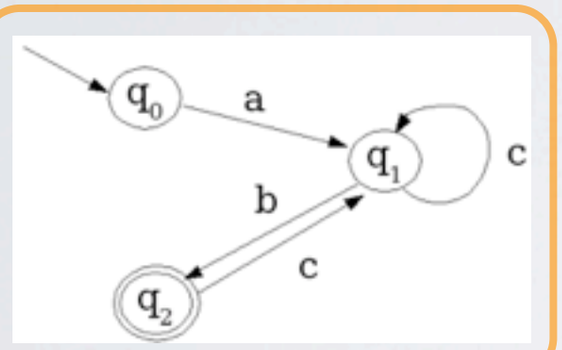


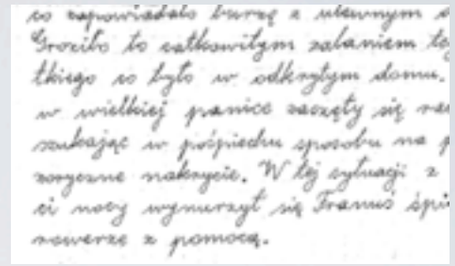
$$G = L_1 \times \ldots \times L_n$$



MODEL CHECKING







$\exists_X((\forall_x \exists_{y \leq x} y \in X)$

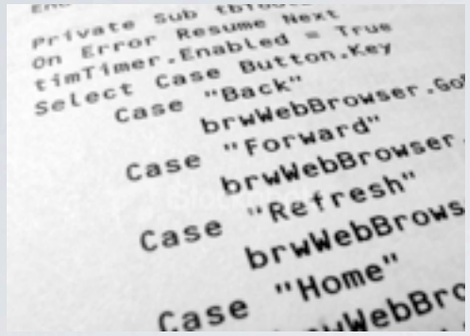
model checker

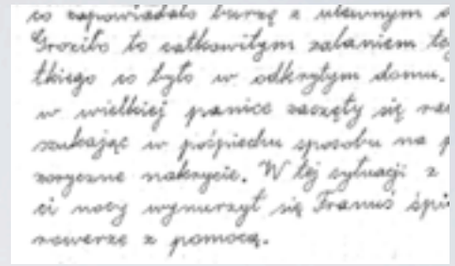
✔

counterexample

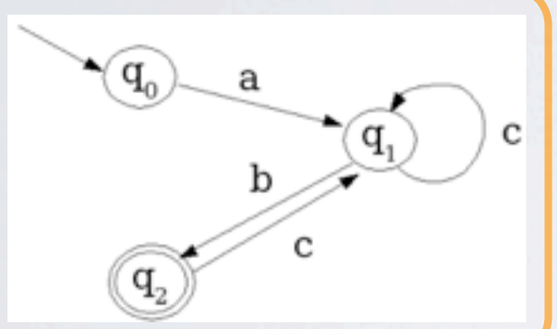
error

MODEL CHECKING





$$\exists_X((\forall_x \exists_{y \leq x} y \in X)$$



model checker

✔

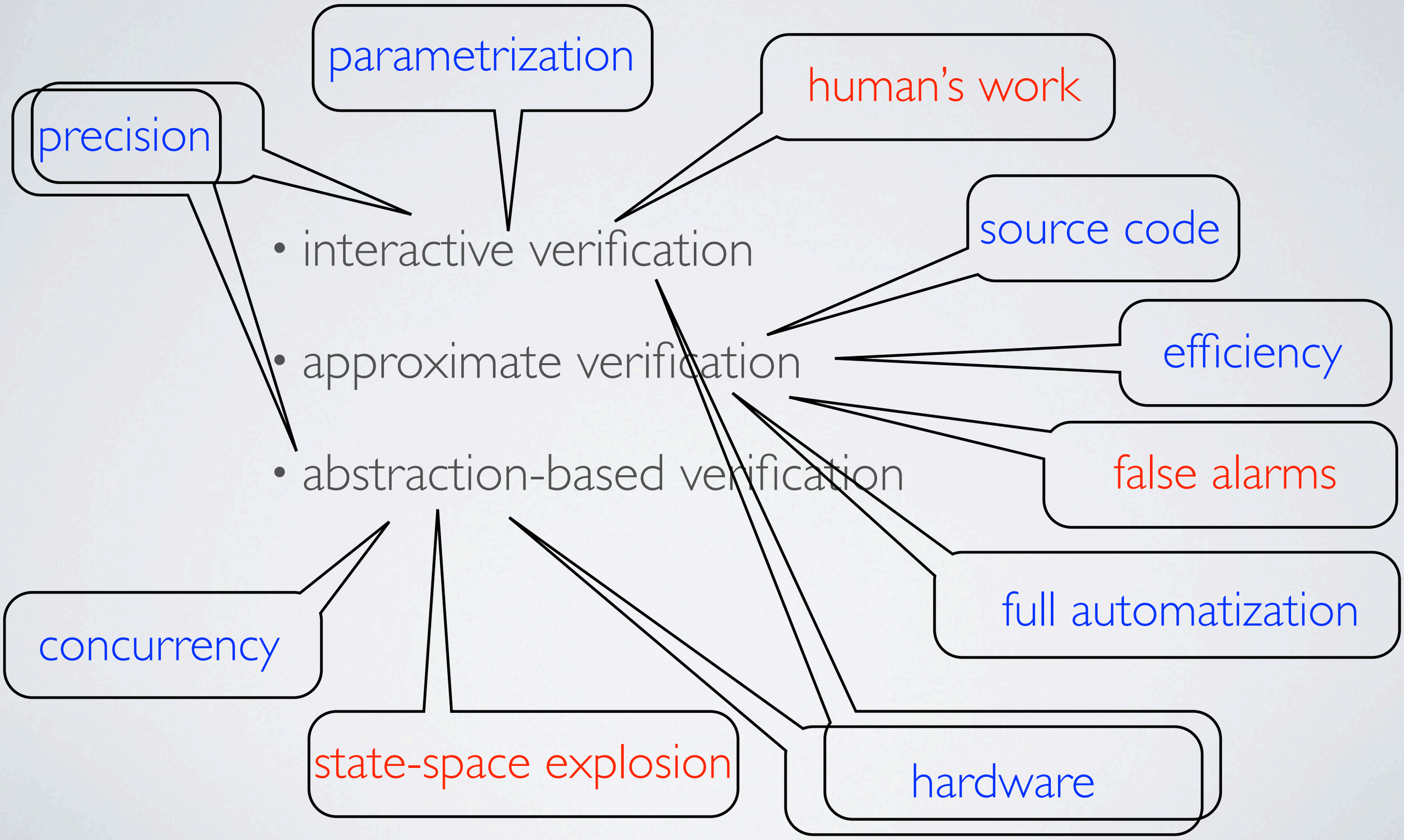
counterexample

error

COMPARISON

- interactive verification
- approximate verification
- abstraction-based verification

COMPARISON


parametrization
precision
human's work
source code
efficiency
false alarms
full automatization
concurrency
state-space explosion
hardware
• interactive verification
• approximate verification
• abstraction-based verification

History

PREHISTORY

- Goldstine, v. Neumann (1947), Turing (1949)

} diagrams, assertions

- Floyd (1967), Hoare (1969), Dijkstra (1976)
- Pratt, Harel (1976-79): dynamic logic of programs
- Owicki, Gries (1976): Hoare's logic for concurrent programs
- Kamp (1968): LTL, Pnueli (1977): application in verification
- 70': static analysis in compiler optimization
- (1979) lint - static analysis of C programs
- (1971) Boyer-Moore theorem prover

PREHISTORY

- Goldstine, v. Neumann (1947), Turing (1949)

} diagrams, assertions

- Floyd (1967), Hoare (1969), Dijkstra (1976)
- Pratt, Harel (1976-79): dynamic logic of programs
- Owicki, Gries (1976): Hoare's logic for concurrent programs
- Kamp (1968): LTL, Pnueli (1977): application in verification
- 70': static analysis in compiler optimization
- (1979) lint - static analysis of C programs
- (1971) Boyer-Moore theorem prover

Turing award 1996

HISTORY (80')

- Clarke, Emerson (1980), Ben-Ari, Manna, Pnueli (1981): CTL*
- Clarke, Emerson (1981), Queille, Sifakis (1982): invention of model checking
- EMC: tens of thousands of states
- 80': proof assistants, applications in verification:
    - Boyer-Moore, Isabelle, HOL, PVS, Coq, Mizar, ...

HISTORY (90')

- Clarke, McMillan, and others (1988-1990): symbolic model checking based on OBDDs
    - SMV: 10^20 ... 10^50 states (circuits)
- (1994-95) commercial tools:
    - model checkers, proof assistants
- Clarke, Biere and others (1998-99): bounded model checking based on SAT
- Valmari, Peled, Godefroid (1990-1994): partial order reductions

HISTORY (00')

- development of methods based on SAT and SMT
- software model checking (abstractions)
- tools (examples for C and Java):
  - proving correctness: ESC/Java2, KeY
  - static analysis: FindBugs, PMD, Splint, Coverity, SLAM
  - model checking: CBMC, Java Pathfinder, Bandera, Bogor, BLAST, Magic
- timed and probabilistic systems

APPLICATION AREAS OF MODEL CHECKING

APPLICATION AREAS OF MODEL CHECKING

- hardware (NuSMV)

APPLICATION AREAS OF MODEL CHECKING

- hardware (NuSMV)
- protocols, system software, drivers (Spin)

APPLICATION AREAS OF MODEL CHECKING

- hardware (NuSMV)
- protocols, system software, drivers (Spin)
- software (CBMC)

APPLICATION AREAS OF MODEL CHECKING

- hardware (NuSMV)
- protocols, system software, drivers (Spin)
- software (CBMC)
- time-dependent systems (UPPAAL)

APPLICATION AREAS OF MODEL CHECKING

- hardware (NuSMV)
- protocols, system software, drivers (Spin)
- software (CBMC)
- time-dependent systems (UPPAAL)
- probabilistic systems (PRISM)

APPLICATION AREAS OF MODEL CHECKING

- hardware (NuSMV)
- protocols, system software, drivers (Spin)
- software (CBMC)
- time-dependent systems (UPPAAL)
- probabilistic systems (PRISM)
- systems biology (PRISM)

BOUNDARIES

- frontiers between approaches are not rigid
- combining model checking with static analysis and with correctness proving
- initial (light) static analysis preceding (heavy) model checking
- model checking as correctness proving, or as static analysis

The following lectures

FUNDAMENTALS OF MODEL CHECKING

- temporal logics: LTL, CTL, CTL*
- LTL model checking via translation to omega-automata
- partial order reductions for LTL
- CTL symbolic model checking using OBDDs
- LTL bounded model checking using SAT
- abstractions, CEGAR

WHAT IS NOT COVERED?

- tuning general methodologies to specific application domains
- inclusion of formal verification into the development cycle of computer systems
- verification process management
- applications to realistic systems
- heuristics for efficiency
- ...

OTHER APPROACHES

- dynamic analysis of programs
- testing/simulations, test coverage measures
- source code quality metrics (code quality management)
- source code audit
- correct by design: systematic construction of correct systems
- ...

PREREQUISITES

- logic, set theory (e.g. fixed points theorems)
- automata theory
- models of concurrent systems
- graph algorithms