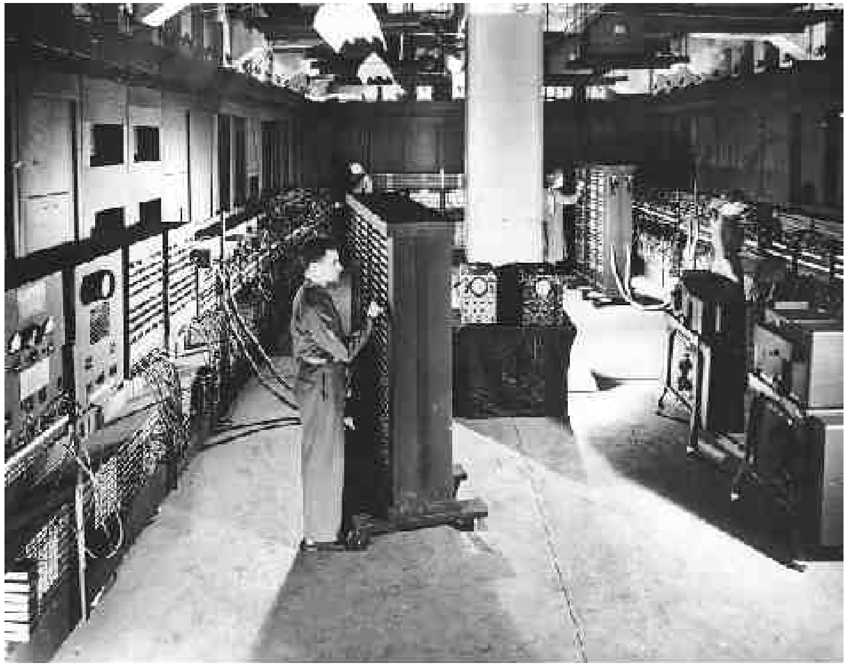
The Long-Standing Software Safety and Security Problem

What is (or should be) the essential preoccupation of computer scientists?

The production of reliable software, its maintenance and safe evolution year after year (up to 20 even 30 years).

Computer hardware change of scale

The 25 last years, computer hardware has seen its performances multiplied by $\mathbf{10^4}$ to $\mathbf{10^6/10^9}$;



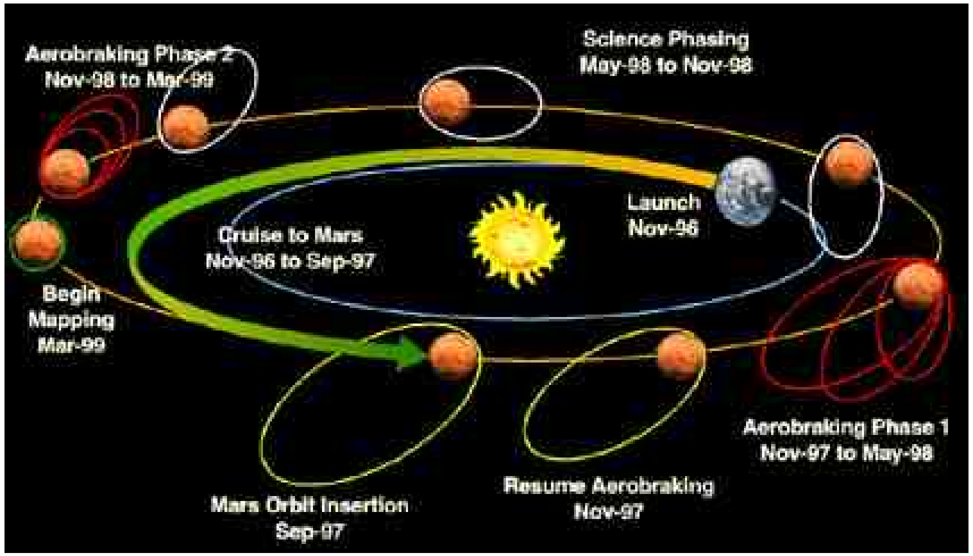
ENIAC (5000 flops)

Intel/Sandia Teraflops System (10^{12} flops)

The information processing revolution

A scale of 10^6 is typical of a significant **revolution**:

- Energy: nuclear power station / Roman slave;
- Transportation: distance Earth — Mars / Boston — Washington

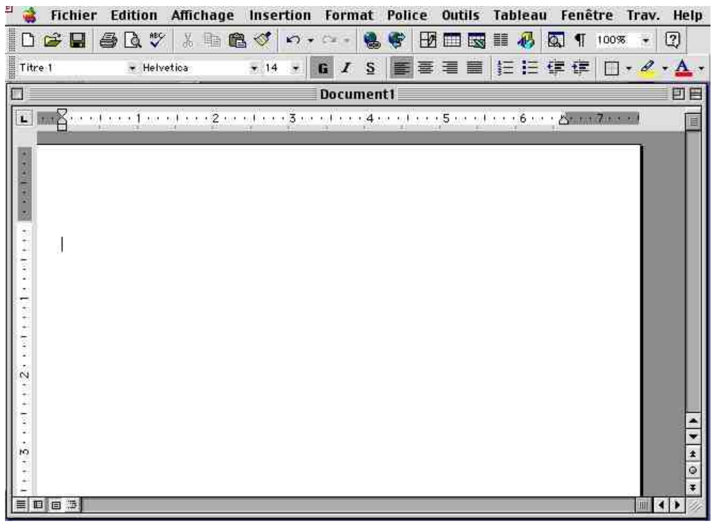


Computer software change of scale

– The size of the programs executed by these computers has grown up in similar proportions;

– **Example 1** (modern text editor for the general public):

- $>$ 1 700 000 lines of C [1];
- 20 000 procedures;
- 400 files;
- $>$ 15 years of development.

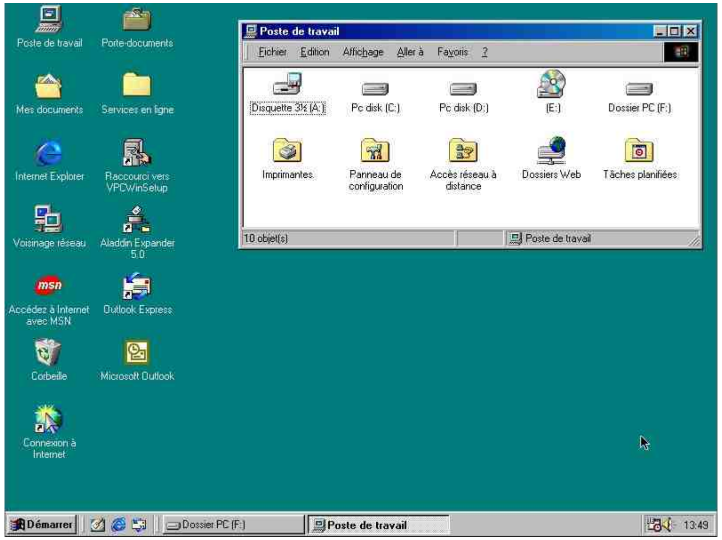


[1] full-time reading of the code (35 hours/week) would take at least 3 months!

Computer software change of scale (cont'd)

– **Example 2** (professional computer system):

- 30 000 000 lines of code;
- 30 000 (known) bugs!



Bugs

– Software bugs
  - whether anticipated (Y2K bug)
  - or unforeseen (failure of the 5.01 flight of Ariane V launcher)

  are quite frequent;

– Bugs can be very difficult to discover in huge software;

– Bugs can have catastrophic consequences either very costly or inadmissible (embedded software in transportation systems);

The estimated cost of an overflow

– **500 000 000 $**;
– Including indirect costs (delays, lost markets, etc):

2 000 000 000 $;

– The financial results of Arianespace were **negative** in 2000, for the first time since 20 years.

Who cares?

- No one is legally responsible for bugs:

  This software is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

- So, no one cares about software verification
- And even more, one can even make money out of bugs (customers buy the next version to get around bugs in software)

Why no one cares?

- Software designers don't care because there is no risk in writing bugged software
- The law/judges can never enforce more than what is offered by the state of the art
- Automated software verification by formal methods is undecidable whence thought to be impossible
- Whence the state of the art is that no one will ever be able to eliminate all bugs at a reasonable price
- And so no one ever bear any responsability

Current research results

– Research is presently changing the state of the art (e.g. ASTRÉE)

– We can check for the absence of large categories of bugs (may be not all of them but a significant portion of them)

– The verification can be made automatically by mechanical tools

– Some bugs can be found completely automatically, without any human intervention

The next step (5/10 years)

- If these tools are successful, their use can be enforced by quality norms
- Professional have to conform to such norms (otherwise they are not credible)
- Because of complete tool automaticity, no one can be discharged from the duty of applying such state of the art tools
- Third parties of confidence can check software a posteriori to trace back bugs and prove responsabilities

A foreseeable future (10/15 years)

- The real take-off of software verification must be enforced
- Development costs arguments have shown to be ineffective
- Norms/laws might be much more convincing
- This requires effectiveness and complete automation (to avoid acquittal based on human capacity limitations arguments)

Why will “partial software verification” ultimately succeed?

– The state of the art will change toward complete automation, at least for common categories of bugs

– So responsabilities can be established (at least for automatically detectable bugs)

– Whence the law will change (by adjusting to the new state of the art)

– To ensure at least partial software verification

– For the benefit of all of us

Program Verification Methods

Testing

- To prove the presence of bugs relative to a specification;
- Some bugs may be missed;
- Nothing can be concluded on correctness when no bug is found;
- E.g.: debugging, simulation, code review, bounded model checking.

Verification

- To prove the absence of bugs relative to a specification;
- No bug is ever missed [2];
- Inconclusive situations may exist (undecidability) $\rightarrow$ bug or false alarm
- Correctness follows when no bug is found;
- E.g.: deductive methods, static analysis.

[2] ralative to the specification which is checked.

An historical perspective on formal software verification

The origins of program proving

– The idea of proving the correctness of a program in a mathematical sense dates back to the early days of computer science with John von Neumann [1] and Alan Turing [2].

Reference

[1] J. von Neumann. "Planning and Coding of Problems for an Electronic Computing Instrument", U.S. Army and Institute for Advanced Study report, 1946. In *John von Neumann, Collected Works*, Volume V, Pergamon Press, Oxford, 1961, pp. 34-235.

[2] A. M. Turing, " Checking a Large Routine". In *Report of a Conference on High Speed Automatic Calculating Machines*, Univ. Math. Lab., Cambridge, pp 67-69 (1949).

John Von Neumann

Alan Turing

The pioneers

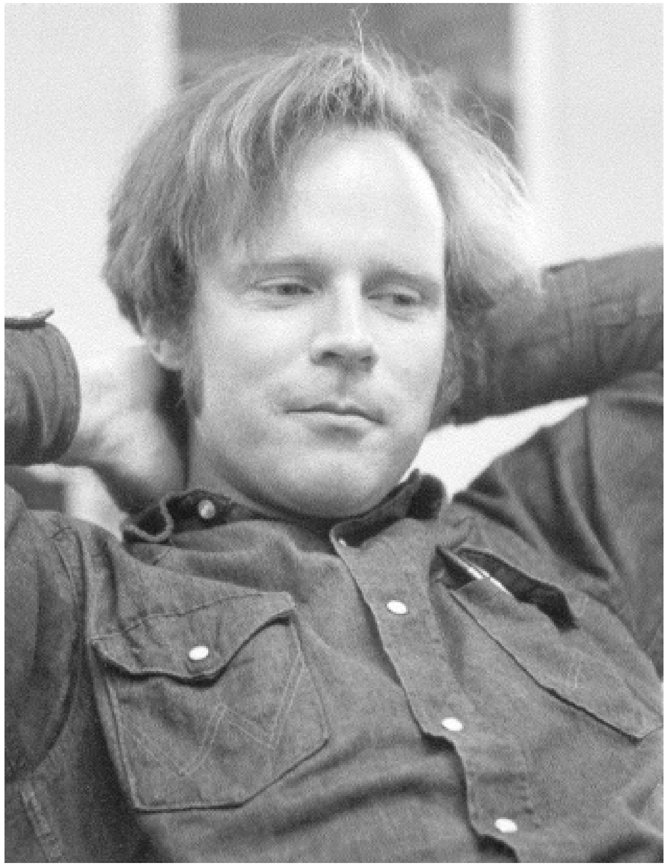
– R. Floyd [3] and P. Naur [4] introduced the "partial correctness" specification together with the "invariance proof method";

– R. Floyd [3] also introduced the "variant proof method" to prove "program termination";

Reference

[3] Robert W. Floyd. "Assigning meanings to programs". In *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, vol. 19, pp. 19–31, 1967.

[4] Peter Naur. "Proof of Algorithms by General Snapshots", BIT 6 (1966), pp. 310-316.



Robert Floyd

Peter Naur

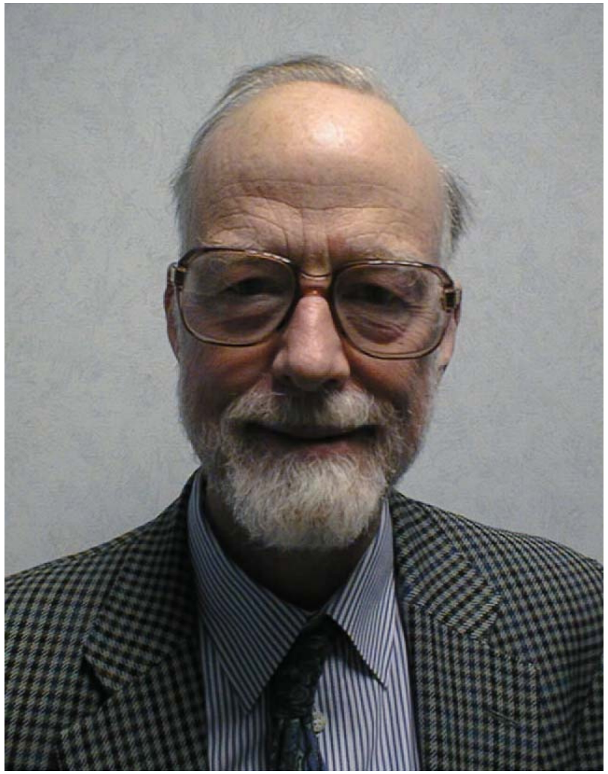
The pioneers (Cont'd)

- C.A.R. Hoare formalized the Floyd/Naur partial correctness proof method in a logic (so-called "Hoare logic") using an Hilbert style inference system;
- Z. Manna and A. Pnueli extended the logic to "total correctness" (i.e. partial correctness + termination).

Reference

[5] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming. Commun. ACM 12(10): 576-580 (1969)

[6] Zohar Manna, Amir Pnueli. "Axiomatic Approach to Total Correctness of Programs". Acta Inf. 3: 243-263 (1974)



C.A.R. Hoare

Zohar Manna

Amir Pnueli

Assertions

– An *assertion* is a statement (logical predicate) about the values of the program variables (i.e., the memory state [3]), which may or may not be valid at some point during the program computation;

– A *precondition* is an assertion at program entry;

– A *postcondition* is an assertion at program exit;

[3] This may also include auxiliary variables to denote initial/intermediate values of program variables.

Partial correctness

- *Partial correctness* states that if a given precondition P holds on entry of a program C and program execution terminates, then a given postcondition Q holds, if and when execution of C terminates;
- *Hoare triple* notation [5]: $\{P\}C\{Q\}$.

Partial correctness (example)

- Tautologies: $\{P\}C\{\text{true}\}$
  $\{\text{false}\}C\{Q\}$
- Nontermination: $\{P\}C\{\text{false}\}$
  $\{P\}C\{Q\}$ if $\{P\}C\{\text{false}\}$

The Euclidian integer division example [3]

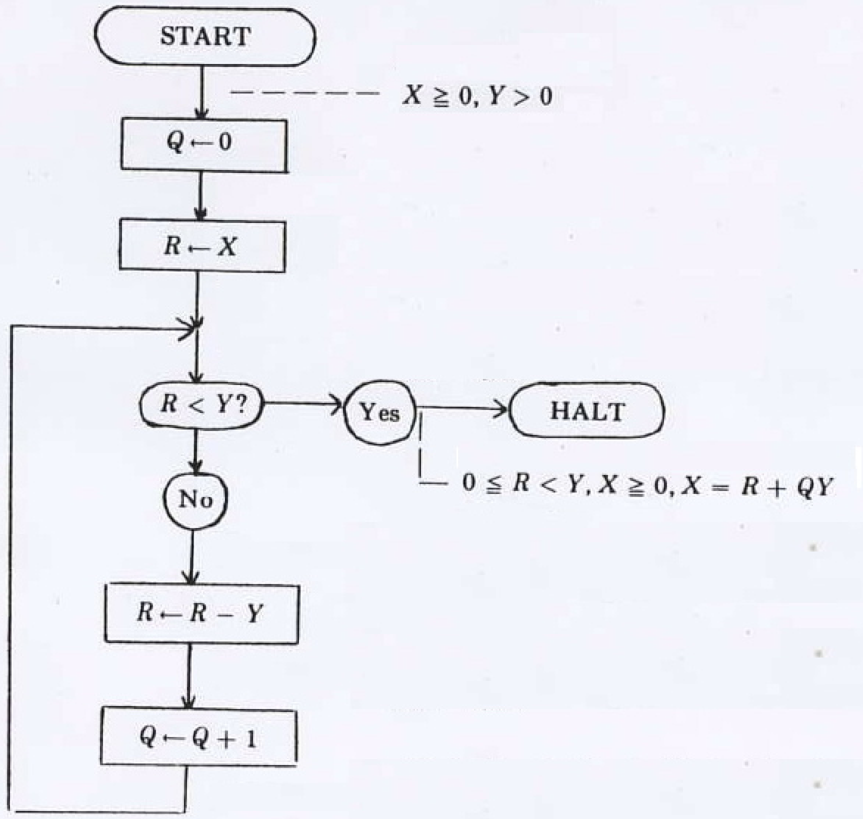


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geqq 0$, $Y > 0$

$$\{X \geq 0 \wedge Y > 0\}$$
$$C$$
$$\{0 \leq R < Y \wedge X \geq 0$$
$$\wedge X = R + QY\}$$

Invariant

- An *invariant* at a given program point is an assertion which holds during execution whenever control reaches that point

The Euclidian integer division example [3]

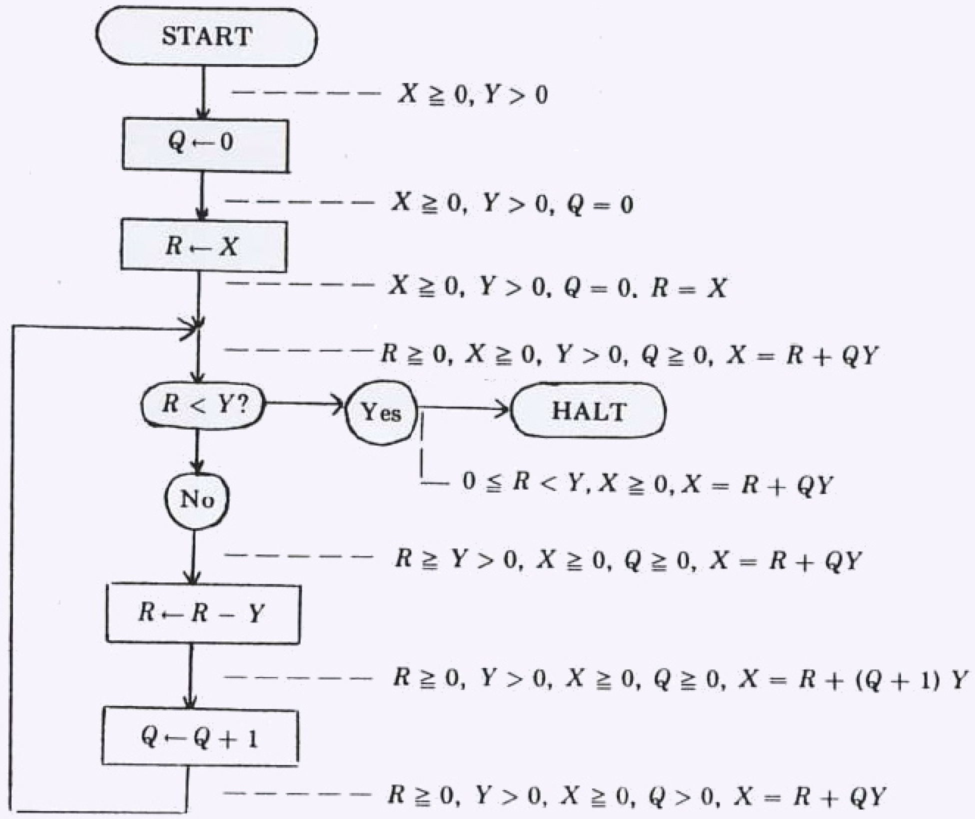


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geqq 0, Y > 0$

Floyd/Naur invariance proof method

To prove that assertions attached to program points are invariant:

- Basic verification condition: Prove the assertion at program entry holds (e.g. follows from a precondition hypothesis);
- Inductive verification condition: Prove that if an assertions holds at some program point and a program step is executed then the assertion does hold at next program point.

Soundness of Floyd/Naur invariance proof method

By induction on the number of program steps, all assertions are invariants [4].

[4] Aslo called inductive invariants

Assignment verification condition

$$\{P(X, Y, \ldots)\}$$

```
X := E(X,Y,...)
```

$$\{Q(X, Y, \ldots)\}$$

- $$\forall X, Y, \ldots : (\exists X' : P(X', Y, \ldots) \wedge X = E(X', Y, \ldots)) \Longrightarrow Q(X, Y, \ldots)$$ R. Floyd
- $$\forall X, Y, \ldots : P(X, Y, \ldots) \Longrightarrow Q(X, Y, \ldots)[X := E]$$ [5] C.A.R. Hoare

---

[5] $B[x := A]$ is the substitution of A for x in B.

Assignment verification condition (example)

$$\{X \geq 0\}$$

```
X := X + 1
```

$$\{X > 0\}$$

- $\forall X : (\exists X' : X' \geq 0 \wedge X = X' + 1)$
  $\Longrightarrow$
  $X > 0$ R. Floyd

- $\forall X : X \geq 0$
  $\Longrightarrow$
  $(X + 1) > 0$ C.A.R. Hoare

Conditional verification condition

$\{P_1(X, Y, \ldots)\}$
`if` $B(X, Y, \ldots)$ `then`
  $\{P_2(X, Y, \ldots)\}$
  $\ldots$
  $\{P_3(X, Y, \ldots)\}$
`else`
  $\{P_4(X, Y, \ldots)\}$
  $\ldots$
  $\{P_5(X, Y, \ldots)\}$
`fi`
$\{P_6(X, Y, \ldots)\}$

- $P_1(X, Y, \ldots) \wedge B(X, Y, \ldots) \Longrightarrow P_2(X, Y, \ldots)$
- $P_1(X, Y, \ldots) \wedge \neg B(X, Y, \ldots) \Longrightarrow P_4(X, Y, \ldots)$
- $P_3(X, Y, \ldots) \vee P_5(X, Y, \ldots) \Longrightarrow P_6(X, Y, \ldots)$

Conditional verification condition (example)

$\{X = x_0\}$
`if` $X \geq 0$ `then`
  $\{X = x_0 \geq 0\}$
  `skip`
  $\{X = x_0 \geq 0\}$
`else`
  $\{X = x_0 < 0\}$
  `X := -X`
  $\{X = -x_0 > 0\}$
`fi`
$\{X = |x_0|\}$

- $X = x_0 \wedge X \geq 0 \implies X = x_0 \geq 0$
- $X = x_0 \wedge \neg X \geq 0 \implies X = x_0 < 0$
- $X = x_0 \geq 0 \vee X = -x_0 > 0 \implies X = |x_0|$ [6]

[6] $|a|$ is the absolute value of a.

While loop verification condition

$\{P_1(X, Y, \ldots)\}$
`while` $B(X, Y, \ldots)$ `do`
  $\{P_2(X, Y, \ldots)\}$
  $\ldots$
  $\{P_3(X, Y, \ldots)\}$
`od`
$\{P_4(X, Y, \ldots)\}$

- $P_1(X, Y, \ldots) \wedge B(X, Y, \ldots) \Longrightarrow P_2(X, Y, \ldots)$
- $P_1(X, Y, \ldots) \wedge \neg B(X, Y, \ldots) \Longrightarrow P_4(X, Y, \ldots)$
- $P_3(X, Y, \ldots) \wedge B(X, Y, \ldots) \Longrightarrow P_2(X, Y, \ldots)$
- $P_3(X, Y, \ldots) \wedge \neg B(X, Y, \ldots) \Longrightarrow P_4(X, Y, \ldots)$

While loop verification condition (example)

$\{X \geq 0\}$
`while` $X \neq 0$ `do`
  $\{X > 0\}$
  `X := X - 1`
  $\{X \geq 0\}$
`od`
$\{X = 0\}$

- $X \geq 0 \wedge X \neq 0 \implies X > 0$
- $X \geq 0 \wedge \neg X \neq 0 \implies X = 0$
- $X \geq 0 \wedge X \neq 0 \implies X > 0$
- $X \geq 0 \wedge \neg X \neq 0 \implies X = 0$

Floyd/Naur partial correctness proof method

- Let be given a precondition P and a postcondition Q;
- Find assertions A_i attached to all program points i;
- Assuming precondition P, prove all assertions A_i to be invariants (using the assignment/conditional and loop verification conditions);
- Prove the invariant on exit implies the postcondition Q.

The Euclidian integer division example

$\{x \geq 0 \wedge y \geq 0\}$ initial condition

$\mathtt{a}{:=}\ 0;\ \mathtt{b}{:=}\mathtt{x}$

$\{b = x \geq 0 \wedge y \geq 0 \wedge a.y + b = x\}$

`while b≥y do`

$\{x \geq 0 \wedge b \geq y \geq 0 \wedge a.y + b = x\}$

$\{x \geq 0 \wedge b \geq y \geq 0 \wedge (a+1).y + (b-y) = x\}$

$\mathtt{b}{:=}\mathtt{b} - \mathtt{y};\ \ \mathtt{a}{:=}\mathtt{a}+1$

$\{x \geq 0 \wedge b \geq 0 \wedge y \geq 0 \wedge a.y + b = x\}$

`od`

$\{a.y + b = x \wedge 0 \leq b < y\}$ partial correctness

Hoare logic

- $\{P[\mathrm{x} := \mathrm{e}]\}\ \mathtt{x{:=}e}\ \{P\}$ assignment axiom (1)

- $$\frac{\{P\}C_1\{R\}, \quad \{R\}C_2\{Q\}}{\{P\}C_1; C_2\{Q\}}$$ composition rule (2)

- $$\frac{\{P \wedge b\}C_1\{Q\}, \quad \{P \wedge \neg b\}C_2\{Q\}}{\{P\}\ \mathtt{if}\ b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi}\ \{Q\}}$$ if-the-else rule (3)

- $$\frac{\{P \wedge b\}C\{P\}}{\{P\}\ \mathtt{while}\ b\ \mathtt{do}\ C\ \mathtt{od}\ \{P \wedge \neg b\}}$$ while rule (4)

- $$\frac{(P \Longrightarrow P'), \quad \{P'\}C\{Q'\}, \quad (Q' \Longrightarrow Q)}{\{P\}C\{Q\}}$$ consequence rule (5)

Formal Partial Correctness Proof of Integer Division

We let $\mathtt{p} \stackrel{\text{def}}{=} \mathtt{while\ b{\geq}y\ do\ b{:=}b - y;\ a{:=}a + 1\ od}$

(a) $\{0.y + x = x \wedge x \geq 0\}\ \mathtt{a{:=}\ 0}\{a.y + x = x \wedge x \geq 0\}$

by the assignment axiom (1)

(b) $\{a.y + x = x \wedge x \geq 0\}\ \mathtt{b{:=}x}\ \{a.y + b = x \wedge b \geq 0\}$

by the assignment axiom (1)

(c) $\{0.y + x = x \wedge x \geq 0\}\ \mathtt{a{:=}\ 0;\ b{:=}x}\ \{a.y + b = x \wedge b \geq 0\}$

by (a), (b) and the composition rule (2)

(d) $(x \geq 0 \wedge y \geq 0) \Longrightarrow (0.y + x = x \wedge x \geq 0)$

by first-order logic

(e) $\{x \geq 0 \wedge y \geq 0\}$ `a:=` $0;$ `b:=x` $\{a.y + b = x \wedge b \geq 0\}$

by (d), (c) and the consequence rule (5)

(f) $\{(a + 1).y + b - y = x \wedge b - y \geq 0\}$ `b:=b - y` $\{(a + 1).y + b = x \wedge b \geq 0\}$

by the assignment axiom (1)

(g) $\{(a+1).y+b = x\wedge b \geq 0\}$ `a:=a +1` $\{a.y+b = x\wedge b \geq 0\}$

by the assignment axiom (1)

(h) $\{(a + 1).y + b - y = x \wedge b - y \geq 0\}$ `b:=b - y; a:=a +1` $\{a.y + b = x \wedge b \geq 0\}$

by (f), (g) and the composition rule (2)

(i) $(a.y+b=x \wedge b \geq 0 \wedge b \geq y) \Longrightarrow ((a+1).y+b-y = x \wedge b-y \geq 0)$

by first-order logic

(j) $\{a.y+b=x \wedge b \geq 0 \wedge b \geq y\}$ `b:=b - y; a:=a +1`$\{a.y+b = x \wedge b \geq 0\}$

by (h), (i) and the consequence rule (5)

(k) $\{a.y+b=x \wedge b \geq 0\}$ `p` $\{a.y+b=x \wedge b \geq 0 \wedge \neg(b \geq y)\}$

by (j) and the while rule (4)

(ℓ) $\{x \geq 0 \wedge y \geq 0\}$ `a:= 0; b:=x; p` $\{a.y+b = x \wedge b \geq 0 \wedge \neg(b \geq y)\}$

by (e), (k) and the composition rule (2)

Q.E.D.

Soundness and Completeness

- Soundness: no erroneous fact can be derived by Hoare logic;
- Completeness: all true facts can be derived by Hoare logic;
- If the first-order logic includes arithmetic then there exists no complete axiomatization of $\Longrightarrow$ in the consequence rule (5) (Gödel theorem)

Relative Completeness

– Relative completeness [7]: all true facts can be derived by Hoare logic provided:
  - the first-order assertion language is rich enough to express loop invariants;
  - all first-order theorems needed in the consequence rule are given (e.g. by an oracle).

Reference

[7] Stephen A. Cook: "Soundness and Completeness of an Axiom System for Program Verification". SIAM J. Comput. 7(1): 70-90 (1978)

Termination

- Termination: no program execution can run for ever;
- Bounded termination: the program terminates in a time bounded by some function of the input;
- Example of unbounded termination:

```
X := ?;              ← random number generator
while X > 0 do
  Y := ?;
  while Y > 0 do
    Y := Y - 1
  od;
  X := X - 1
od
```

Well-founded relation

- A relation r is well-founded on a set S if and only if there is no infinite sequence s of elements of S which are r-related:

$$\neg(\exists s \in \mathbb{N} \mapsto S : \forall i \in \mathbb{N} : r(s_i, s_{i+1}))$$

- Examples: $>$ on $\mathbb{N}$ (the naturals, $n > n-1 > \ldots > 0$)
- Counter-examples: $>$ on $\mathbb{Z}$ (the integers, $0 > -1 > -2 > \ldots$), $>$ on $\mathbb{Q}$ (the rationals, $1 > \frac{1}{2} > \frac{1}{3} > \frac{1}{4} \ldots$)

Floyd termination proof method

– Exhibit a so-called *ranking function* from the values of the program variables to a set S and a well-founded relation r on S;

– Show that the ranking function takes r-related values on each program step.

Soundness: non-termination would be in contradiction with well-foundedness

Completeness: for a terminating program, the number of remaining steps [7] strictly decreases.

---

[7] This is meaningfull for bounded termination only, otherwise one has to resort to ordinals.

The Euclidian integer division example [3]

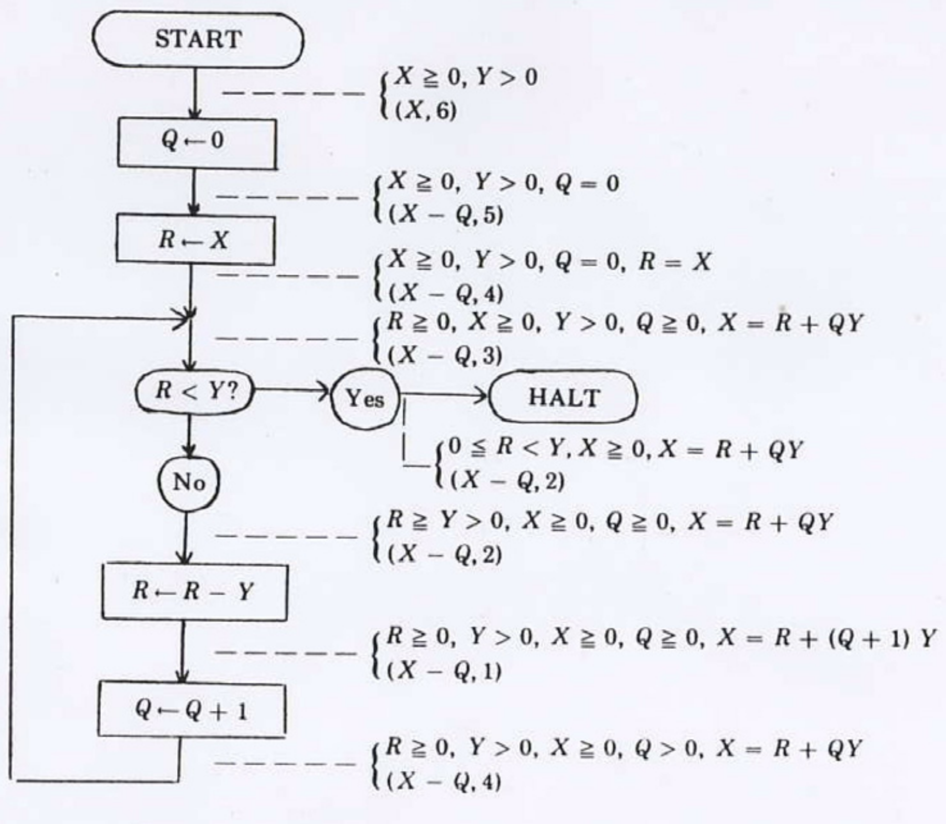


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geqq 0$, $Y > 0$

Suppose, for example, that an interpretation of a flowchart is supplemented by associating with each edge in the flowchart an expression for a function, which we shall call a W-function, of the free variables of the interpretation, taking its values in a well-ordered set W. If we can show that after each execution of a command the current value of the W-function associated with the exit is less than the prior value of the W-function associated with the entrance, the value of the function must steadily decrease. Because no infinite decreasing sequence is possible in a well-ordered set, the program must sooner or later terminate. Thus, we prove termination, a global property of a flowchart, by local arguments, just as we prove the correctness of an algorithm.

Termination of structured programs

Its sufficient to prove termination of loops[8]. Example:

$\{x \geq 0 \wedge y > 0\}$ initial condition

a:= 0; b:=x

$\{b = x \geq 0 \wedge y > 0 \wedge a.y + b = x\}$

while b≥y do

$\{x \geq 0 \wedge b \geq y > 0 \wedge a.y + b = x\}$

b:=b − y; a:=a +1

$\{x \geq 0 \wedge b \geq 0 \wedge y > 0 \wedge a.y + b = x\}$

od

$\{a.y + b = x \wedge 0 \leq b < y\}$ total correctness

---

8 and recursive functions.

Example: Integer Division by Euclid's Algorithm

- Assume the initial condition $y > 0$;
- The value b of variable $\mathtt{b}$ within the loop is positive whence belongs to the well-ordering $\langle \mathbb{N}, < \rangle$;
- The value b of variable $\mathtt{b}$ strictly decreases (by $y > 0$) on each loop iteration.

Note:

- Partially but not totally correct when initially $y = 0$.

Total correctness

Total correctness = partial correctness $\wedge$ termination

Ordinals

– An extension of naturals for ranking (1^{st}, 2^{nd}, 3^{rd}, ...) beyond infinity

– The first ordinals are 0, 1, 2, ..., ω [9], $\omega+1$, $\omega+2$, ..., $\omega+\omega=2\omega$, $2\omega+1$, ..., 3ω, $3\omega+1$, ..., $\omega.\omega=\omega^2$, ω^2+1, ..., ω^3, ..., ω^ω, ω^{ω^ω}, ..., ϵ_0 [10] $= \omega^{\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}}\Big\}\ \omega$ times, ...

[9] ω is the first transfinite ordinal.

[10] ϵ_0 is the first ordinal numbers which cannot be constructed from smaller ones by <u>finite</u> additions, multiplications, and exponentiations.

The Manna/Pnueli logic

– $[P]C[Q]$ Hoare total correctness triple

– Interpretation:

If the assertion P[11] holds before the execution of command C then execution of C terminates and assertion Q holds upon termination

– $$\frac{(P(\alpha) \wedge \alpha > 0) \Rightarrow b, [P(\alpha) \wedge \alpha > 0]C[\exists \beta < \alpha : P(\beta)], P(0) \Rightarrow \neg b}{[\exists \alpha : P(\alpha)]\ \texttt{while}\ b\ \texttt{do}\ C\ \texttt{od}\ [P(0)]}$$

while rule (6)[12]

[11] on the values of the program variables and auxiliary mathematical variables

[12] α, β, ... are ordinals.

Formal Total Correctness Proof of Integer Division

- $R \stackrel{\text{def}}{=} a.y + b = x \wedge b \geq 0$
- $P(n) \stackrel{\text{def}}{=} R \wedge n.y \leq b < (n+1).y$
- We have:
  - $(P(n) \wedge n > 0) \Longrightarrow (b \geq y)$
  - $[P(n+1)]$ b:=b − y; a:=a +1$[P(n)]$
  - $P(0) \Longrightarrow \neg(b \geq y)$
  - $R \wedge y > 0 \Longrightarrow \exists n : P(n)$

  so that by the while rule (6) and the consequence rule (5), we conclude:

  $$[a.y + b = x \wedge b \geq 0 \wedge y > 0] \text{ p } [a.y + b = x \wedge b \geq 0 \wedge \neg(b \geq y)]$$

Predicate transformers

Edsger W. Dijkstra introduced predicate transformers:

– $\mathrm{wlp}[\![C]\!]Q$ is the weakest liberal[13] precondition:

  - $\{\mathrm{wlp}[\![C]\!]Q\}C\{Q\}$
  - $\{P\}C\{Q\} \Longrightarrow (P \Rightarrow \mathrm{wlp}[\![C]\!]Q)$

– $\mathrm{wp}[\![C]\!]Q$ is the weakest precondition:

  - $[\mathrm{wp}[\![C]\!]Q]C[Q]$
  - $[P]C[Q] \Longrightarrow (P \Rightarrow \mathrm{wp}[\![C]\!]Q)$

Reference

[8] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". Commun. ACM 18(8): 453-457 (1975)

13 "liberal" means nontermination is possible i.e. partial correctness.

Edsger W. Dijkstra

Predicate transformer calculus

– `skip` is the command that leaves the state unchanged

$\mathrm{wlp}[\![\mathtt{skip}]\!]\,P = P$

$\mathrm{wp}[\![\mathtt{skip}]\!]\,P = P$

– `abort` is the command that never terminates

$\mathrm{wlp}[\![\mathtt{abort}]\!]\,P = \mathrm{tt}$

$\mathrm{wp}[\![\mathtt{abort}]\!]\,P = \mathrm{ff}$

– ; is the sequential composition of commands

$\mathrm{wlp}[\![\mathsf{C}_1 \,;\, \mathsf{C}_2]\!]\,P = \mathrm{wlp}[\![\mathsf{C}_1]\!](\mathrm{wlp}[\![\mathsf{C}_2]\!]\,P)$

$\mathrm{wp}[\![\mathsf{C}_1 \,;\, \mathsf{C}_2]\!]\,P = \mathrm{wp}[\![\mathsf{C}_1]\!](\mathrm{wp}[\![\mathsf{C}_2]\!]\,P)$

Nondeterministic Choice

- $\|$ is the nondeterministic choice of commands

  $\mathrm{wlp}[\![\mathsf{C}_1 \;\|\; \mathsf{C}_2]\!]\,P = \mathrm{wlp}[\![\mathsf{C}_1]\!]\,P \wedge \mathrm{wlp}[\![\mathsf{C}_2]\!]\,P$

  $\mathrm{wp}[\![\mathsf{C}_1 \;\|\; \mathsf{C}_2]\!]\,P = \mathrm{wp}[\![\mathsf{C}_1]\!]\,P \wedge \mathrm{wp}[\![\mathsf{C}_2]\!]\,P$

- Example:

  $\mathrm{wp}[\![\mathbf{skip} \;\|\; \mathbf{abort}]\!]\,P = \mathrm{wp}[\![\mathbf{skip}]\!]\,P \wedge \mathrm{wp}[\![\mathbf{abort}]\!]\,P = P \wedge \mathit{ff} = \mathit{ff}$

  $\mathrm{wlp}[\![\mathbf{skip} \;\|\; \mathbf{abort}]\!]\,P = \mathrm{wlp}[\![\mathbf{skip}]\!]\,P \wedge \mathrm{wlp}[\![\mathbf{abort}]\!]\,P = P \wedge \mathit{tt} = P$

Guards

– If b is a *guard* (precondition), then $?\mathsf{b}$ is defined by [14]:

$\mathrm{wlp}[\![?\mathsf{b}]\!]\, P = \neg\mathsf{b} \vee P$

$\mathrm{wp}[\![?\mathsf{b}]\!]\, P = \neg\mathsf{b} \vee P$

– If b is a *guard* (precondition), then $!\mathsf{b}$ skips if b holds and does not terminate if $\neg\mathsf{b}$ holds;

$\mathrm{wlp}[\![!\mathsf{b}]\!]\, P \stackrel{\text{def}}{=} \neg\mathsf{b} \vee P$

$\mathrm{wp}[\![!\mathsf{b}]\!]\, P \stackrel{\text{def}}{=} \mathsf{b} \wedge P$

14 $\mathrm{wp}[\![?\mathsf{ff}]\!]\,\mathsf{ff} = \mathsf{tt}$ so the $?\mathsf{ff}$ command is not implementable since it should miraculously terminate in a state where ff holds!

Conditional

- $\texttt{if}\ \mathsf{b}\ \texttt{then}\ \mathsf{C}_1\ \texttt{else}\ \mathsf{C}_2 \stackrel{\mathrm{def}}{=} (?\mathsf{b};\mathsf{C}_1) \parallel (?\neg\mathsf{b};\mathsf{C}_2)$
- Below, $\mathtt{w}[\![\mathsf{C}]\!]\,P$ is either $\mathtt{wp}[\![\mathsf{C}]\!]\,P$ or $\mathtt{wlp}[\![\mathsf{C}]\!]\,P$

$$
\begin{aligned}
&\mathtt{w}[\![\texttt{if}\ \mathsf{b}\ \texttt{then}\ \mathsf{C}_1\ \texttt{else}\ \mathsf{C}_2]\!]\,P \\
&= \mathtt{w}[\![(?\mathsf{b};\mathsf{C}_1) \parallel (?\neg\mathsf{b};\mathsf{C}_2)]\!]\,P = \mathtt{w}[\![?\mathsf{b};\mathsf{C}_1]\!]\,P \wedge \mathtt{w}[\![?\neg\mathsf{b};\mathsf{C}_2]\!]\,P \\
&= (\mathtt{w}[\![?\mathsf{b}]\!](\mathtt{w}[\![\mathsf{C}_1]\!]\,P)) \wedge (\mathtt{w}[\![?\neg\mathsf{b}]\!](\mathtt{w}[\![\mathsf{C}_2]\!]\,P)) \\
&= (\neg\mathsf{b} \vee \mathtt{w}[\![\mathsf{C}_1]\!]\,P) \wedge (\neg\neg\mathsf{b} \vee \mathtt{w}[\![\mathsf{C}_2]\!]\,P) \\
&= (\mathsf{b} \Longrightarrow \mathtt{w}[\![\mathsf{C}_1]\!]\,P) \wedge (\neg\mathsf{b} \Longrightarrow \mathtt{w}[\![\mathsf{C}_2]\!]\,P) \\
&= (\mathsf{b} \wedge \mathtt{w}[\![\mathsf{C}_1]\!]\,P) \vee (\neg\mathsf{b} \wedge \mathtt{w}[\![\mathsf{C}_2]\!]\,P)
\end{aligned}
$$

Conditional

– $\mathtt{if}\ \mathsf{b}_0 \to \mathsf{C}_0 \mathbin{[\!]} \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{fi} \stackrel{\text{def}}{=} \,!(\mathsf{b}_0 \vee \mathsf{b}_1); (?\mathsf{b}_0; \mathsf{C}_0 \mathbin{[\!]} ?\mathsf{b}_1; \mathsf{C}_1)$

$$\begin{aligned} &\mathbf{wp}[\![\mathtt{if}\ \mathsf{b}_0 \to \mathsf{C}_0 \mathbin{[\!]} \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{fi}]\!]\, P \\ = {}& (\exists i \in [0,1] : \mathsf{b}_i) \wedge (\forall i \in [0,1] : \mathsf{b}_i \implies \mathbf{wp}[\![\mathsf{C}_i]\!]\, P) \end{aligned}$$

"The first term '$\exists i \in [0,1] : \mathsf{b}_i$' requires that the alternative construct as such will not lead to abortion on account of all guards false; the second term requires that each guarded list eligible for execution will lead to an acceptable final state" [8].

Iteration

– The execution of Dijkstra's repetitive construct:

$$\texttt{do}\ \mathsf{b}_0 \rightarrow \mathsf{C}_0 \mathbin{[\!]} \mathsf{b}_1 \rightarrow \mathsf{C}_1\ \texttt{od}$$

immediately terminates if both guards b_0 and b_1 are false otherwise it consists in executing one of the alternatives $\mathsf{C}_i, i \in [1, 2]$ which guard b_i is true before repeting the execution of the loop.

- $\mathtt{wp}[\![\mathtt{do}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{od}]\!] =$ [15]

$$\lambda Q \cdot \mathsf{lfp}^{\Longrightarrow}\ F^{\mathrm{wp}}[\![\mathtt{do}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{od}]\!](Q)$$

- $F^{\mathrm{wp}}[\![\mathtt{do}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{od}]\!](Q) =$

$$\lambda P \cdot (Q \wedge \forall i \in [0,1] : \neg \mathsf{b}_i) \vee \mathtt{wp}[\![\mathtt{if}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{fi}]\!]\, P$$

- $\mathtt{wlp}[\![\mathtt{do}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{od}]\!] =$

$$\lambda Q \cdot \mathsf{gfp}^{\Longrightarrow}\ F^{\mathrm{wlp}}[\![\mathtt{do}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{od}]\!](Q)$$

- $F^{\mathrm{wlp}}[\![\mathtt{do}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{od}]\!](Q) =$

$$\lambda P \cdot (Q \wedge \forall i \in [0,1] : \neg \mathsf{b}_i) \vee \mathtt{wlp}[\![\mathtt{if}\ \mathsf{b}_0 \to \mathsf{C}_0 \parallel \mathsf{b}_1 \to \mathsf{C}_1\ \mathtt{fi}]\!]\, P$$

---

15 $\mathsf{lfp}^{\sqsubseteq} f$ is the $\sqsubseteq$-least fixpoint of f, if any. Dually, $\mathsf{gfp}^{\sqsubseteq}_{f}$ is the $\sqsubseteq$-greatest fixpoint of f, if any.

Automatic
Program Verification Methods

First attempts towards automation

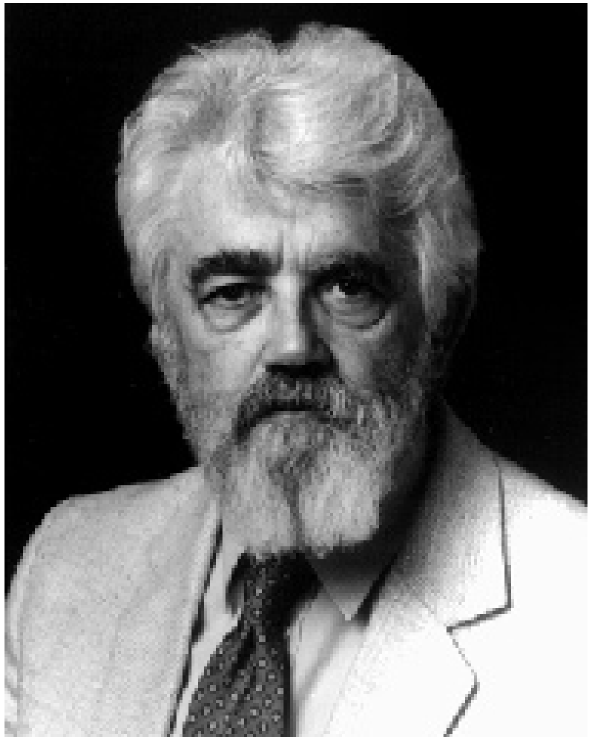
- James C. King, a student of Robert Floyd, produced the first automated proof system for numerical programs, in 1969 [9].
- The use of automated theorem proving in the verification of symbolic programs (à la LISP [10]) was pioneered, a.o., by Robert S. Boyer and J. Strother Moore [11].

Reference

[9] King, J. C., "A Program Verifier", Ph.D. Thesis, Carnegle-Mellon University (1969).

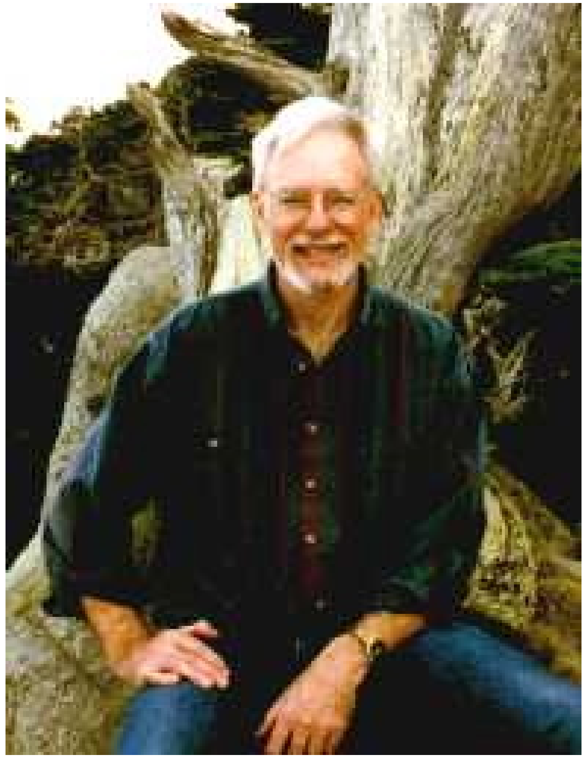
[10] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine (Part I)". Communications of the ACM (CACM), April 1960.

[11] Robert S. Boyer and J. Strother Moore, "Proving Theorems about LISP Functions". Journal of the ACM (JACM), Volume 22, Issue 1 (January 1975) pp. 129–144.



John McCarthy

Robert S. Boyer



J. Strother Moore

Present day theorem-proving based followers

Automatic deductive methods (based on theorem provers or checkers with user-provided assertions and guidance):

- ACL2
- B
- COQ
- ESC/Java & ESC/Java2
- PVS
- Why

Very useful for small programs, huge difficulties to scale up.

A Grand Challenge

A grand challenge in computer science

"The construction and application of a verifying compiler that guarantees correctness of a program before running it" [12].

Reference

[12] Tony Hoare. "The verifying compiler: A grand challenge for computing research", Journal of the ACM (JACM), Volume 50, Issue 1 (January 2003), pp. 63–69.