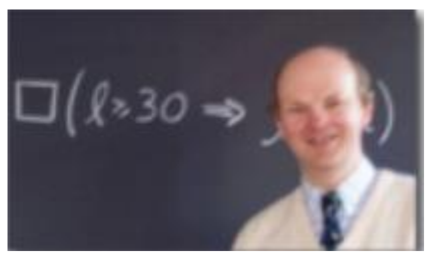
程序验证方法
研究生课程

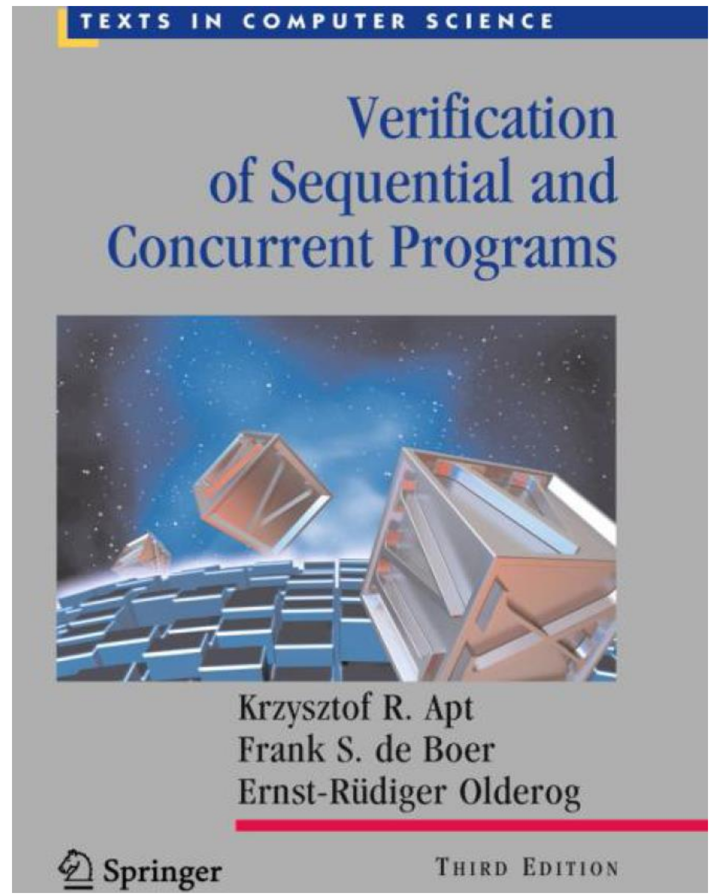
Chapter 1
Introduction

朱惠彪
华东师范大学 软件工程学院

Textbook：
Verification of Sequential and Concurrent Programs

Krzysztof R. Apt
Frank S. de Boer
Ernst-Rüdiger Olderog





Hoare Logic

- **In 1969 Hoare introduced an axiomatic methods for proving programs.**
- **This approach was partially based on the so-called intermediate assertion method of Floyd.**
- **Hoare's approach has received a great deal of attention during the last decade, and it has had a significant impact upon the methods of both designing and verifying programs.**

References:

- **C. A. R. Hoare. An Axiomatic Basis for Computer Programming. Commun. ACM 12(10): 576-580 (1969)**
- **David Gries: The Science of Programming. Texts and Monographs in Computer Science, Springer 1981, ISBN 978-0-387-96480-5, pp. i-xv, 1-368**
- **Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey - Part 1. ACM Trans. Program. Lang. Syst. 3(4): 431-483 (1981)**
- **Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey Part II: Nondeterminism. Theor. Comput. Sci. 28: 83-109 (1984)**
- **Krzysztof R. Apt, Ernst-Rüdiger Olderog: Fifty years of Hoare's logic. Formal Aspects Comput. 31(6): 751-807 (2019)**

Course Schedule

- Chapter 1: Introduction
- Chapter 3: while Programs
- Chapter 7: Disjoint Parallel Programs
- Chapter 8: Parallel Programs with Shared Variables
- Chapter 9: Parallel Programs
- with Synchronization
- Chapter 10: Nondeterministic Programs
- Chapter 11: Distributed Programs
- Chapter 12: Fairness

Part II
Deterministic
Programs

Part III
Parallel
Programs

Part IV
Nondeterministic
and Distributed
Programs

Course on Concurrent Program Verification

Class of programs	Syntax	Semantics	Proof theory	Case studies
while programs	3.1	3.2	3.3, 3.4	3.9
Disjoint parallel programs	7.1	7.2	7.3	7.4
Parallel programs with shared variables	8.1, 8.2	8.3	8.4, 8.5	8.6
Parallel programs with synchronization	9.1	9.2	9.3	9.4, 9.5
Nondeterministic programs	10.1	10.2	10.4	10.5
Distributed programs	11.1	11.2	11.4	11.5

What is "Proving Correctness of Program"？

- Correctness means that the programs enjoy certain desirable properties:
  - Sequential program: the delivery of correct results and termination.
  - Concurrent program: interference freedom, deadlock freedom or fair behaviour.

An Example of Concurrent System

Problem: Let f be a function from integers to integers with a zero. Write a concurrent program *ZERO* that finds such a zero. (page 4)

f : **integer** $\rightarrow$ **integer**

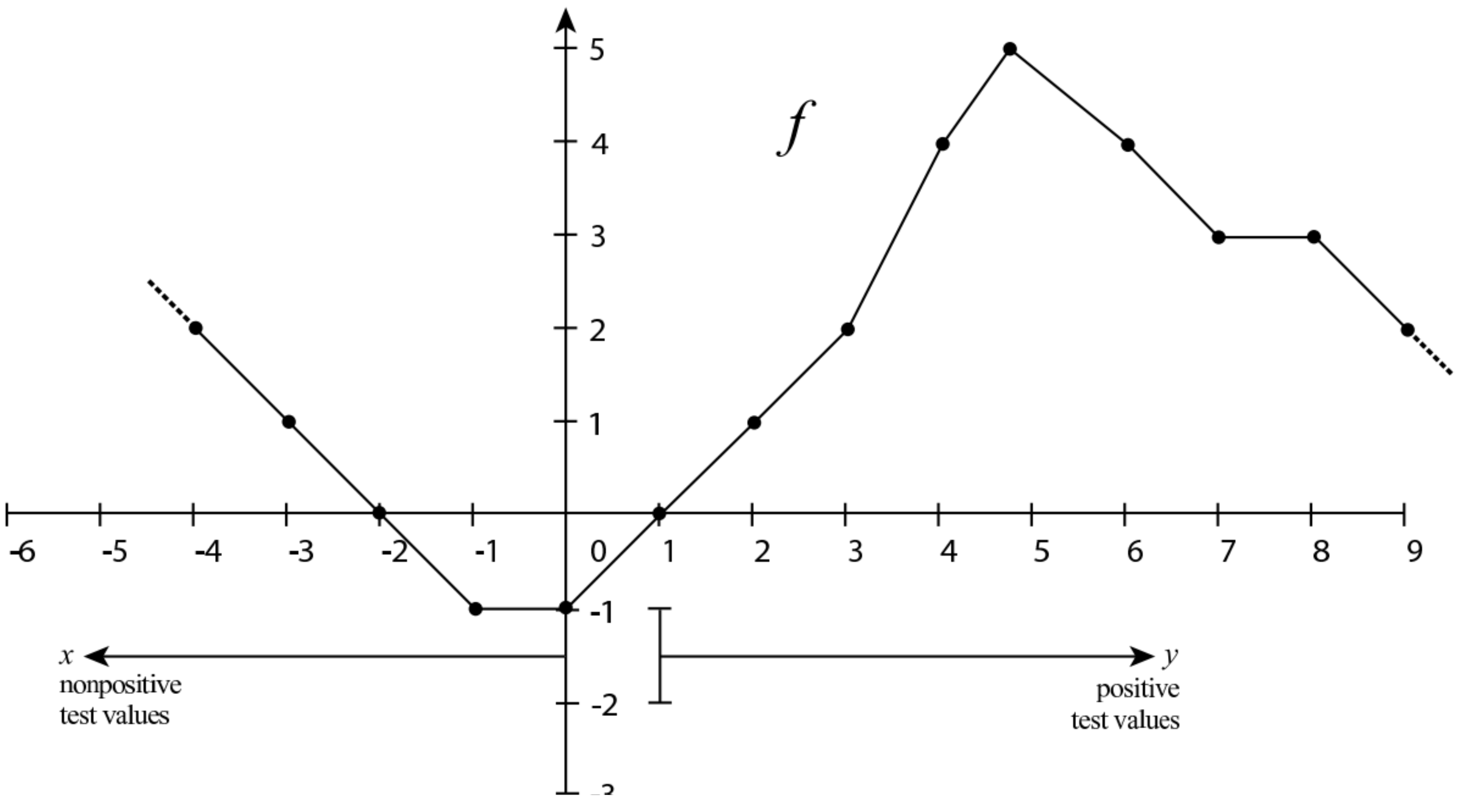


Fig. 1.1 Zero search of a function f : **integer** $\rightarrow$ **integer** split into two subproblems of finding a positive zero and a nonpositive zero.

Solution 1 (page 4)

$$\begin{aligned} S_1 \equiv\ & found := \textbf{false};\ \ x := 0; \\ & \textbf{while}\ \neg found\ \textbf{do} \\ & \qquad x := x + 1; \\ & \qquad found := f(x) = 0 \\ & \textbf{od}. \end{aligned}$$

$$\begin{aligned} S_2 \equiv\ & \underline{found := \textbf{false};}\ \ y := 1; \\ & \textbf{while}\ \neg found\ \textbf{do} \\ & \qquad y := y - 1; \\ & \qquad found := f(y) = 0 \\ & \textbf{od}. \end{aligned}$$

$$ZERO\text{-}1 \equiv [S_1 \| S_2]$$

- **Does ZERO-1 terminates? No, ‘found’ can be set to false eventually.**
- **Scenario: Let f have only one zero, *a positive one*. Consider an execution of *ZERO*-1, where initially only the program’s first component S1 is active, until it terminates when the zero of f is found. At this moment the second component S2 is activated, found is reset to false, and since no other zeroes of f exist, found is never reset to true. In other words, this execution of *ZERO*-1 never terminates.**
- **Obviously our mistake consisted of initializing found to false twice—once in each component.**

Solution 2 (page 5)

$$S_1 \equiv x := 0; \quad \mathbf{while}\ \neg found\ \mathbf{do} \quad x := x + 1; \quad found := f(x) = 0 \quad \mathbf{od}$$

$$S_2 \equiv y := 1; \quad \mathbf{while}\ \neg found\ \mathbf{do} \quad y := y - 1; \quad found := f(y) = 0 \quad \mathbf{od}.$$

$$ZERO\text{-}2 \equiv found := \mathbf{false};\ [S_1 \| S_2]$$

- Does ZERO-2 terminates? Not again, 'found' can be set to false eventually.
- Scenario: Suppose again that f has exactly one zero, a positive one, and consider an execution of *ZERO*-2 where, initially, its second component S2 is activated until it enters its loop. From that moment on only the first component S1 is executed until it terminates upon finding a zero. Then the second component S2 is activated again and so found is reset to false. Now, since no other zeroes of f exist, found is never reset to true and this execution of *ZERO*-2 will never terminate!

Solution 3 (page 6)

$$S_1 \equiv x := 0;$$

$$\begin{array}{l} \textbf{while } \neg found \textbf{ do} \\ \quad x := x + 1; \\ \quad \textbf{if } f(x) = 0 \textbf{ then } found := \textbf{true fi} \\ \textbf{od} \end{array}$$

$$S_2 \equiv y := 1;$$

$$\begin{array}{l} \textbf{while } \neg found \textbf{ do} \\ \quad y := y - 1; \\ \quad \textbf{if } f(y) = 0 \textbf{ then } found := \textbf{true fi} \\ \textbf{od}. \end{array}$$

$$ZERO\text{-}3 \equiv found := \textbf{false}; \ [S_1 \| S_2]$$

- Will ZERO-3 eventually terminates? No for some special cases.
- Scenario: But is it really a solution? Suppose that f has only positive zeroes, and consider an execution of *ZERO-3* in which the *first component S1 of the parallel program [S1||S2] is never activated*. Then this execution never terminates even though f has a zero.
- Fainess

Fairness

- What is the definition of parallel program?
  - An arbitrary interleaving of the executions of each component.
  - Each component progress with a positive speed (Fairness).

Solution 4 (page 8)

$$
\begin{aligned}
S_1 \equiv\ & x := 0; \\
& \quad \textbf{while}\ \neg found\ \textbf{do} \\
& \qquad \textbf{await}\ turn = 1\ \textbf{then}\ turn := 2\ \textbf{end}; \\
& \qquad x := x + 1; \\
& \qquad \textbf{if}\ f(x) = 0\ \textbf{then}\ found := \textbf{true fi} \\
& \quad \textbf{od}
\end{aligned}
$$

$$
\begin{aligned}
S_2 \equiv\ & y := 1; \\
& \quad \textbf{while}\ \neg found\ \textbf{do} \\
& \qquad \textbf{await}\ turn = 2\ \textbf{then}\ turn := 1\ \textbf{end}; \\
& \qquad y := y - 1; \\
& \qquad \textbf{if}\ f(y) = 0\ \textbf{then}\ found := \textbf{true fi} \\
& \quad \textbf{od}.
\end{aligned}
$$

one and half iterations

$$ZERO\text{-}4 \equiv turn := 1;\ found := \textbf{false};\ [S_1 \| S_2]$$

Is ZERO-4 fair? Yes, but can cause deadlock.

Solution 5 (page 8)

- **This solution is correct.**
- **It can, moreover, be improved.**

$$
\begin{aligned}
S_1 \equiv\ & x := 0; \\
& \textbf{while } \neg found \textbf{ do} \\
& \qquad \boxed{\textbf{await } turn = 1 \textbf{ then } turn := 2 \textbf{ end};} \\
& \qquad x := x + 1; \\
& \qquad \textbf{if } f(x) = 0 \textbf{ then } found := \textbf{true fi} \\
& \textbf{od}; \\
& \underline{turn := 2}
\end{aligned}
$$

- **An execution of *await B then R* end, temporarily blocks all other components of the parallel program until execution of R is completed.**

$$
\begin{aligned}
S_2 \equiv\ & y := 1; \\
& \boxed{\textbf{while } \neg found \textbf{ do}} \\
& \qquad \textbf{await } turn = 2 \textbf{ then } turn := 1 \textbf{ end}; \\
& \qquad y := y - 1; \\
& \qquad \textbf{if } f(y) = 0 \textbf{ then } found := \textbf{true fi} \\
& \textbf{od}; \\
& \underline{turn := 1.}
\end{aligned}
$$

$$ZERO\text{-}5 \equiv turn := 1;\ found := \textbf{false};\ [S_1 \| S_2]$$

Solution 6 (page 10)

$S_1 \equiv x := 0;$
$\quad \textbf{while } \neg found \textbf{ do}$
$\quad\quad \textbf{wait } turn = 1;$
$\quad\quad turn := 2;$
$\quad\quad x := x + 1;$
$\quad\quad \textbf{if } f(x) = 0 \textbf{ then } found := \textbf{true fi}$
$\quad \textbf{od};$
$\quad turn := 2$

$S_2 \equiv y := 1;$
$\quad \textbf{while } \neg found \textbf{ do}$
$\quad\quad \textbf{wait } turn = 2;$
$\quad\quad turn := 1;$
$\quad\quad y := y - 1;$
$\quad\quad \textbf{if } f(y) = 0 \textbf{ then } found := \textbf{true fi}$
$\quad \textbf{od};$
$\quad turn := 1$

- **The only difference from Solution 5 is that now component S2 can be activated —or as one says, *interfere*— between *wait turn = 1* and *turn := 2* of S1, and analogously for component S1.**

$$ZERO\text{-}6 \equiv turn := 1;\ found := \textbf{false};\ [S_1 \| S_2].$$

Program Correctness: A Summary

- **Sequential Program**
  - Partial correctness. If terminates, deliver a correct result.
  - Termination. If a program always terminate.
  - Absence of failures. e.g., no division by zero.
- **Concurrent Program**
  - Interference freedom. No component can manipulate in an undesirable way the shared variables.
  - Deadlock freedom. a parallel program does not end up in a situation where all nonterminated components are waiting indefinitely for a condition to become true.
  - Fairness. Each component progress with a positive speed.

History of Program Verification

- 1949. Turing. Origin ideas.
- 1967-1969. Floyd [an axiomatic verifcation method for flowcharts] and Hoare [hoare logic].
- 1976-1977. Owicki and Gries, Lesile Lamport [a verification method for concurrent program].
- 1980, 1981. Apt, Francez and de Roever [1980], and Levin and Gries [1981] [distributed programs].
- 1991. Boer. Parallel object-oriented language.

History of Program Verification

- **Limitations in early years:**
  - **the proof rules are designed only for the a posteriori verification of existing programs, not for their systematic development [systematic development of programs together with their correctness proofs, high-level system development].**
    - **Dijkstra [1976], Gries [1981], Backhouse [1986], Kaldewaij [1990], Morgan [1994]**
    - **Abrial [Event-B]**
  - **the proof rules reflect only the input/output behavior of programs, not properties of their finite or infinite executions as they occur [temporal logic].**
    - **Pnueli [1977].**
  - **the proof rules cannot deal with fairness [temporal logic].**

Thank You!