

IA159 Formal Verification Methods

Deductive Software Verification

Jan Strejček

Department of Computer Science
Faculty of Informatics
Masaryk University

Focus

- first formal approach to verification of algorithms and computer programs
- partial and total correctness
- formal system for verification of flowcharts by Floyd (1967)
- axiomatic program verification by Hoare (1969)

Source

- Chapter 7 of
D. A. Peled: Software Reliability Methods, Springer, 2001.

Assumptions and basic terminology

- for simplicity we consider only **deterministic** programs where the initial values of a program are stored in **input variables** x_0, x_1, \dots and these variables do not change their values during any execution of the program
- a **state** of a program is an assignment to the program variables
- given a program P and its states a, b , by $P(a, b)$ we denote the fact that the execution of P starting from the state a terminates with the state b
- $a \models \varphi$ denotes that the state a satisfies the formula φ

A **specification** (or a desired property) of program P is given by two first order formulae:

- **initial condition** φ is a formula with all its free variables among input variables of P
- **final assertion** ψ

Two notions of correctness

The program P is

partially correct with respect to φ and ψ , written $\{\varphi\}P\{\psi\}$, iff for all states a, b it holds

$$P(a, b) \wedge a \models \varphi \implies b \models \psi.$$

If the program starts with a state satisfying φ and then terminates, then the terminal state satisfies ψ .

totally correct with respect to φ and ψ , written $\langle\varphi\rangle P\langle\psi\rangle$, iff $\{\varphi\}P\{\psi\}$ and for every state a satisfying φ the program terminates.

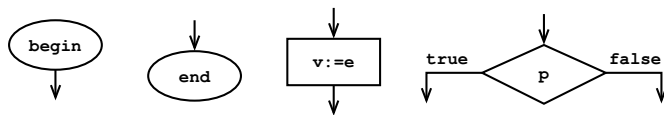
If the program starts with a state satisfying φ , then it terminates and the terminal state satisfies ψ .



by **Robert W Floyd** (1936–2001)

- 1965: associate professor at Carnegie–Mellon University
- 1968: full professor at Stanford University, without Ph.D.
- Floyd–Warshall algorithm: shortest paths in a graph
- Floyd–Steinberg dithering: rendering images
- program verification, parsing, sorting

Flowcharts: four kinds of nodes



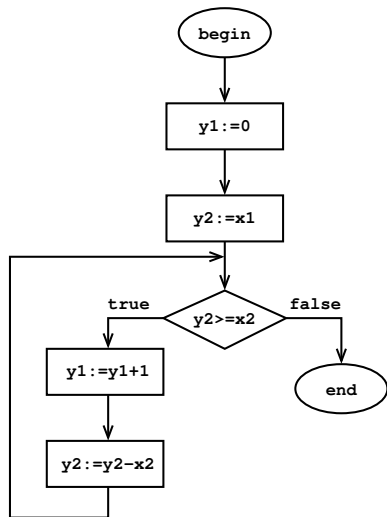
begin one outgoing edge, no incoming edges

end one incoming edge, no outgoing edges

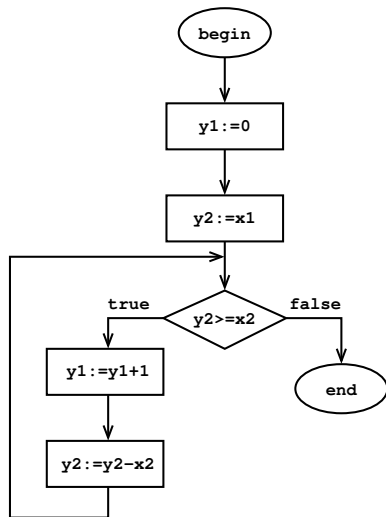
assignment $v := e$, where v is a variable, e is a first order term;
one or more incoming edges, one outgoing edge

decision predicate p is an unquantified first order formula;
one or more incoming edges, two outgoing edges
marked with **true** and **false**

Example: what is this program good for?



Example: what is this program good for?



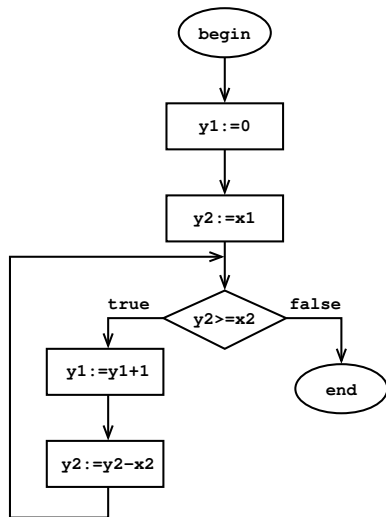
initial condition

$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$

final assertion

$$\psi \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq 0 \wedge y2 < x2$$

Example: what is this program good for?



initial condition

$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$

final assertion

$$\psi \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq 0 \wedge y2 < x2$$

It computes an integer division.

Proving partial correctness

Proving partial correctness

A **location** of a flowchart program is an edge connecting two flowchart nodes.

To verify that a program P is partially correct with respect to an initial condition φ and a final assertion ψ , it is sufficient to perform the following **two steps**.

Proving partial correctness: step 1

Step 1

- to each location of the flowchart we attach a first order formula called **assertion** or **invariants**
- to the location exiting from **begin** we attach φ
- to the location entering **end** we attach ψ

Idea

These assertions should be satisfied by every state reachable in the corresponding location by an execution starting in a state satisfying φ .

Proving partial correctness: step 2

Given an assignment or decision node c , every assumption on

- an incoming edge is called **precondition**, written $pre(c)$
- an outgoing edge is called **postcondition**, written $post(c)$

Idea of step 2

We have to prove that whenever the control of the program is just before a node c with a state satisfying $pre(c)$ and execution of c moves the control to the location annotated with $post(c)$, then the state after the move satisfies $post(c)$.

Step 2

Every triple $pre(c), c, post(c)$ is treated according to its form.

- 1 c is a **decision node** with a predicate p and $post(c)$ is associated to the outgoing edge marked with **true**.

Then we need to prove:

$$pre(c) \wedge p \implies post(c)$$

- 2 c is a **decision node** with a predicate p and $post(c)$ is associated to the outgoing edge marked with **false**.

Then we need to prove:

$$pre(c) \wedge \neg p \implies post(c)$$

Proving partial correctness: step 2

- 3 c is an **assignment** of the form $v := e$, where v is a variable and e an expression.

The states before and after the assignment are different (i.e. $pre(c)$ and $post(c)$ reason about different states). Therefore, we **relativize** the postcondition to assert about the states before the assignment.

Hence, we have to prove

$$pre(c) \implies post(c)[v/e]$$

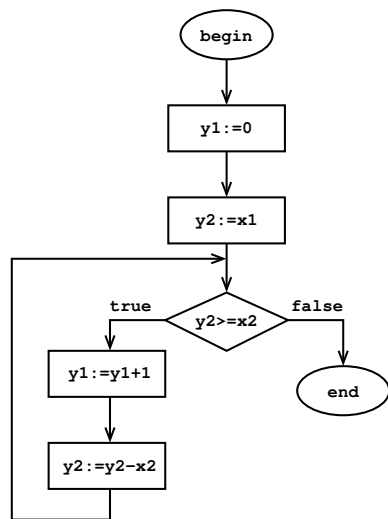
where $post(c)[v/e]$ is the assertion $post(c)$ where all occurrences of v are replaced with e .

Proving partial correctness

- proving the consistency between each precondition and postcondition of all nodes guarantees that $\{\varphi\}P\{\psi\}$
- in fact, it guarantees even a stronger property:

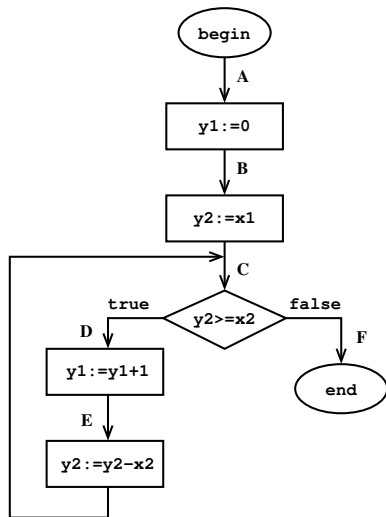
In each execution that starts with a state satisfying the initial condition of the program, when the control of the program is at some location, the assumption attached to that location holds.

Example: partial correctness



$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$
$$\psi \equiv (x1 = y1 * x2 + y2) \wedge$$
$$\wedge y2 \geq 0 \wedge y2 < x2$$

Example: partial correctness



$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$

$$\psi \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq 0 \wedge y2 < x2$$

$$\varphi_A \equiv \varphi$$

$$\varphi_B \equiv x1 \geq 0 \wedge x2 > 0 \wedge y1 = 0$$

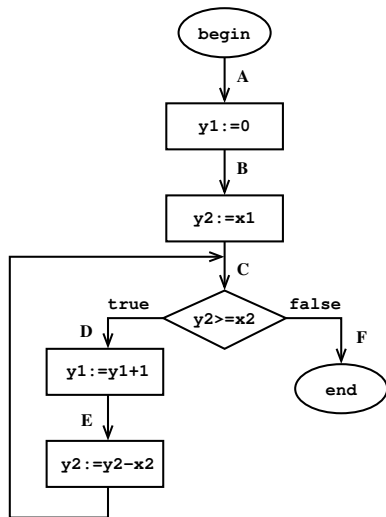
$$\varphi_C \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq 0$$

$$\varphi_D \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq x2$$

$$\varphi_E \equiv (x1 = y1 * x2 + y2 - x2) \wedge \\ \wedge y2 - x2 \geq 0$$

$$\varphi_F \equiv \psi$$

Example: partial correctness



$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$

$$\psi \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq 0 \wedge y2 < x2$$

$$\varphi_A \equiv \varphi$$

$$\varphi_B \equiv x1 \geq 0 \wedge x2 > 0 \wedge y1 = 0$$

$$\varphi_C \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq 0$$

$$\varphi_D \equiv (x1 = y1 * x2 + y2) \wedge \\ \wedge y2 \geq x2$$

$$\varphi_E \equiv (x1 = y1 * x2 + y2 - x2) \wedge \\ \wedge y2 - x2 \geq 0$$

$$\varphi_F \equiv \psi$$

Step 2: check the consistency

- finding assertions for the proof may be a difficult task
- there are some heuristics and tools suggesting invariants
- there cannot be a fully automatic way of finding them (the problem is undecidable)
- in some programming languages, assertions can be inserted into the code as additional runtime checks so that the program will break with a warning message whenever an invariant is violated

Programs with array variables: a problem

Example

- precondition $pre(c) \equiv x[1] = 1 \wedge x[2] = 3$
- assignment $x[x[1]] := 2$
- postcondition $post(c) \equiv x[x[1]] = 2$
- it is easy to prove

$$pre(c) \implies post(c)[x[x[1]]/2]$$

as $post(c)[x[x[1]]/2]$ is in fact $2 = 2$

- but if $pre(c)$ holds and the assignment is performed, then $x[1] = 2$ and $x[x[1]] = 3$ and $post(c)$ does not hold

To handle programs with array variables, the method has to be modified in one point: relativization of postconditions of assignment nodes.

Modification for array variables

- let x be an array variable and e_1, e_2, e_3 terms
- the syntax of terms is extended with a new construct $(x; e_1:e_2)[e_3]$, where $(x; e_1:e_2)$ represents almost the same array as x , only the element with the index e_1 has been set to e_2
- to check the consistency of an assignment $x[e_1] := e_2$ with a precondition $pre(c)$ and postcondition $post(c)$, we have to prove

$$pre(c) \implies post(c)[x/(x; e_1:e_2)]$$

- the added construct does not increase the expressiveness of the logic: a formula ρ containing $(x; e_1:e_2)[e_3]$ can be translated into an equivalent formula

$$\begin{aligned} & (e_1 = e_3 \wedge \rho[(x; e_1:e_2)[e_3]/e_2]) \vee \\ & \vee (\neg(e_1 = e_3) \wedge \rho[(x; e_1:e_2)[e_3]/x[e_3]]) \end{aligned}$$

Proving termination

Proving termination: terminology

- a **partially ordered domain** is a pair (W, \prec) where W is a set and \prec is a strict partial order relation over W (i.e. irreflexive, asymmetric, and transitive)
- $u \succ v$ has the same meaning as $v \prec u$
- we denote $u \succeq v$ when $u \succ v$ or $u = v$
- a **well founded domain** is a partially ordered domain containing no infinite sequence of the form

$$w_0 \succ w_1 \succ w_2 \succ w_3 \succ \dots$$

(i.e. no infinite decreasing sequence)

Proving termination

To prove the termination with respect to the initial condition φ , we need to do the following steps.

- 1 We select a well founded domain (W, \prec) such that W is a subset of the domain of program variables and \prec is expressible using the signature of the program.
- 2 To each location in the flowchart we attach an invariant and an expression. To the location exiting from **begin** we attach φ .
- 3 We show the consistency for each triple $pre(c), c, post(c)$, as in the partial correctness proof.

- 4 We show that whenever an execution starting in a state satisfying φ reaches some location, the value of the expression associated to this location is within W .

Formally, we prove that for each location with the associated invariant ρ and expression e it holds:

$$\rho \implies (e \in W)$$

Note that $e \in W$ is not, in general, a first order logic formula. However, it can often be translated into a first order formula.

Proving termination

- 5 We show that in each execution of the program, when proceeding from a location to its successor location, the value of the associated expressions does not increase.

Formally, for every node c , an incoming edge with the associated invariant $pre(c)$ and expression $e1$, and an outgoing edge with the associated expression $e2$

- if c is a decision node with a predicate p and $e2$ is associated with the **true** edge, then we prove:

$$pre(c) \wedge p \implies e1 \succeq e2$$

- if c is a decision node with a predicate p and $e2$ is associated with the **false** edge, then we prove:

$$pre(c) \wedge \neg p \implies e1 \succeq e2$$

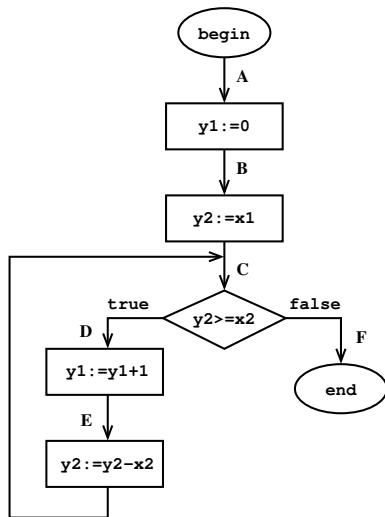
- if c is an assignment $v := e$, then we prove:

$$pre(c) \implies e1 \succeq e2[v/e]$$

- 6 In each execution of the program, during a traversal of a cycle (a loop) in the flowchart there is some point where a decrease occurs in the value of the associated expressions from one location to its successor.

Formally, for each cycle we have to find a node with an incoming and an outgoing edge such that the corresponding implication above holds even if $\underline{\succ}$ is replaced with \succ .

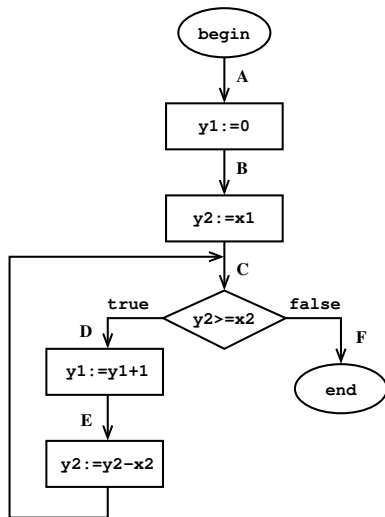
Example: termination



initial condition

$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$

Example: termination



initial condition

$$\varphi \equiv x1 \geq 0 \wedge x2 > 0$$

$$\varphi_A \equiv \varphi$$

$$\varphi_B \equiv x1 \geq 0 \wedge x2 > 0$$

$$\varphi_C \equiv x2 > 0 \wedge y2 \geq 0$$

$$\varphi_D \equiv x2 > 0 \wedge y2 \geq x2$$

$$\varphi_E \equiv x2 > 0 \wedge y2 \geq x2$$

$$\varphi_F \equiv y2 \geq 0$$

$$e_A = x1$$

$$e_B = x1$$

$$e_C = y2$$

$$e_D = y2$$

$$e_E = y2$$

$$e_F = y2$$

- it may be difficult to find the right well founded domain, invariants, and expressions
- termination and partial correctness can be proven simultaneously



by **sir Charles Antony Richard Hoare** (1934)

- studied in Oxford University and Moscow State University
- Quicksort algorithm (1960)
- Hoare logic: program verification
- Communicating Sequential Processes (CSP)
- now in Microsoft Research

- a proof system that includes both logic and pieces of code
- allows to prove different sequential parts of the program separately (and combine the proofs later)
- constructed on top of some first order deduction system

Hoare logic

- contains **Hoare triples** of the form $\{\varphi\}S\{\psi\}$, where φ, ψ are first order formulae and S is (a part of) a program with the syntax:

$$S ::= v := e \mid skip \mid S; S \mid \textit{if } p \textit{ then } S \textit{ else } S \textit{ fi} \mid \\ \textit{while } p \textit{ do } S \textit{ end} \mid \textit{begin } S \textit{ end}$$

where v is a variable, e is a first order expression, and p is an unquantified first order formula

- a Hoare triple $\{\varphi\}S\{\psi\}$ means that if an execution of S starts with a state satisfying φ and S terminates from that state, then a state satisfying ψ is reached
- if S is the entire program, then $\{\varphi\}S\{\psi\}$ claims that S is partially correct with respect to initial condition φ and final assertion ψ

Axioms and proof rules

Assignment axiom

$$\{\varphi[v/e]\} v := e \{\varphi\}$$

Skip axiom

$$\{\varphi\} \mathit{skip} \{\varphi\}$$

Left strengthening rule

$$\frac{\varphi \implies \var' \quad \{\var'\} \mathbf{S}\{\psi\}}{\{\varphi\} \mathbf{S}\{\psi\}}$$

Right weakening rule

$$\frac{\{\varphi\} \mathbf{S}\{\psi'\} \quad \psi' \implies \psi}{\{\varphi\} \mathbf{S}\{\psi\}}$$

Axioms and proof rules

Sequential composition rule

$$\frac{\{\varphi\}S_1\{\eta\} \quad \{\eta\}S_2\{\psi\}}{\{\varphi\}S_1; S_2\{\psi\}}$$

If-then-else rule

$$\frac{\{\varphi \wedge p\}S_1\{\psi\} \quad \{\varphi \wedge \neg p\}S_2\{\psi\}}{\{\varphi\}\text{if } p \text{ then } S_1 \text{ else } S_2 \text{ fi}\{\psi\}}$$

While rule

$$\frac{\{\varphi \wedge p\}S\{\varphi\}}{\{\varphi\}\text{while } p \text{ do } S \text{ end}\{\varphi \wedge \neg p\}}$$

Begin-end rule

$$\frac{\{\varphi\}S\{\psi\}}{\{\varphi\}\text{begin } S \text{ end}\{\psi\}}$$

Assignment axiom + left strengthening rule

$$\frac{\varphi \implies \psi[v/e] \quad \{\psi[v/e]\} v := e \{\psi\} \text{ (axiom)}}{\{\varphi\} v := e \{\psi\}}$$

Sequential composition + right weakening rule

$$\frac{\{\psi\} S_1 \{\eta_1\} \quad \eta_1 \implies \eta_2 \quad \{\eta_2\} S_2 \{\psi\}}{\{\varphi\} S_1; S_2 \{\psi\}}$$

The proof trees are constructed as usual...

Extensions of the Hoare proof system for verifying concurrent programs provide axioms for

- dealing with shared variables
- synchronous and asynchronous communication
- procedure calls

They are usually tailored for a particular programming language, e.g. **Pascal** or **CSP**.

Soundness and completeness

- Hoare's proof system is sound.
- It is not complete due to incompleteness of first order logic with natural numbers and basic arithmetic operations over them (Gödel's incompleteness theorem).
- It is **relatively complete**, i.e. any correct assertion can be proved under the following two (sometimes unrealistic) conditions:
 - Every correct (first order) logic assertion that is needed in the proof is already included as an axiom in the proof system. (Alternatively: there is an **oracle** (e.g. a human) deciding whether such an assertion is correct or not.)
 - Every invariant and intermediate assertion that we need for the proof can be expressed using the underlying (first order) logic.
- The relative completeness implies that the system is complete for first order logic with natural numbers with addition and subtraction as the only operators.

Deductive verification

- is not limited to finite state systems.
- can handle programs of various domains and datastructures (and even parametrized programs).
- can be applied directly to the code (in principle).
- can verify that the program is correct (but a bug can occur in compiler, in hardware, due to a wrong initial condition or difference between an assumed semantics of code and the real one, etc.).
- is not scalable.

In practice, deductive verification

- needs a great mental effort as it is mostly manual (the result depends strongly on the ingenuity of the people performing verification).
- is significantly slower than the typical speed of effective programming.
- is not performed frequently on the actual code (this is slowly changing with new tools).
- can be performed on basic algorithms or on abstractions of the code. The faithfulness of the translation of a program into an abstracted one can sometimes also be formally verified.

Theorem prover ACL2

<http://www.cs.utexas.edu/users/moore/acl2/>

- How it works?
- What is it good for?
- Including a **live show!**