# Fifty years of Hoare's Logic

Krzysztof R. Apt
CWI, Amsterdam, The Netherlands
MIMUW, University of Warsaw, Warsaw, Poland

Ernst-Rüdiger Olderog
University of Oldenburg, Oldenburg, Germany

**Abstract**

We discuss a history of Hoare's logic.

## Contents

# 1 Introduction

Hoare's logic is a formalism allowing us to reason about program correctness. It was introduced fifty years ago in the seminal article [Hoa69] of Tony Hoare that focused on a small class of **while** programs, and was soon extended by him in [Hoa71a] to programs allowing local variables and recursive procedures. This approach became the most influential method of verifying programs, mainly because its syntax-oriented style made it possible to extend it to almost any type of programs. Also, thanks to parallel developments in program semantics, this approach leans itself naturally to a rigorous analysis based on the methods of mathematical logic. Since then several books appeared that discuss Hoare's logic, or at least have a chapter on it: [dB80, LS87, TZ88, Fra92, Win93, AFPdS11, Ben12], to name a few.

More than thirty years ago two surveys of Hoare's logic appeared, [Apt81], concerned with deterministic programs, and [Apt84], concerned with nondeterministic programs. At the beginning of nineties an extensive survey [Cou90] was published that also included an account of verification of parallel programs and a discussion of alternative approaches to program verification.

A systematic exposition of Hoare's logics for deterministic, nondeterministic and parallel programs appeared in our book [AO91]. The last edition of it, [AdBO09], written jointly with F.S. de Boer, extended the presentation to recursive procedures and object-oriented programs. In this paper we occasionally rely on the

material presented in this book, notably to structure the presentation, but we also analyze various matters omitted there, for example the issues concerning local variables, parameter mechanisms, auxiliary rules, the full power of ALGOL 60, and the problem of completeness. We also discuss various alternative approaches.

The literature on the subject is really vast. In particular, according to Google Scholar, the original article [Hoa69] has been cited more than 7000 times. This forced us to make some selection in the presented material. Some omissions, such as the treatment of the now hardly used **goto** statement or coroutines, or logical analysis of issues related to completeness, were dictated by our effort to trace and explain the developments that withstood the test of time.

Further, we did not introduce any program semantics. Consequently, we do not establish any soundness or completeness results. Instead, we focus on a systematic account of the established results combined with an explanation of the reasons some concepts were introduced, and on a discussion of some, occasionally subtle, ways Hoare's logic differs from customary logics.

We begin the exposition by discussing in the next section the contributions to program verification by Alan Turing and Robert Floyd that preceded those of Hoare. Then, in Section 3, we discuss Hoare's initial contributions that focused on the **while** programs and programs with recursive procedures, though we extend the exposition by an account of program termination. Next, we discuss in Section 4 the soundness and completeness of the discussed proof systems. An essential difference between Hoare's logic and first-order logic has to do with the features specific to programming languages, such as subscripted variables, local variables, and parameter mechanisms. We discuss these matters in Section 5. This provides a natural starting point for an account of verification of programs with arbitrary procedures, notably procedures that allow procedures as parameters. This forms the subject of Section 6.

In Section 7 we discuss verification of nondeterministic programs, the corresponding issue of fairness, and verification of probabilistic programs. Then, in Section 8 we focus on the verification of parallel and distributed programs. Next, in Section 9, we provide an account of verification of object-oriented programs. The final two sections, 10 and 11, shed light on alternative approaches to program verification and attempt to explain and assess the impact of Hoare's logic.

# 2 Precursors

## 2.1 Turing

The concern about correctness of computer programs is as old as computers themselves. In 1949, Alan Turing gave a presentation entitled "Checking a Large Routine" at a conference in Cambridge U.K. at the occasion of the launching the computer EDSAC (Electronic Delay Storage Automatic Calculator), published as [Tur49]. F.L. Morris and C.B. Jones recovered [MJ84] the original typescript of Turing's presentation and made it available for a wider audience, thereby correcting several typing errors. Turing started by asking

> "How can one check a routine in the sense of making sure that it is right?"

and proposed that

> "... the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows."

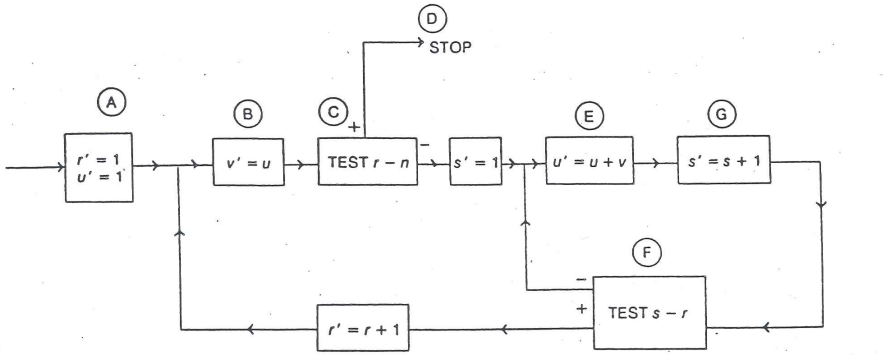


Figure 1: Turing's flowchart, reconstructed by F.L. Morris and C.B. Jones [MJ84].

| STORAGE LOCATION | (INITIAL) Ⓐ $k = 6$ | Ⓑ $k = 5$ | Ⓒ $k = 4$ | (STOP) Ⓓ $k = 0$ | Ⓔ $k = 3$ | Ⓕ $k = 1$ | Ⓖ $k = 2$ |
|---|---|---|---|---|---|---|---|
| s 27<br>r 28<br>n 29<br>u 30<br>v 31 | $n$ | $r$<br>$n$<br>$\lfloor r$ | $r$<br>$n$<br>$\lfloor r$<br>$\lfloor r$ | $n$<br>$\lfloor n$ | $s$<br>$r$<br>$n$<br>$s \lfloor r$<br>$\lfloor r$ | $s+1$<br>$r$<br>$n$<br>$(s+1) \lfloor r$<br>$\lfloor r$ | $s$<br>$r$<br>$n$<br>$(s+1) \lfloor r$<br>$\lfloor r$ |
| | TO Ⓑ<br>WITH $r' = 1$<br>$u' = 1$ | TO Ⓒ | TO Ⓓ<br>IF $r = n$<br>TO Ⓔ<br>IF $r < n$ | | TO Ⓖ | TO Ⓑ<br>WITH $r' = r + 1$<br>IF $s \geqslant r$<br>TO Ⓔ<br>WITH $s' = s + 1$<br>IF $s < r$ | TO Ⓕ |

Figure 2: Turing's assertions, reconstructed by F.L. Morris and C.B. Jones [MJ84]. Turing writes $\lfloor n$ for the faculty of $n$.

Turing demonstrated his ideas for a flowchart program with nested loops computing the factorial $n!$ of a given natural number $n$, where multiplication is achieved by repeated addition; see Figure1. Note that the effect of a command in the flowchart is represented by an equation like $u' = u + v$, where $u'$ denotes the value of the variable $u$ after the execution of the command. Today, this notation is still in use in logical representations of computation steps like in the specification language Z (see, e.g., [Spi92]) and bounded model checking.

Turing referred already to *assertions*. In the example he presented them in the form of a table referring to the numbers of the locations storing the variables $s, r, n, u, v$, see Figure 2. From today's viewpoint these assertions are admittedly very specific and difficult to read.

Turing was not only concerned with delivering correct values, but also with termination. He wrote

> "Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the

form of a quantity which is asserted to decrease continually and vanish when the machine stops."

This refers already to the concept of a termination function. Turing stated a global termination function for the example program, i.e., an integer expression yielding a non-negative value that decreases with every step of the program.

Summarizing, Turing introduced the notions of assertions and termination functions but did not state loop invariants and local termination functions for the two loops of the program. Still, as we explain the Appendix, his approach can be represented within the framework of Hoare's logic.

### 2.2 Floyd

Robert Floyd was the first to propose in [Flo67] a fully formal method for proving the correctness of flowchart programs known as *inductive-assertions method*. Here the assertions are logical formulas in terms of the variables appearing in the flowcharts. The begin of the flowchart is annotated with an assertion stating the assumptions under which the flowchart is supposed to work. The end of the flowchart is annotated with an assertion specifying the desired result. To verify that these input-output annotations are correct, each loop of the flowchart needs to be cut and annotatated with an assertion that should hold whenever the control reaches this cut point. The assertion should thus be an *invariant* at the cut point. Floyd states rules how to verify this by completing the flowchart so that there is at least one assertion between any two subsequent statements. The rules explain how to modify a given assertion when passing a test statement and when passing an assignment statement. When two assertions are adjacent to the same arc then the logical implication has to hold in the direction of the arc.

In Figure 3 we show Turing's example as a flowchart with annotations according to Floyd's method. At the begin $B$ of the flowchart the annotation $n \geq 1$ states the assumption of the computation, at the end $E$ the annotation $v = n!$ specifies the desired result that should hold whenever the computation reaches $E$. To verify that this annotation is correct, every loop has to be cut and annotatated by an *invariant*. In this example, we cut the loops at the bullet points and annotate them with the assertions

$$P_1 \equiv v = r! \wedge u = r! \wedge 1 \leq r \leq n$$

and

$$P_2 \equiv v = r! \wedge u = s \cdot v \wedge 1 \leq s \leq r + 1 \leq n.$$

## 3 Hoare's Contributions

### 3.1 Reasoning about while programs

To reason about programs Hoare introduced in [Hoa69] a new notation

$$P \ \{S\} \ Q,$$

with the interpretation

> "If the assertion $P$ is true before initiation of a program $S$, then the assertion $Q$ will be true on its completion."

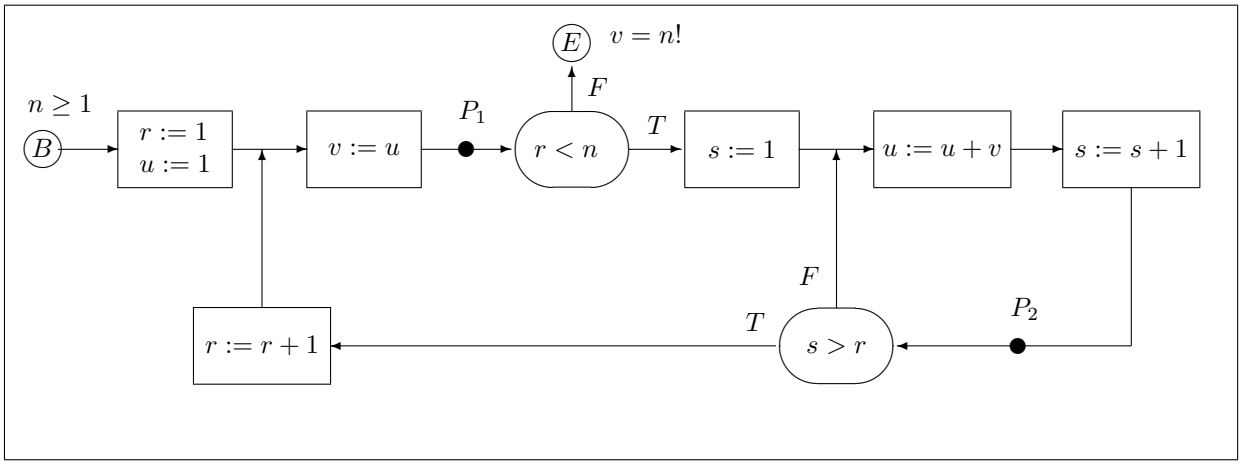


Figure 3: Turing's example as a flowchart with annotations according to Floyd's method, where $P_1$ and $P_2$ are the following invariants: $P_1 \equiv v = r! \wedge u = r! \wedge 1 \leq r \leq n$ and $P_2 \equiv v = r! \wedge u = s \cdot v \wedge 1 \leq s \leq r+1 \leq n$. See also [dB75].

Nowadays one rather writes

$$\{P\}\ S\ \{Q\}$$

so that additional assertions can be freely inserted in the program text, by putting the $\{\cdot\}$ brackets around them. Such a possibility will turn out to be especially important when reasoning about parallel programs. In what follows we shall use the latter notation. In this context $P$ is referred to as a *precondition* and $Q$ as a *postcondition*.

Subsequently Hoare introduced an axiom to reason about the assignment statement and the proof rules to reason about the program composition and the **while** statement. He also introduced two consequence rules, now combined into one, that allow one to strengthen the precondition and to weaken the postcondition. He then used these axioms and rules to establish correctness of the following simple program, let us call it `DIV`, that finds "the quotient $q$ and a remainder $r$ obtained on dividing $x$ by $y$":

$$r := x;\ q := 0;\ \textbf{while}\ y \leq r\ \textbf{do}\ r := r - y;\ q := 1 + q\ \textbf{od}.$$

All variables are assumed to range over the nonnegative integers.

In what follows we review these steps. The assignment axiom has the form

ASSIGNMENT

$$\{P[x := t]\}\ x := t\ \{P\},$$

where $x$ is a variable, $t$ is an expression, and $P[x := t]$ is the result of substituting $t$ for all free occurrences of $x$.

The already mentioned consequence rule has the following form:

| Line number | Formal proof | Justification |
|---|---|---|
| 1 | $\mathbf{true} \to x = x + y \cdot 0$ | logic |
| 2 | $\{x = x + y \cdot 0\}\ r := x\ \{x = r + y \cdot 0\}$ | ASSIGNMENT |
| 3 | $\{x = r + y \cdot 0\}\ q := 0\ \{x = r + y \cdot q\}$ | ASSIGNMENT |
| 4 | $\{\mathbf{true}\}\ r := x\ \{x = r + y \cdot 0\}$ | CONSEQUENCE (1,2) |
| 5 | $\{\mathbf{true}\}\ r := x;\ q := 0\ \{x = r + y \cdot q\}$ | COMPOSITION (4,3) |
| 6 | $x = r + y \cdot q \wedge y \leq r \to x = (r - y) + y \cdot (1 + q)$ | logic |
| 7 | $\{x = (r - y) + y \cdot (1 + q)\}\ r := r - y\ \{x = r + y \cdot (1 + q)\}$ | ASSIGNMENT |
| 8 | $\{x = r + y \cdot (1 + q)\}\ q := 1 + q\ \{x = r + y \cdot q\}$ | ASSIGNMENT |
| 9 | $\{x = (r - y) + y \cdot (1 + q)\}\ r := r - y;\ q := 1 + q\ \{x = r + y \cdot q\}$ | COMPOSITION (7,8) |
| 10 | $\{x = r + y \cdot q \wedge y \leq r\}\ r := r - y;\ q := 1 + q\ \{x = r + y \cdot q\}$ | CONSEQUENCE (6,9) |
| 11 | $\{x = r + y \cdot q\}\ \mathbf{while}\ y \leq r\ \mathbf{do}\ r := r - y;\ q := 1 + q\ \mathbf{od}$ $\{\neg y \leq r \wedge x = r + y \cdot q\}$ | WHILE (10) |
| 12 | $\{\mathbf{true}\}\ r := x;\ q := 0;\ \mathbf{while}\ y \leq r\ \mathbf{do}\ r := r - y;\ q := 1 + q\ \mathbf{od}$ $\{\neg y \leq r \wedge x = r + y \cdot q\}$ | CONSEQUENCE (5,11) |

The arguments in the right column of the rules refer to the line numbers to which they were applied and 'logic' indicates that the relevant formulas are true (Hoare referred to specific axioms of Peano arithmetic).

Figure 4: Correctness proof of the `DIV` program

CONSEQUENCE

$$\frac{P \to P_1, \{P_1\}\ S\ \{Q_1\}, Q_1 \to Q}{\{P\}\ S\ \{Q\}}$$

Here it is assumed that the mentioned implications can be established in some further unspecified proof system exclusively concerned with the assertions. (Hoare just referred to $\to$ as 'logical implication'.)

The final two rules were:

COMPOSITION

$$\frac{\{P\}\ S_1\ \{R\}, \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

and

WHILE

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{P \wedge \neg B\}}$$

Nowadays one refers to the assertion $P$ that satisfies the premise of this rule as a *loop invariant*.

Hoare's correctness proof of the `DIV` program is presented in Figure 4. (Hoare wrote the postcondition of the conclusion of the WHILE rule as $\neg B \wedge P$ and this is how it is recorded in Figure 4.) It yields the desired conclusion that $q$ is the quotient and $r$ the remainder resulting from dividing $x$ by $y$. The crucial step in this proof is line 10 that clarifies the role played by the assertion $x = r + y \cdot q$. This line establishes that $x = r + y \cdot q$ is a loop invariant of the considered **while** statement and its discovery is essential for the proof to succeed.

As pointed out in [JR10] the assignment axiom was originally proposed in [Kin69], the PhD thesis of J. King. From [Flo67] one can distill a more complex assignment axiom

ASSIGNMENT I

$$\{P\}\ x := t\ \{\exists y : (P[x := y] \wedge x = t[x := y])\}.$$

that reasons "forward" starting from the precondition $P$.

The striking simplicity of the ASSIGNMENT axiom reveals a close relation between the assignment statement and the substitution operation. This is achieved, in contrast to Floyd's approach, by reasoning 'backwards', so starting from the postcondition $P$. The adoption of this axiom by Hoare probably influenced a couple of years later Edsger W. Dijkstra to propose the weakest precondition semantics that adopted this reasoning 'backward' to all program statements. We shall discuss this alternative approach to program verification in Section 10. From the mathematical point of view Hoare's proof rules and axioms form an unusual mix: the assignment axiom adopts the 'backward' reasoning, while all the proof rules embrace the 'forward' reasoning.

Hoare's paper turned out to be a beginning of a far reaching change in reasoning about programs, resulting from moving from flowcharts to programs expressed in the customary textual form. This opened the way to reasoning about programs that cannot be readily expressed as flowcharts, for example, recursive procedures or programs with variable declarations. Also it made it possible to adopt a syntax-directed reasoning about programs by using their structure as a guidance in organizing the proof.

A related, implicit, feature of the proof system proposed by Hoare is that it encourages program development by allowing one to first specify the desired preconditions and postconditions of a program component and subsequently to look for a program fragment for which the corresponding correctness statement can be established. Hoare took a lead in this novel view of program correctness by publishing in [Hoa71b] a correctness proof of the `FIND` program the purpose of which is to find the $f$th largest element of an array $A[1 : N]$ by rearranging its elements so that upon termination

$$A[1], A[2], \ldots, A[f-1] \leq A[f] \leq A[f+1], \ldots, A[N].$$

The program is very subtle —it uses a triply nested **while** loop— and as a result its correctness proof is highly nontrivial. The proof is not carried out in the proof system of [Hoa69] but from the way it is written it is clear that it can be done so. In fact, Hoare refers in a number of places to *invariants* that he defines as formulas that remain true throughout the execution of the program independently of the values of the program variables.

In [Hoa71b], Hoare also showed *termination* of the program. Since this property is not captured by his proof system [Hoa69], he used informal arguments. Nowadays, one talks of *partial correctness*, which refers to the conditional statement 'if the program terminates starting from a given precondition, then it satisfies the desired postcondition' and this is precisely what Hoare's proof system allows one to accomplish. A more demanding property is *total correctness*, which stipulates that all program computations starting from a given precondition terminate and satisfy the desired postcondition. We shall formalize these notions in the next section.

According to this terminology, Hoare established total correctness of the program `FIND`. He noticed that the termination proof required invariants in addition to those needed for proving partial correctness. However, he did not introduce the concept of a *termination function* (sometimes called a *bound function* or a *variant*) with a corresponding proof rule for total correctness of **while** programs.

Hoare expressed already in [Hoa71b] the desire for computer support in "formulating the lemmas, and perhaps even checking the proofs." Only much later, Filliâtre [Fil07] published a mechanized proof of `FIND` using the theorem prover `Coq` and following Hoare's proof as closely as possible. Filliâtre noticed that Hoare's informal termination proof does not meet the requirements of a termination function in the sense that the additional invariants used by Hoare are not real invariants.

A similar in style contribution is [Hoa72b], in which a correctness proof was given of a program encoding the sieve of Eratosthenes. The difference was that the program was developed together with its correctness proof and presented using non-recursive procedures and classes, drawing on the contemporary works of E.W. Dijkstra on structured programming and O.J. Dahl on the object-oriented programming language SIMULA 67, which appeared as chapters in [DDH72]. These two contributions of Hoare, [Hoa71b] and [Hoa72b], showed that his original logic could be seen not only as a tool to verify programs but also as a guide to design correct programs. These ideas were further developed by Dijkstra, notably in his book [Dij76a].

All approaches to proving program termination formalize Floyd's [Flo67] observation that

> "Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely."

The challenge is to incorporate such a reasoning into Hoare's framework in a simple way. The first extension of Hoare's proof system to total correctness was proposed in [MP74], but the proposed strengthening of the WHILE rule was somewhat elaborate. In [Har79] the appropriate rule took a simpler form:

WHILE I

$$\frac{P(n+1) \to B,\ \{P(n+1)\}\ S\ \{P(n)\},\ P(0) \to \neg B}{\{\exists n\, P(n)\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{P(0)\}}$$

where $P(n)$ is an assertion with a free variable $n$ that does not appear in $S$ and ranges over natural numbers.

Still, a disadvantage of this rule is that it requires to find a parameterized loop invariant $P(n)$ such that the value of $n$ decreases exactly by 1 with each loop iteration. Such a precise information is not needed to establish termination and sometimes is difficult to come up with. Additionally, as witnessed by Hoare's correctness proof of the `FIND` program, it is often inconvenient to reason about partial correctness and termination at the same time. These concerns were addressed in the following proof rule introduced in [OG76a] that adds two new premises to the original WHILE rule:

WHILE II

$$\frac{\begin{array}{l}\{P \wedge B\}\ S\ \{P\},\\ \{P \wedge B \wedge t = z\}\ S\ \{t < z\},\\ P \to t \geq 0\end{array}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{P \wedge \neg\, B\}}$$

where $t$ is an integer expression, called a *termination function*, and $z$ is an integer variable that does not appear in $P, B, t$ or $S$.

This proof rule corresponds to Dijkstra's modification of his weakest precondition semantics proposed in [Dij76b] and reproduced as [Dij82]. Returning to the above `DIV` program note that it does not terminate

when $y = 0$. To prove its termination one needs to assume that initially $x \geq 0 \wedge y > 0$ and use a stronger loop invariant, namely $P' \equiv r \geq 0 \wedge y > 0 \wedge x = r + y \cdot q$. The termination function is particularly simple here: it is just $r$. The relevant claims, so

$$\{P' \wedge y \leq r \wedge r = z\}\ r := r - y;\ q := 1 + q\ \{r < z\}$$

and

$$P' \to r \geq 0,$$

are straightforward to prove.

## 3.2 Reasoning about recursive procedures

Let us continue with another milestone in the history of Hoare's logic. In [FH71] Foley and Hoare established correctness of the program `Quicksort`, originally proposed by Hoare in [Hoa61]. Foley and Hoare stated:

> "The purpose of the program `Quicksort` is to sort the elements $a[m]$ to $a[n]$ of an array into ascending order, while leaving untouched those below $a[m]$ and above $a[n]$."

The main difficulty was that `Quicksort` uses recursion. (Actually it was the first non-trivial example of a successful use of recursion.) This required appropriate proof rules that were introduced by Hoare in [Hoa71a].

In what follows given a program $S$ we denote by $change(S)$ the set of variables that are subject to change in it. Further, we use $\mathbf{proc}\, p(\mathbf{x} : \mathbf{v}) : S$ to denote the declaration of a procedure $p$ with the body $S$ and two sorts of formal parameters: $\mathbf{x}$ is the list of all global variables of $S$ which are subject to change by $S$, i.e., $\{\mathbf{x}\} = change(S)$, and $\mathbf{v}$ is the list of all other global variables of $S$ (read-only variables). (Hoare actually used a slightly different notation that is now obsolete.)

Legal procedure calls are of the form $\mathbf{call}\, p(\mathbf{a} : \mathbf{e})$, where

- $\mathbf{a}$ is a list of distinct variables of the same length as $\mathbf{x}$ that are substituted for $\mathbf{x}$,
- $\mathbf{e}$ is a list of expressions not containing any variable of $\mathbf{a}$, of the same length as $\mathbf{v}$, that are substituted for $\mathbf{v}$.

The following proof rule dealt with a 'generic' procedure call $\mathbf{call}\, p(\mathbf{x} : \mathbf{v})$:

RECURSION

$$\frac{\{P\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{Q\} \vdash \{P\}\ S\ \{Q\}}{\{P\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{Q\}}$$

where the procedure $p$ is declared by $\mathbf{proc}\, p(\mathbf{x} : \mathbf{v}) : S$.

(Hoare actually included the procedure declaration as an additional premise of the rule.) What is the intuition behind this rule? Hoare states in [Hoa71a] that it permits

> "the use of the desired conclusion as a hypothesis in the proof of the body itself."

More specifically, the symbol $\vdash$ in the premise denotes the provability relation. So this rule is actually a metarule. According to [FH71] the premise of this rule

"permits $\{P\}$ **call** $p(\mathbf{x} : \mathbf{v})$ $\{Q\}$ to be assumed as a hypothesis in the proof of $\{P\}$ $S$ $\{Q\}$."

This proof is supposed to be carried out using the remaining axioms and proof rules. The conclusion of the rule then coincides with this hypothesis.

To transfer a result established by the recursion rule to any other procedure call with actual parameters, say the lists **a** and **e**, the following substitution rule was introduced:

SUBSTITUTION

$$\frac{\{P\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{Q\}}{\{P[\mathbf{k} := \mathbf{k}', \mathbf{x} := \mathbf{a}, \mathbf{v} := \mathbf{e}]\}\ \mathbf{call}\, p(\mathbf{a} : \mathbf{e})\ \{Q[\mathbf{k} := \mathbf{k}', \mathbf{x} := \mathbf{a}, \mathbf{v} := \mathbf{e}]\}}$$

where the following holds for the substitutions applied to $P$ and $Q$:

- **k** is a list of free variables of $P$ or $Q$ that do not occur in **x** or **v**, but which occur in **a** or **e**. Then $\mathbf{k}'$ is a list of fresh variables of the same length as **k** that are substituted for **k**,
- **a** and **e** are such that the call **call** $p(\mathbf{a} : \mathbf{e})$ is legal.

So the substitution $[\mathbf{x} := \mathbf{a}, \mathbf{v} := \mathbf{e}]$ of the formal parameters by the actual ones is carried out together with an appropriate renaming $[\mathbf{k} := \mathbf{k}']$ of the 'potentially conflicting' variables in $P$ and $Q$.

Hoare noted that the above two rules are not sufficient to reason about recursive procedures. To have a more powerful proof method, he introduced the following rule, where $\mathit{free}(P)$ stands for the set of free variables in an assertion $P$ and similarly with $\mathit{free}(P, Q)$:

ADAPTATION

$$\frac{\{P\}\ \mathbf{call}\, p(\mathbf{a} : \mathbf{e})\ \{Q\}}{\{\exists\, \mathbf{z}\ (P \wedge \forall \mathbf{a}\ (Q \to R))\}\ \mathbf{call}\, p(\mathbf{a} : \mathbf{e})\ \{R\}}$$

where **z** is a list of variables with $\{\mathbf{z}\} = \mathit{free}(P, Q) \setminus (\mathit{free}(R) \cup \{\mathbf{a}, \mathbf{e}\})$.

The precondition of the conclusion of this rule looks complicated. What does it express? Hoare explained in [Hoa71a]:

> "If $R$ is the desired result of executing a procedure call, **call** $p(\mathbf{a} : \mathbf{e})$, and $\{P\}$ **call** $p(\mathbf{a} : \mathbf{e})$ $\{Q\}$ is already given, what is the weakest precondition $W$ such that $\{W\}$ **call** $p(\mathbf{a} : \mathbf{e})$ $\{R\}$ is universally valid? It turns out that this precondition is $\exists\, \mathbf{z}\ (P \wedge \forall \mathbf{a}\ (Q \to R))$."

To deal with the declarations of local variables Hoare introduced the following rule:

DECLARATION

$$\frac{\{P\}\ S[x := y]\ \{Q\}}{\{P\}\ \mathbf{begin\ var}\ x;\ S\ \mathbf{end}\ \{Q\}}$$

where $y \notin \mathit{free}(P, Q)$ and $y$ does not appear in $S$ unless the variables $x$ and $y$ are the same.

Additionally, the following proof rule, originally proposed in [Lau71], to reason about the conditional statement was used:

CONDITION

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{\textbf{if } B \textbf{ then } P \textbf{ else } Q\}\ \textbf{if } B \textbf{ then } S \textbf{ fi } \{Q\}}$$

The correctness proof of `Quicksort` by Foley and Hoare in [FH71] was carried out using the above proof rules for partial correctness, originally presented in [Hoa71a]. The authors formulated two correctness criteria that should hold upon termination of `Quicksort`:

- *Sorted*: the output array should be sorted within the given bounds $m$ and $n$.
- *Perm*: the output array should be a permutation of the original input array within the given bounds $m$ and $n$ but untouched outside these bounds.

The proof established these properties simultaneously, using appropriate assertions. On termination of `Quicksort` only very few remarks were spent. Since the recursive procedure `Quicksort` calls the non-recursive procedure `Partition`, the correctness of `Partition` was also shown. `Partition` is an instantiation of a part of the **while** program `FIND` (see Subsection 3.1).

In [AdBO09] a detailed modular proof of total correctness `Quicksort` was given. Modular means that first the property *Perm* was proved and next, based on this result, the property *Sorted*. Also, termination was proved separately. These proofs relied on corresponding results for `Partition` that were established first. In particular, the termination proof of `Partition` required a more subtle invariant for the termination function of the outer loop than anticipated by Hoare in his termination proof of `FIND` [Hoa71b]. This agrees with the observation made by Filliâtre [Fil07].

A tool-supported correctness proof of `Quicksort` and (recursive and sequential) variants of it has recently been reported in [CDE+16]. The authors use the Hoare's logic based verification tool `Dafny` [Lei10] that builds upon the SMT solver `Z3` of [dMB08].

# 4 Soundness and Completeness Matters

## 4.1 Soundness

In mathematical logic a standard way to judge the adequacy of a proof system is by means of the soundness and completeness concepts. It is then natural to address these matters for the proof systems introduced in the previous section. This requires some care since the CONSEQUENCE rule also uses customary formulas as premises, the WHILE II rule refers to integer variables and expressions, while the RECURSION rule refers in its premise to the provability relation.

For these considerations one needs to define some semantics with respect to which the introduced axioms and proof rules can be assessed. The first step is to define semantics of the underlying programming concepts. This can be done in a number of ways. The common denominator of all approaches is the concept of a *state*, a function that assigns appropriate values to all variables. In the case of simple variables these values should be taken from the domain corresponding to the variable type. In the case of array variables such a value should be a function from the domain of the array to the range type. Using such a function we can then assign the values to subscripted variables. As the complexity of the considered programming language grows, the concept of the state gets more complex. At this stage we limit ourselves to the notion of a state that assigns values to all simple and array variables.

In Hoare's logic the types of the variables in the considered programs, for instance in the program `DIV` in Figure 4, are usually omitted and one simply assumes that all variables are typed and that the considered programs are correctly typed.

The second step is to define *semantics of the programs*. Several approaches were proposed in the literature. Their discussion and comparison is beyond the scope of this paper. For the sake of the subsequent discussion we assume a semantics of the programs that allows us to define *computations* of each considered program, which are identified here with the sequences of states that can be generated by it.

The final step is to define when a state satisfies an assertion and when the implications used in the premises of the CONSEQUENCE rule are true. To proceed in a systematic way we need to recall some basic notions from mathematical logic. Assume a first-order language $\mathcal{L}$. An *interpretation* $I$ for $\mathcal{L}$ consists of

- a non-empty domain $D$,
- an assignment to each $n$-ary function symbol in $\mathcal{L}$ an $n$-ary function over $D$,
- an assignment to each $n$-ary predicate symbol $\mathcal{L}$ an $n$-ary relation over $D$.

Given an interpretation $I$ each state $\sigma$ is just a function from the set of variables to the domain $D$. The definition of $I$ disregards our assumption that all variables are typed. However, it is easy to amend it by replacing the domain $D$ by the set of typed domains and by stipulating that each variable ranges over the domain associated with its type. Another, natural, adjustment can be done to include array variables in this framework.

The next step is to define when, given an interpretation $I$, a state $\sigma$ *satisfies* a formula $\phi$ of $\mathcal{L}$, written as $\sigma \models_I \phi$, a definition we omit. We then say that a formula $\phi$ is *true* in $I$, written as $\models_I \phi$, if for all states $\sigma$ we have $\sigma \models_I \phi$.

Let us return now to assertions and programs. Suppose that all assertions are formulas in a given first-order language $\mathcal{L}$ and that all considered programs use function and predicate symbols of $\mathcal{L}$. Each interpretation $I$ for $\mathcal{L}$ then determines the set of states and thus allows us for each program to define the set of its computations over $I$. This in turn allows us to introduce the following notions.

Fix an interpretation $I$. We say that the correctness formula $\{P\}\ S\ \{Q\}$ is true in $I$ in the sense of *partial correctness* if the following holds:

every terminating computation of $S$ over $I$ that starts
in a state that satisfies $P$ ends in a state that satisfies $Q$.

Further, we say that the correctness formula $\{P\}\ S\ \{Q\}$ is true in $I$ in the sense of *total correctness* if the following holds:

every computation of $S$ over $I$ that starts in a state that
satisfies $P$ terminates and ends in a state that satisfies $Q$.

Denote now by $\mathcal{H}$ the original proof system of Hoare presented in Subsection 3.1 and by $\mathcal{HT}$ the proof system obtained from $\mathcal{H}$ by replacing the WHILE rule by the WHILE II rule. The following two results capture the crucial properties of these proof systems.

**Soundness Theorem 1** Consider a proof of the correctness formula $\{P\}\ S\ \{Q\}$ in the system $\mathcal{H}$ that uses a set $\mathcal{A}$ of assertions for the CONSEQUENCE rule. Consider an interpretation $I$ in which all assertions from $\mathcal{A}$ are true. Then $\{P\}\ S\ \{Q\}$ is true in $I$ in the sense of partial correctness.

This property of the proof system $\mathcal{H}$ is called soundness in the sense of *partial correctness*. It was first established in [HL74] w.r.t. the relational semantics in which programs were represented as binary relations on the sets of states.

The following counterpart of it justifies the reasoning about termination. It is, however, important to read it in conjunction with the qualifications that follow.

**Soundness Theorem 2** Consider a proof of the correctness formula $\{P\}\ S\ \{Q\}$ in the system $\mathcal{HT}$ that uses a set $\mathcal{A}$ of assertions for the CONSEQUENCE rule. Consider an interpretation $I$ in which all assertions from $\mathcal{A}$ are true. Then $\{P\}\ S\ \{Q\}$ is true in $I$ in the sense of total correctness.

This property of the proof system $\mathcal{HT}$ is called soundness in the sense of *total correctness*. The first proof was given in [Har79] and referred to the proof system in which instead of the WHILE II rule the WHILE I rule was used. In this rule the assertion $P(n)$ refers to a free variable $n$ that ranges over natural numbers. To guarantee the correct interpretation of such assertions one needs to ensure that in each state such a variable $n$ is interpreted as a variable of type 'natural number'. In [Har79] this is achieved by considering assertion languages that extend the language of Peano arithmetics and by limiting one's attention to *arithmetic interpretations*. These are interpretations that extend the standard model for arithmetic. Additionally one stipulates that there is a formula in the assertion language that, when interpreted, encodes finite sequences of the domain elements by one element. (This requirement is only needed for the completeness.)

In the case of the WHILE II rule similar considerations are needed to ensure the correct interpretation of the integer expression $t$ and the integer variable $z$. The corresponding result was given in [AO91] and reproduced in the subsequent two editions of the book. As in [AO91] all variables are assumed to be typed, $t$ and $z$ are correctly interpreted and the need for the arithmetic interpretations disappears.

## 4.2 Completeness

The completeness of the $\mathcal{H}$ and $\mathcal{HT}$ proof systems aims at establishing some form of converse of the Soundness Theorems. It is a subtle matter and requires a careful analysis. Let us start with the proof system $\mathcal{H}$. It is incomplete for an obvious reason. Consider for instance the correctness formula $\{\mathbf{true}\}\ x := 0\ \{x \neq 1\}$. By the ASSIGNMENT axiom we get $\{0 \neq 1\}\ x := 0\ \{x \neq 1\}$. To conclude the proof we need to establish the obvious implication $\mathbf{true} \to 0 \neq 1$ and apply the CONSEQUENCE rule. However, we have no proof rules and axioms that allow us to derive this implication.

A way out is to augment $\mathcal{H}$ by a proof system allowing us to prove all true implications between the assertions. Unfortunately, in general such proof systems do not exist. This is a consequence of two results in mathematical logic. The first one states that the set of theorems in a proof system with recursive sets of axioms and finitary rules is recursively enumerable. The second one is Tarski's undefinability theorem of [Tar36]. It implies that the set of formulas of Peano arithmetic that are true in the standard model of arithmetic is not arithmetically definable, so in particular not recursively enumerable. This means that completeness of the proof system $\mathcal{H}$ cannot be established even if we add to it a proof system concerned with the assertions.

A natural solution is to try to establish completeness *relative* to the set of true assertions, that is to use the set of true assertions as an 'oracle' that can be freely consulted in the correctness proof. However, even then a problem arises because the assertion language can fail to be sufficiently expressive. Namely [Wan78] exhibited a true correctness formula that cannot be proved because the necessary intermediate assertions

cannot be expressed in the considered assertion language. Simpler examples of such assertion languages were provided in [BT82].

A solution to these complications was proposed by S.A. Cook in [Coo78]. To explain it we need to introduce some additional notions. We call set of states $\Sigma$ is *definable* in an interpretation $I$ iff for some formula $\phi$ we have $\Sigma = \{\sigma \mid \sigma \models_I \phi\}$.

Next, we assign to each program $S$ its *meaning* $\mathcal{M}_I[\![S]\!]$ relative to $I$, defined by

$$\mathcal{M}_I[\![S]\!](\sigma) = \{\tau \mid \text{there exists a computation of } S \text{ over } I \text{ that starts in } \sigma \text{ and terminates in } \tau\}.$$

At this moment the set $\mathcal{M}_I[\![S]\!](\sigma)$ has at most one element, which will not be anymore the case when nondeterministic or parallel programs are considered.

Then given an assertion $P$ and a program $S$ we define

$$sp_I(P, S) = \{\tau \mid \exists\sigma(\sigma \models_I P \wedge \tau \in \mathcal{M}_I[\![S]\!](\sigma))\}.$$

So $sp_I(P, S)$ is the set of states that can be reached by executing $S$ over $I$ starting in a state satisfying $P$; '*sp*' stands for the *strongest postcondition*.

We then say that the language $\mathcal{L}$ is *expressive* relative to an interpretation $I$ and a class of programs $\mathcal{S}$ if for every assertion $P$ and program $S \in \mathcal{S}$ the set of states $sp_I(P, S)$ is definable. Finally, given a first-order language $\mathcal{L}$, a proof system $PS$ for a class of programs $\mathcal{S}$ is called *complete in the sense of Cook* if for every interpretation $I$ such that $\mathcal{L}$ is expressive relative to $I$ and $\mathcal{S}$ the following holds:

every correctness formula true in $I$ in the sense of partial correctness can be proved in $PS$ assuming all true formulas in $I$.

In other words, completeness in the sense of Cook is a restricted form of relative completeness mentioned above, in which we limit ourselves to the class of interpretations w.r.t. which the underlying language $\mathcal{L}$ is expressive.

The result presented in [Coo78] shows in particular that the proof system $\mathcal{H}$ for partial correctness of **while** programs is complete in the sense of Cook. The main difficulty in the proof, that proceeds by induction on the program structure, consists in finding the loop invariants. A simpler argument was provided in [Cla79], where a dual definition of expressiveness was used. Instead of the strongest postcondition it relied on the so-called *weakest liberal precondition*, which, given an interpretation $I$, assertion $Q$ and a program $S$, is defined by

$$wlp_I(S, Q) = \{\sigma \mid \forall\tau(\tau \in \mathcal{M}_I[\![S]\!](\sigma) \rightarrow \tau \models_I Q)\}.$$

So $wlp_I(S, Q)$ is the set of states from which all terminating computations of $S$ over $I$ end in a state satisfying $Q$. The qualification 'liberal' refers to the fact that termination is not guaranteed. The assumption that the set of states $wlp_I(Q, S)$ is definable makes it possible to find a very simple loop invariant. Namely, assuming that $\{P\}$ **while** $B$ **do** $S$ **od** $\{Q\}$ is true in an interpretation $I$ such that $\mathcal{L}$ is expressive relative to it in this revised sense, it turns out that a loop invariant is simply an assertion $R$ defining $wlp_I(S, Q)$. Additionally, both $P \rightarrow R$ and $R \wedge \neg B \rightarrow Q$ are true in $I$, which allows one to establish $\{P\}$ **while** $B$ **do** $S$ **od** $\{Q\}$ by the WHILE and CONSEQUENCE rules.

In that context it is useful to mention a proposal put forward in [BG87] by A. Blass and Y. Gurevich. They suggested to use a different assertion language than first-order logic (or its multi-sorted variants dealing with subscripted variables or typed variables). The proposed assertion language is a fragment of the second-order logic, called *existential fixed-point logic* (EFL). EFL extends a fragment of first-order logic, in which

negation is applied only to atomic formulas and the universal quantifier is absent, by a fixed-point operator. The authors showed that EFL is sufficient for proving relative completeness of the proof system $\mathcal{H}$ without any expressiveness assumption. The reason is that both the strongest postconditions and the weakest liberal preconditions of the **while** programs (also in presence of recursive parameterless procedures) are definable in EFL.

Consider now the proof system $\mathcal{HT}$ for total correctness. To establish its completeness in the appropriate sense we encounter the same complications as in the case of $\mathcal{H}$, but additionally we have to deal with the problem of definability of the termination functions used in the WHILE II rule. In [Har79] completeness was established for the assertion languages that extend the language of Peano arithmetic and for arithmetic interpretations defined in the previous subsection, but the paper considered the WHILE I rule in which the termination functions are absent. In [AO91] and the subsequent two editions of the book relative completeness of $\mathcal{HT}$ was established. To this end, it was assumed that the underlying assertion language is *expressive*, which meant that for every **while** loop $S$ there exists an integer expression $t$ such that whenever $S$ terminates when started in a state $\sigma$, then the value $\sigma(t)$ is the number of loop iterations. In the adopted setup the assumption that all variables are typed automatically ensures that the considered interpretations included the standard model of Peano arithmetic and that $\sigma(t)$ is a natural number.

# 5 Fine-tuning the Approach

The matters discussed until now gloss over certain issues that have to do with the adjustments of the preconditions and postconditions, various uses of variables, and procedure parameters. In this section we discuss closely these matters, as they reveal some differences between customary logics and Hoare's logic and show the subtleties of reasoning about various uses of variables in the context of programs.

## 5.1 Adaptation rules

In Hoare's logic we see two types of rules. First, for each programming construct there is at least one axiom or rule dealing with its correctness. Together, they make possible a syntax-directed reasoning about program correctness. Second, there are proof rules where the same program $S$ is considered in the premise and the conclusion. These rules allow us to *adapt* an already established correctness formula $\{P_1\}\ S\ \{Q_1\}$ about $S$ to another proof context. Most prominent is the CONSEQUENCE rule that allows us to strengthen the precondition $P_1$ to a precondition $P$ with $P \to P_1$ and to weaken the postcondition $Q_1$ to a postcondition $Q$ with $Q_1 \to Q$, thus arriving at the conclusion $\{P\}\ S\ \{Q\}$. Another one is Hoare's ADAPTATION rule dealing with procedure calls. Hoare stated in [Hoa71a] that in the absence of recursion, i.e., in his proof system for **while** programs, his ADAPTATION rule is a derived rule. So the power of this rule is only noticeable in the context of recursion.

Other rules can be conceived that are concerned with the same program in the premise and conclusion. For example, the following rules were used in various proof systems in the literature. Here and elsewhere we denote the set of variables of a program $S$ by $var(S)$.

INVARIANCE

$$\frac{\{R\}\ S\ \{Q\}}{\{P \wedge R\}\ S\ \{P \wedge Q\}}$$

where $free(P) \cap var(S) = \emptyset$.

∃-INTRODUCTION

$$\frac{\{P\}\ S\ \{Q\}}{\{\exists\, x : P\}\ S\ \{Q\}}$$

where $x \notin var(S) \cup free(Q)$.

SUBSTITUTION I

$$\frac{\{P\}\ S\ \{Q\}}{\{P[\mathbf{z} := \mathbf{t}]\}\ S\ \{Q[\mathbf{z} := \mathbf{t}]\}}$$

where $(\{\mathbf{z}\} \cup var(\mathbf{t})) \cap var(S)) = \emptyset$.

We shall return to these rules shortly. But first, following [Old83b], let us discuss the ADAPTATION rule in the more general setting of programs. We say that a program $S$ is *based on* a finite set $X$ of variables if $var(S) \subseteq X$ holds. Now we can recast Hoare's ADAPTATION rule as follows:

ADAPTATION I

$$\frac{\{P\}\ S\ \{Q\}}{\{\exists\, \mathbf{z}(P \wedge \forall \mathbf{x}(Q \to R))\}\ S\ \{R\}}$$

where $\mathbf{x}$ and $\mathbf{z}$ are lists of variables, $S$ is based on $\{\mathbf{x}\}$, and $\{\mathbf{z}\} = free(P, Q) \setminus (free(R) \cup \{\mathbf{x}\})$.

Following Hoare, the precondition in the conclusion of this rule intends to express the weakest precondition $W$ such that $\{W\}\ S\ \{R\}$ holds (in the sense of partial correctness), assuming that $R$ is the desired result of executing $S$ and $\{P\}\ S\ \{Q\}$ is already established. This intention can be phrased as follows: find the weakest assertion $W$ such that $\{W\}\ S\ \{R\}$ holds for *all* programs based on $\{\mathbf{x}\}$ that satisfy $\{P\}\ S\ \{Q\}$. In [Old83b] this precondition was calculated as follows:

$$W \equiv \forall \mathbf{y}\, (\forall \mathbf{u}\, (P \to Q[\mathbf{x} := \mathbf{y}]) \to R[\mathbf{x} := \mathbf{y}]).$$

Comparing $W$ with the precondition used in the conclusion of the ADAPTATION I rule shows that the implication

$$\exists\, \mathbf{z}(P \wedge \forall \mathbf{x}(Q \to R)) \to W$$

holds but the converse is false. Thus Hoare's precondition is sound but is stronger than necessary. This suggests the following variant of the rule:

ADAPTATION II

$$\frac{\{P\}\ S\ \{Q\}}{\{W\}\ S\ \{R\}}$$

where $W$ is the precondition calculated above and $S, \mathbf{x}, \mathbf{z}$ are as in the ADAPTATION I rule.

To compare the power of different adaptation rules, S. de Gouw and J. Rot [dGR16] used the following notion due to [Kle99]. A set $\mathcal{R}$ of proof rules for a class $\mathcal{S}$ of programs is called *adaptation complete* if for all assertions $P, Q, P', Q'$ and finite sets $X$ of variables

- whenever for all programs $S \in \mathcal{S}$ based on $X$ the truth of $\{P\}\ S\ \{Q\}$ implies the truth of $\{P'\}\ S\ \{Q'\}$ in the sense of partial correctness,

- then for all program $S \in \mathcal{S}$ based on $X$ there is a derivation of $\{P'\}\ S\ \{Q'\}$ from $\{P\}\ S\ \{Q\}$ using only rules of $\mathcal{R}$, written as $\{P\}\ S\ \{Q\} \vdash_{\mathcal{R}} \{P'\}\ S\ \{Q'\}$.

By the result of [Old83b], the set $\mathcal{R}_{\mathrm{O}} = \{\text{ADAPTATION II}, \text{CONSEQUENCE}\}$ is adaptation complete. Further, $\mathcal{R}_{\mathrm{O}}$ enjoys two properties, as noted in [dGR16]:

1. Other adaptation rules, like INVARIANCE, $\exists$-INTRODUCTION, SUBSTITUTION I, are derivable from $\mathcal{R}_{\mathrm{O}}$.
2. Any derivation in $\mathcal{R}_{\mathrm{O}}$ can be replaced by a single application of each of the two rules in $\mathcal{R}_{\mathrm{O}}$.

What about Hoare's adaptation rule? Let $\mathcal{R}_{\mathrm{H}} = \{\text{ADAPTATION I}, \text{CONSEQUENCE}\}$. From a counter-example given in [Old83b] it follows that this set is *not* adaptation complete. Nevertheless, $\mathcal{R}_{\mathrm{H}}$ enjoys property 1, but not property 2 of $\mathcal{R}_{\mathrm{O}}$.

The paper [Old83b] also investigated three other adaptation rules proposed in the literature. An adaptation rule introduced in [GL80] turned out to be sound but *not* adaptation complete when grouped together with the CONSEQUENCE rule. In turn an adaptation rule for the programming language EUCLID given in [LGH+78] is not even sound, while an adaptation rule introduced in [CO81] is both sound and adaptation complete when grouped together with the CONSEQUENCE rule.

## 5.2 Subscripted and local variables

**Subscripted variables** In both [Hoa71b] and [FH71] the ASSIGNMENT axiom was applied to subscripted variables, by implicitly assuming that the definition of substitution is obvious for such variables. This is indeed the case when the subscripts are simple expressions, for example a constant or a simple variable, which was indeed the case for both programs analyzed there. However, in the case of more complex subscripts difficulties may arise, as the following example discussed in [dB80] shows. In the case of an assignment to a simple variable any correctness formula $\{P\}\ x := t\ \{x = t\}$, where $t$ is a constant is true. However, the correctness formula

$$\{a[1] = 2 \wedge a[2] = 2\}\ a[a[2]] := 1\ \{a[a[2]] = 1\}$$

is false. Indeed, given the precondition the execution of the assignment $a[a[2]] := 1$ amounts to executing the assignment $a[2] := 1$ after which the expression $a[a[2]]$ evaluates to 2 and not 1. This suggests that the ASSIGNMENT axiom cannot be used for arbitrary subscripted variables.

This complication was clarified and solved in a systematic way in [dB80], by extending the definition of substitution to an arbitrary subscripted variable. The crucial step in the inductive definition of the substitution $s[u := t]$ deals with the case when $s \equiv a[s_1]$ and $u \equiv a[u_1]$, for which one defines

$$s[u := t] \equiv \textbf{if } s_1[u := t] = u_1 \textbf{ then } t \textbf{ else } a[s_1[u := t]] \textbf{ fi}.$$

So in the **if** case one checks whether after performing the substitution $[u := t]$ on $s_1$ the subscripts $s_1$ and $u_1$ are *aliases* —and substitutes in that case $a[s_1]$ by $t$— while in the **else** case one applies the substitution $[u := t]$ inductively to the subscript $s_1$ of $a[s_1]$.

J.W. de Bakker showed that with this extended definition of substitution the ASSIGNMENT axiom remains sound for subscripted variables. Different axioms for assignment to subscripted variables are given in [HW73, Gri78, Apt81].

**Local variables** Consider now the local variables. They can be viewed as a counterpart of bound variables in logical formulas. However, the situation is more complicated because of the dynamic character of variables in programming languages and the presence of procedures.

We discussed already completeness in the sense of Cook of the proof system $\mathcal{H}$ given in [Coo78]. Cook actually considered an extension of the proof system $\mathcal{H}$ by axioms and proof rules for a small programming language that allows variable declarations and nonrecursive procedures and proved its completeness in the above sense. However, the semantics of the block statement made the corresponding completeness result invalid. It is useful to discuss this matter more closely.

Local variables were already dealt with in the DECLARATION rule mentioned in Subsection 3.2. This rule was slightly adjusted in [Coo78] so that one could reason about variable declarations in the context of non-recursive procedures. But even without this adjustment a possible problem arises. Consider the program

$$\textbf{begin var } x;\ x := 1 \textbf{ end}; \textbf{begin var } y;\ z := y \textbf{ end}.$$

In many programming languages it would yield an error because the right-hand side of the second assignment refers to a value of the uninstantiated variable $y$. However, according to the semantics proposed in [Coo81] such assignments were allowed. Local variables were modelled using a stack in which the last used value was kept on the stack and implicitly assigned to the next local variable. As a result the correctness formula

$$\{\textbf{true}\}\ \textbf{begin var } x;\ x := 1 \textbf{ end}; \textbf{begin var } y;\ z := y \textbf{ end}\ \{y = 1\}$$

was true according to the semantics though there is no way to prove it.

[Coo81] provided a corrigendum in which two possible fixes were suggested. One was to modify the semantics so that the proposed proof system is still complete. This can be achieved by assigning to each newly declared variable a register that has not been used before and modifying the notion of a state accordingly.

Another one was to require all newly declared variables to be initialized to some fixed value, say $\omega$. This option, first used in [Gor75], results in the following rule:

BLOCK

$$\frac{\{P[x := y] \wedge x = \omega\}\ S\ \{Q[x := y]\}}{\{P\}\ \textbf{begin var } x;\ S \textbf{ end}\ \{Q\}}$$

where $y \notin \mathit{free}(P, Q) \cup \mathit{var}(S)$.

To correct the proof of the relative completeness result given in [Coo78] one should then replace the DECLARATION rule by the BLOCK rule. Yet another option is to require all newly declared variables to be explicitly initialized to some arbitrary expression. This approach was taken in [AdBO09], where the following more general version of the corresponding rule was used that allowed a declaration of a list of new variables:

BLOCK I

$$\frac{\{P\}\ \mathbf{x} := \mathbf{t}; S\ \{Q\}}{\{P\}\ \textbf{begin var } \mathbf{x} := \mathbf{t}; S \textbf{ end}\ \{Q\}}$$

where $\{\mathbf{x}\} \cap \mathit{free}(Q) = \emptyset$.

Here $\mathbf{x} := \mathbf{t}$, where $\mathbf{x}$ is a list of different variables and $\mathbf{t}$ a corresponding list of expressions, is a *parallel assignment*, introduced in [Dij75] and further discussed in Section 10.

It is natural to postulate in the BLOCK I rule that the variables listed in $\mathbf{x}$ do not appear in the expressions from $\mathbf{t}$. However, this is a syntactic condition concerning the program formation that is not needed to reason about partial correctness. Further, as we shall soon see, putting no restrictions on $\mathbf{x}$ and $\mathbf{t}$ turns out to be useful for modelling parameter passing in a subtle situation when some formal parameters happen to coincide with the global variables that are used in actual parameters.

An observant reader will notice that in the discussed rules substitution is used differently. In the DECLARATION rule the substitution is applied to the programs, in the BLOCK rule it is applied to the assertions, while —interestingly— in the BLOCK I rule it is not used at all. The resulting proof systems yield different results when applied to programs that use procedures. To illustrate the problem consider the parameterless procedure declared by $\mathbf{proc}\, p : z := x$, the program

$$S_0 \equiv \mathbf{begin\ var}\ x;\ x := 0;\ \mathbf{call}\, p\ \mathbf{end},$$

and the correctness formula

$$\{x = 1\}\ S_0\ \{z = 1\}. \tag{1}$$

To reason about the procedure call $\mathbf{call}\, p$ we add to the proof system $\mathcal{H}$ the following degenerated version of the RECURSION rule:

COPY

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ \mathbf{call}\, p\ \{Q\}}$$

assuming the declaration of a parameterless non-recursive procedure $\mathbf{proc}\, p : S$.

In our case it allows us to derive $\{x = 1\}\ \mathbf{call}\, p\ \{z = 1\}$ from $\{x = 1\}\ z := x\ \{z = 1\}$. This in turn allows us to derive

$$\{x = 1\}\ y := 0;\ \mathbf{call}\, p\ \{z = 1\}.$$

Now, applying the DECLARATION rule we get (1).

However, using the BLOCK rule we get a different conclusion. Namely, we first establish

$$\{y = 1 \wedge x = \omega\}\ x := 0;\ \mathbf{call}\, p\ \{z = 0\},$$

from which

$$\{x = 1\}\ S_0\ \{z = 0\}. \tag{2}$$

follows.

Finally, if we use the BLOCK I rule, and therefore consider a slightly modified program

$$S' \equiv \mathbf{begin\ var}\ x := 0;\ \mathbf{call}\, p\ \mathbf{end},$$

we get $\{x = 1\}\ S'\ \{z = 0\}$.

These differences have to do with the way local variables are interpreted in the presence of procedures. According to the *static* scope the procedures should be evaluated in the environment in which they were declared, while according to the *dynamic* scope they should be evaluated in the environment in which they were called. So according to the static scope, which is adopted in most imperative languages, we should conclude (1) and not (2).

In [AdBO09] static scope is achieved by ensuring that the local variables are first renamed so that they differ from global variables. In the above example one thus considers the statement

$$S_1 \equiv \textbf{begin var } y;\ y := 0;\ \textbf{call } p \textbf{ end}$$

instead of $S_0$. Then we get $\{x = 1\}\ S_1\ \{z = 1\}$, as desired.

## 5.3 Parameter mechanisms and procedure calls

The *call-by-name* parameter mechanism was originally proposed in ALGOL 60. It was used in [Hoa71a] and [Coo78] and adopted in all subsequently discussed papers on procedures, unless stated otherwise. It boils down to a simultaneous substitution of the actual parameters for the formal ones, so it is natural that it was modelled in the SUBSTITUTION rule by a straightforward substitution.

However, a most commonly used parameter mechanism is *call-by-value*. According to its semantics the actual parameters are evaluated first and subsequently their values assigned to the formal parameters. Some other parameter mechanisms were occasionally used. For example, the programming language PASCAL (see [JW75]) also allows the *call-by-variable* mechanism (also called *call-by-reference*), which is a mixture of call-by-name and call-by-value. The actual parameter has to be a variable. In case it is a subscripted variable, its index is evaluated first and the resulting subscripted variable is substituted for the formal parameter.

In [AdB77] it was proposed to model these two parameter mechanisms of PASCAL by means of a 'syntactic application'. In what follows we use in the procedure declaration the qualification **val** to indicate call-by-value and **var** to indicate call-by-variable. Given a procedure declaration $\textbf{proc}\, p(\textbf{val}\ x, \textbf{var}\ y) : S$, so with $x$ called by value and $y$ called by variable, the call $p(t, v)$, where $t$ is an expression and $v$ a, possibly subscripted, variable, was modelled by the program $S[t, v]$ defined by

$$\begin{array}{rcl} S[t, z] & \equiv & \textbf{begin var } u;\ u := t;\ S[x := u, y := z] \textbf{ end}, \\ S[t, a[s]] & \equiv & \textbf{begin var } u_1, u_2;\ u_1 := t;\ u_2 := s;\ S[x := u_1, y := a[u_2]] \textbf{ end}, \end{array}$$

where $z$ is a simple variable and $u, u_1, u_2$ do not appear in $s, t, z$ or $S$.

This naturally leads to the following generalization of the COPY rule from the previous subsection:

CALL-BY-VALUE/CALL-BY-VARIABLE

$$\frac{\{P\}\ S[t, v]\ \{Q\}}{\{P\}\ \textbf{call}\, p(t, v)\ \{Q\}}$$

where the non-recursive procedure $p$ is declared by $\textbf{proc}\, p(\textbf{val}\ x, \textbf{var}\ y) : S$.

In [AdBO09] this approach to call-by-value was slightly simplified by noticing that no variable renaming is needed to model it. The resulting rule, that needs to be used in conjunction with the BLOCK I rule, became:

CALL-BY-VALUE

$$\frac{\{P\}\ \textbf{begin var } \mathbf{x} := \mathbf{t}; S \textbf{ end}\ \{Q\}}{\{P\}\ \textbf{call}\, p(\mathbf{t})\ \{Q\}}$$

where the non-recursive procedure $p$ is declared by $\textbf{proc}\, p(\textbf{val}\ \mathbf{x}) : S$.

To see how this rule correctly handles a subtle situation when a formal parameter coincides with a global variable used in an actual parameter, consider a procedure declared by

$$\mathbf{proc}\, p(\mathbf{val}\ x) : x := x + 2;\ y := y + x.$$

Using the BLOCK I rule we can then establish the correctness formula

$$\{y = 0 \wedge x = 1\}\ \mathbf{begin\ var}\ x;\ x := x + 1;\ x := x + 2;\ y := y + x\ \mathbf{end}\ \{y = 4\},$$

from which

$$\{y = 0 \wedge x = 1\}\ p(x + 1)\ \{y = 4\}$$

follows by the CALL-BY-VALUE rule. This agrees with the semantics of the call-by-value parameter mechanism. (The stronger postcondition, $y = 4 \wedge x = 1$ can be established using the axioms and proof rules introduced in the next section.) So the assignment $x := x + 1$ refers on the left-hand side to the formal parameter $x$ and on the right-hand side to the actual parameter $x + 1$ that contains the global variable $x$.

An obvious drawback of these two proof rules is that each procedure call has to be dealt with separately. It would be preferable if we had to our disposal a counterpart of the SUBSTITUTION rule that would allow us to establish a desired property for a 'generic call' just once, from which the needed properties of all specific calls would follow. In [CO81] it was shown that this can be done under some assumptions that in particular imply that static and dynamic scopes coincide. More recently, in [AdBO09] it was observed that the same be achieved for the calls in which no actual parameter happens to coincide with a global variable.

These two rules can be modified in a natural way to deal with recursive procedures. For the first one we then get the following rule to which we shall return in the next section:

RECURSION I

$$\frac{\{P_i\}\ \mathbf{call}\, p(t_i, v_i)\ \{Q_i\}, i \in \{1, \ldots, n\} \vdash \{P_i\}\ S[t_i, v_i]\ \{Q_i\}, i \in \{1, \ldots, n\}}{\{P_1\}\ \mathbf{call}\, p(t_1, v_1)\ \{Q_1\}}$$

where the procedure $p$ is declared by $\mathbf{proc}\, p(\mathbf{val}\ x, \mathbf{var}\ y) : S$ and $p(t_i, v_i)$, $i \in \{1, \ldots, n\}$, are the procedure calls that appear in $p(t_1, v_1)$ and $S[t_1, v_1]$.

In [Apt81] it was suggested that other parameter mechanisms can be modelled by syntactic application and subsequently reasoned about within Hoare's logic. An example is the *call-by-result* parameter mechanism of ALGOL W, a precursor of PASCAL (see [WH66]). According to it the actual parameter is either a simple or a subscripted variable. Upon termination of the call the value of the formal parameter is assigned to the actual parameter. In the case the actual parameter is a subscripted variable, its index is evaluated first. This parameter mechanism is used in conjunction with the call-by-value.

# 6 Reasoning about Arbitrary Procedures

## 6.1 Completeness results for recursive procedures

**Partial correctness** The relative completeness result established in [Coo78] dealt with the language considered in [Hoa71a] and Subsection 3.2, except that recursion was disallowed. To ensure soundness Cook

stipulated that for the procedure calls $\mathbf{call}\, p(\mathbf{a} : \mathbf{e})$ no variable in $(\mathbf{a} : \mathbf{e})$ different from formal parameters occurs globally in the procedure body.

This result was extended to the language in which recursive procedures were allowed in the Master Thesis of G.A. Gorelick, that was written under the supervision of Cook. The details are only available as a technical report [Gor75]. We present the essentials for the case of a single recursive procedure $\mathbf{proc}\, p(\mathbf{x} : \mathbf{v}) : S$, in line with the presentation in Subsection 3.2.

The conceptual contribution of Gorelick is the introduction of *most general formulas*. He wrote:

> "The completeness result for recursive programs is then obtained by exhibiting, for each recursive procedure $p$, a "most general formula" $\alpha_p$ such that $\vdash \alpha_p$, and $\alpha_p \vdash \beta$ for all true formulas $\beta$ about $p$."

Given a procedure declaration $\mathbf{proc}\, p(\mathbf{x} : \mathbf{v}) : S$, a *most general formula for* the procedure $p$ is a correctness formula

$$\{\mathbf{c} = \mathbf{z}\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{G\},$$

where $\mathbf{c}$ is the list of variables that appear in the formal parameters $\mathbf{x}$ and $\mathbf{v}$ or have a global occurrence in $S$, and $\mathbf{z}$ is a list a fresh variables (not occurring in $\mathbf{x}, \mathbf{v}$, or $S$), of the same length as $\mathbf{c}$ that serves to *freeze* the initial values of the variables in $\mathbf{c}$ before they are changed by $S$. The formula $G$ is taken to express the strongest postcondition $sp_I(\mathbf{c} = \mathbf{z}, S)$ introduced in Subsection 4.2. Since the variables in $\mathbf{c}$ and $\mathbf{z}$ may appear in $G$ as free variables, $G$ describes the relationship between the initial and final values of the variables in $\mathbf{c}$ computed by the procedure body $S$.

The crucial properties of most general formulas are captured by the following lemmas due to [Gor75].

**Lemma G1** If $\{P\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{Q\}$ is true in $I$ in the sense of partial correctness then it can be derived from $\{\mathbf{c} = \mathbf{z}\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{G\}$ and the set of all true formulas in $I$ using "suitable adaptation rules".

**Lemma G2** For each procedure call $\mathbf{call}\, p(\mathbf{c} : \mathbf{v})$ the most general formula $\{\mathbf{c} = \mathbf{z}\}\ \mathbf{call}\, p(\mathbf{x} : \mathbf{v})\ \{G\}$ can be derived from the set of all true formulas in $I$ using Lemma G1 and the RECURSION rule.

The proof of Lemma G1 is based on the following axiom and adaptation rules, proposed in [Gor75] for the case where $S$ is a procedure call $\mathbf{call}\, p(\mathbf{a} : \mathbf{e})$:

INVARIANCE

$$\{P\}\ S\ \{P\}$$

where $\mathit{free}(P) \cap \mathit{change}(S) = \emptyset$.

CONJUNCTION

$$\frac{\{P\}\ S\ \{Q\}, \{P\}\ S\ \{R\}}{\{P\}\ S\ \{Q \wedge R\}}$$

VARIABLE SUBSTITUTION

$$\frac{\{P\}\ S\ \{Q\}}{\{P[\mathbf{z} := \mathbf{t}]\}\ S\ \{Q[\mathbf{z} := \mathbf{t}]\}}$$

where

- $\{\mathbf{z}\} \cap \mathit{var}(S) = \emptyset$,

- if for any component $t_i$ of the list $\mathbf{t}$, $var(t_i) \cap var(S) \neq \emptyset$, then the corresponding component $z_i$ of the list $\mathbf{z}$ satisfies $z_i \notin free(Q)$.

Using Lemmas G1 and G2, Gorelick established the following result.

**Completeness Theorem** For programs with recursive procedures as defined in Subsection 3.2, the proof system $\mathcal{H}$ extended by the RECURSION and SUBSTITUTION rules of Subsection 3.2 and the above axiom and adaptation rules is complete in the sense of Cook.

The restrictions imposed on the actual parameters in the procedure calls were partly taken care of in [CO81]. In [Gor75] call-by-name parameter mechanism was used. An analogous work was carried out for the case of the call-by-value and call-by-variable parameter mechanisms. In [dB80] soundness and relative completeness was proved for a proof system in which the RECURSION I rule was used instead of the RECURSION rule and in which in addition to various rules mentioned so far also a proof rule dealing with the renaming of variables in programs was added. However, the proof was established only for the special case of a single recursive procedure, given the combinatorial explosion of the cases concerned with the relation between the actual and formal parameters. The main ideas of this proof were discussed in [Apt81].

**Total correctness** To deal with total correctness of the recursive procedures the following analogue of the WHILE I rule was proposed independently in [Cla76] and [Sok77]:

RECURSION II

$$\frac{\neg P(0), \{P(n)\}\ \mathbf{call}\, p\ \{Q\} \vdash \{P(n+1)\}\ S\ \{Q\}}{\{\exists n P(n)\}\ \mathbf{call}\, p\ \{Q\}}$$

given the procedure declaration $\mathbf{proc}\, p : S$, and where $P(n)$ is an assertion with a free variable $n$ that does not appear in $S$ and ranges over natural numbers.

In [Apt81] it was stated without a proof that the proof system corresponding to the one used in [Gor75], more precisely the one in which the INVARIANCE axiom is dropped (it is not sound for total correctness), the procedures have no parameters and the RECURSION rule is replaced by the RECURSION II rule, is sound in the sense of total correctness. However, it was discovered in [AdB90a] that this claim is false. The problem has to do with the fact that the counter variable $n$ can be subject to quantifier elimination in the $\exists$-INTRODUCTION rule, and to substitution in the SUBSTITUTION rule.

For example, given the procedure declaration $\mathbf{proc}\, p : p; p$ of an obviously nonterminating procedure one can establish the premises $\neg P(0)$ and $\{P(n)\}\ \mathbf{call}\, p\ \{\mathbf{true}\} \vdash \{P(n+1)\}\ S\ \{\mathbf{true}\}$ of the above rule for $P(n) \equiv n > 1$ and conclude $\{\mathbf{true}\}\ \mathbf{call}\, p\ \{\mathbf{true}\}$ by the RECURSION II and CONSEQUENCES rules.

The solution proposed in [AdB90a] was to stipulate that the counter variables are treated as constants in the $\exists$-INTRODUCTION and SUBSTITUTION rules. This allowed the authors to prove both soundness and relative completeness of the resulting proof system for total correctness of recursive procedures without parameters w.r.t. the arithmetic interpretations introduced in Subsection 4.1.

Having in mind the above complications, in [AdBO09] the following analogue of the WHILE II rule was used for recursive procedures with the call-by-value parameter mechanism:

RECURSION III

$$\frac{\{P \wedge t < z\}\ \mathbf{call}\, p(\mathbf{e})\ \{Q\} \vdash \{P \wedge t = z\}\ \mathbf{begin\ var}\ \mathbf{u} := \mathbf{e};\ S\ \mathbf{end}\ \{Q\} \quad P \to t \geq 0}{\{P\}\ \mathbf{call}\, p(\mathbf{e})\ \{Q\}}$$

given the procedure declaration $\mathbf{proc}\, p(\mathbf{u}) : S$, where $t$ is a termination function, $z$ is an integer variable that does not occur in $P, t, Q$ and $S$ and is treated in the proofs as a constant.

The last restriction means that in these proofs neither the ∃-INTRODUCTION rule nor the SUBSTITUTION rule is applied to $z$.

## 6.2 Clarke's incompleteness result

Programming languages like ALGOL 60 [NBB+63] and PASCAL [JW75] contain procedure mechanisms that are considerably more complex than what we discussed so far. In his seminal paper [Cla79], Edmund M. Clarke identified a combination of features for which it is impossible to obtain a Hoare-like proof system that is sound and relatively complete (in the sense of Cook). Clarke proved this incompleteness result for a block-structured programming language $L_0$ which includes the following features that are present in ALGOL 60 and PASCAL:

1. procedure names as parameters of procedure calls,
2. recursion,
3. static scope,
4. global variables in procedure bodies,
5. nested procedure declarations.

Clarke's incompleteness result is based on the following two lemmas for Hoare-like proof systems $H$ and programming languages $L$.

**Lemma A** If $H$ is sound and relatively complete for $L$, then the divergence problem for $L$ is decidable for finite interpretations $I$.

**Lemma B** If $L$ has a rich procedure concept including features 1–5, then its divergence problem is undecidable for all finite interpretations $I$ with at least two domain elements.

An interpretation $I$ is called *finite* if its domain $D$ is finite. A program $S$ in $L$ *diverges* for $I$ if $S$ never terminates when started with any input values from $D$ for its variables. The *divergence problem* for a programming language $L$ for an interpretation $I$ is the problem of deciding whether an arbitrary program $S$ of $L$ *diverges*.

The proof of Lemma A rests on the following general observations. A program $S$ of $L$ diverges for $I$ if and only if the correctness formula $\{true\}\ S\ \{false\}$ is true in $I$ in the sense of partial correctness. Since $H$ is sound and relatively complete, the latter is true if and only if $\{true\}\ S\ \{false\}$ is provable in $H$, when

all assertions in the proofs are interpreted in $I$. Thus all diverging programs $S$ in $L$ can be recursively enumerated by enumerating all proofs in $H$, thereby deciding in $I$ the assertions used in the applications of the CONSEQUENCE rule, which is possible because $I$ is finite. Trivially, also all non-diverging programs $S$ in $L$ can be recursively enumerated: simply start each program $S$ for all of its finitely many inputs from $I$ and enumerate it in case it halts for one of these inputs. Decidability of the set of all diverging programs follows since both the set and its complement are recursively enumerable.

For the proof of Lemma B, Clarke shows that *queue machines* (or *Post machines*), which have an undecidable halting problem, see, e.g., [Man74], can be simulated by programs from $L_0$. A queue machine manipulates a queue and finitely many registers. A machine program is a finite sequence of labelled instructions of three types:

- `enqueue` $x$ that adds the value of register $x$ to the rear of the queue,
- `dequeue` $x$ that removes the front entry from the queue and puts into register $x$, and
- `if` x=y `then go to` $\ell$ that branches to the instruction labelled with $\ell$ if the values of the registers $x$ and $y$ agree.

Clarke [Cla79] described his simulation idea of a queue machine with a program in $L$ as follows:

> "The queue is represented by successive activations of a recursive procedure *sim* with the queue entries being maintained as values of the variable *top* which is local to *sim*. Thus an addition to the rear of the queue may be accomplished by having *sim* call itself recursively. Deletions from the front of the queue are more complicated. *sim* also contains a local procedure *up* which is passed as a parameter during the recursive call which takes place when an entry is added to the rear of the queue. In deleting an entry from the front of the queue, this parameter is used to return control to previous activations of *sim* and inspect the values of *top* local to those activations. The first entry in the queue will be indicated by marking (e.g. negating) the appropriate copy of *top*."

From this description it is clear that the simulation program exploits the features 1, 2 and 5. Feature 4 is also needed since a global variable *program_counter* is used in the body of the procedure *sim*. Feature 3, so static scope, concerns the semantics of procedures and is needed to achieve the correct back pointers for the procedure *up* in the runtime stack generated by the successive activations of the procedure *sim*.

Note that the procedure *sim* has a formal parameter that is instantiated with procedures without own formal procedure parameters. Thus for Clarke's simulation, procedures of *mode depth* $\leq 2$ suffice. This restriction is obeyed in PASCAL. Procedures of arbitrary finite mode depth, i.e. of arbitrary *higher type*, as in ALGOL 68 [vWMP+75], or even with *self-application*, i.e. with procedure calls of the form **call**$p(\ldots, p, \ldots)$, as possible in ALGOL 60 [NBB+63], are not needed.

## 6.3 Clarke's language $L_4$

In [Cla79], Clarke also claimed that each language $L_i$ obtained from $L_0$ by disallowing self-application and the feature $i \in \{1, 2, 3, 4, 5\}$ of the above list has a sound and relatively complete Hoare-like proof system. However, he proved this only for the case of $L_3$ in which dynamic scope replaces static scope. The completeness argument rests on the fact that each program $S$ in $L_3$ has a finite range, where the *range* of

$S$ is the set of different procedure calls that can be invoked in the computations of $S$. He claimed that a similar proof system can be obtained for $L_4$ in which global variables are disallowed. His argument was that programs of $L_4$ can be transformed into schematically equivalent ones in $L_5$, where nested procedure declarations are disallowed. Such programs have a sound and relatively complete Hoare-like proof system.

H. Langmaack and E.R. Olderog [LO80] discovered that this argument is wrong. They considered the *formal execution trees* of programs and showed that programs in $L_4$ may have trees with context-free path languages whereas programs in $L_5$ can generate only trees with regular path languages. Therefore they posed the challenge to develop a sound and relatively complete Hoare-like proof system for $L_4$.

As a first step, Langmaack proved in [Lan82] that for all ALGOL-like programs in $L_4$ the divergence problem is decidable for finite interpretations. This is a necessary condition for the existence of a sound and relatively complete Hoare-like system according to Lemma A. Moreover, due to a theorem by R.J. Lipton [Lip77], this decidability result is also sufficient for the existence of a sound and relatively complete 'Hoare logic' for ALGOL-like programs in $L_4$. However, Lipton's notion of a 'Hoare logic' is rather weak: it means that the set of correctness formulas that are true in the sense of partial correctness is recursively enumerable relative to the underlying interpretation. Lipton's theorem does not yield any usable, syntax-oriented proof rules.

First concrete axiomatizations for $L_4$ programs appeared independently in papers by Olderog [Old84] and W. Damm and B. Josko [DJ83]. The paper [Old84] studied the case of PASCAL-like programs, i.e. with mode depth $\leq 2$. The key idea is the use of second-order relation variables in the assertion language to stand for the behaviour of uninterpeted, symbolic procedures in a new SEPARATION rule. In order to determine what a procedure call $\mathbf{call}\, p(\ldots q \ldots)$ of a procedure $p$ with an actual procedure parameter $q$ does, this rule *first* determines separately what $q$ does and what $p$ does with a symbolic procedure instead of $q$ represented by a formula and *then* composes both results using substitution. Applications of the SEPARATION rule have the effect of a systematic transformation of an initially non-regular formal execution tree into a regular tree at the cost of introducing higher-order tree combinators. The resulting regularity enables a suitably complete Hoare-like proof system.

The paper [DJ83] handled the more general case of ALGOL-like programs with arbitrary finite modes by using higher-order predicate variables and unevaluated substitutions to such variables. Common to both papers is that they deviated from the standard notion of relative completeness in that they used a higher-oder assertion language and an appropriate notion of expressiveness.

The status of the language $L_4$ was finally clarified in 1989 in an over 90 pages long paper by S.M. German, E.M. Clarke and J.Y. Halpern [GCH89] in which a Hoare-like proof system for $L_4$ was provided that is sound and relatively complete in the (original) sense of Cook. The completeness proof heavily relies on an appropriate arithmetic encoding of procedure declarations so that the assertion language remains first-order.

## 6.4 The characterization problem

Exploring the border between relative completeness and incompleteness (in the sense of Cook) of Hoare's logics has been the focus of considerable research in the 1980s. Clarke called it the "Characterization Problem for Hoare Logics" in his survey article [Cla85]. The key question is what is meant by a 'Hoare logic'. Several researchers proved results for a weak interpretation where a Hoare logic is just the set of correctness formulas $\{P\}\ S\ \{Q\}$ that are true in the sense of partial correctness. Here the problem is for which classes of programs $S$, assertions $P, Q$, and underlying interpretations $I$ this set is recursively enumerable or even decidable.

We cited already Lipton's result [Lip77]. Since its proof was only sketched, Langmaack provided in

[Lan79] a rigorous proof for the setting of ALGOL-like programs $S$. Both Lipton and Langmaack restrict themselves to quantifier-free assertions $P, Q$. Clarke, German and Halpern established [CGH83] an "effective axiomatization of Hoare logics" extending Lipton's result to first-order formulas $P, Q$. However, these results do not yield any usable, syntax-directed proof rules that are in the spirit of Hoare-like proof systems that appeared since the publication of [Hoa69]. Therefore Clarke stated in [Cla85]:

> "Certainly the most important research problem is to develop a version of the characterization theorem that provides some insight as to when a syntax-directed proof system can be obtained."

Two such characterization theorems were established in [Old81] and [Old83a]. The paper [Old81] studied a language $L_{Algol}$ of ALGOL-like programs with the features 1, 2, 4 and 5 of [Cla79] (see Subsection 6.2). Feature 3 (static scope) was not fixed but left as a parameter in the form of a *copy rule*. Such a rule defines how during program execution a procedure call is replaced by a copy of the procedure body. In the copy certain occurrences of local variables and procedures in the procedure body are renamed to avoid name clashes. By varying the copy rule, different program semantics varying from dynamic to static scope were defined. The COPY rule from Subsection 5.2 is a trivial example of such a rule in which no renaming takes place.

Parameterized by a given copy rule $\mathcal{C}$, a Hoare-like proof system $\mathcal{H}(\mathcal{C})$ for the partial correctness of ALGOL-like programs was introduced. A program $S$ was called *$\mathcal{C}$-bounded* if applications of the copy rule $\mathcal{C}$ do not lead to programs with "procedural reference chains" of arbitrary length. A program $S$ has a *finite $\mathcal{C}$-index* if the relation of "substitutional equivalence" induces only finitely many equivalance classes in the set of reachable procedure calls. Due to this equivalence, the condition of finite *$\mathcal{C}$-index* is more liberal than the conditions of *finite recursive cycle* and *finite range* used by Gorelick [Gor75] and Clarke [Cla79], respectively, in their completeness proofs. The following result holds.

**Theorem** Suppose the assertion language be expressive relative to the considered interpretation $I$ and the copy rule $\mathcal{C}$. Then the following are equivalent for ALGOL-like programs $S$ and assertions $P, Q$:

1. $\{P\}\ S\ \{Q\}$ can be proved in the proof system $\mathcal{H}(\mathcal{C})$ assuming all true formulas in $I$.
2. $S$ has a finite $\mathcal{C}$-index and $\{P\}\ S\ \{Q\}$ is true in $I$ in the sense of partial correctness.
3. $S$ is $\mathcal{C}$-bounded and $\{P\}\ S\ \{Q\}$ is true in $I$ in the sense of partial correctness.

As corollaries to this theorem various completeness results were obtained in [Old81]. For the naive copy rule $\mathcal{C}_n$, which models "dynamic scoping", the proof system $\mathcal{H}(\mathcal{C}_n)$ is sound and relatively complete for the full set $L_{Algol}$ because all programs in $L_{Algol}$ are $\mathcal{C}_n$-bounded. This corresponds to Clarke's language $L_3$. For the Algol 60 copy rule $\mathcal{C}_{60}$, which models "static scoping", the proof system $\mathcal{H}(\mathcal{C}_{60})$ is sound and relatively complete for the following sublanguages of $L_{Algol}$ because all programs in these sublanguages are $\mathcal{C}_{60}$-bounded:

- $L_{pnes}$ – all programs without procedure nesting, corresponding to Clarke's language $L_5$,
- $L_{par}$ – all programs with parameterless procedures only,
- $L_{pp}$ – all programs without procedures as parameters, corresponding to Clarke's language $L_1$,

- $L_{rp}$ – all programs in which formally recursive procedures are disallowed to have procedure identifiers as formal parameters, a superset of Clarke's language $L_2$,
- $L_{gf}$ – all programs without global formal procedure parameters.

The paper [Old83a] studied a set $L_{Pas}$ of programs with PASCAL-like procedures $S$, i.e., with mode-depth $\leq 2$. Here a sublanguage $L \subseteq L_{Pas}$ is called *admissible* if it is closed under certain program transformations that leave the procedure structure invariant. In particular, these transformations allow for the introduction of global variables. A tree $T$ is *regular* if the set of paths in $T$ is a regular language in the Chomsky hierarchy or, equivalently, if $T$ has only finitely many different patterns of subtrees. The *formal call tree* of a program $S$ describes in which order the procedures of $S$ are called, where a branching appears when from one procedure several immediate successors can be called. The system $H_0$ is a *specific* syntax-directed proof system.

**Theorem** For every admissible language $L \subseteq L_{Pas}$ the following are equivalent:

1. There exists a sound and relatively complete Hoare logic in the sense of Lipton for $L$.
2. The divergence problem for $L$ is decidable for finite interpretations.
3. All programs in $L$ have regular formal call trees.
4. The Hoare-like proof system $H_0$ is sound and relatively complete for $L$.

Since $L_{Pas}$ contains programs with a non-regular formal call tree, the theorem implies that there is no sound and relatively complete Hoare logic for $L_{Pas}$ itself, a fact that follows from Clarke's incompleteness result. Note that this theorem does not contradict the results obtained for $L_4$ because a sublanguage $L \subseteq L_{Pas}$ without global variables is not admissible.

# 7 Nondeterministic and Probabilistic Programs

## 7.1 Reasoning about nondeterminism

In the context of programming languages *nondeterminism* stands for the phenomenon that a program can yield more than one answer. In the sixties and early seventies a couple of simple programming constructs were proposed that introduced nondeterminism. In particular, in [Lau71] the nondeterministic statement $S_1$ **or** $S_2$ was considered with the meaning: execute either $S_1$ or $S_2$, and the following proof rule:

OR

$$\frac{\{P\}\ S_1\ \{Q\}, \{P\}\ S_2\ \{Q\}}{\{P\}\ S_1\ \textbf{or}\ S_2\ \{Q\}}$$

But the undoubtedly most successful and elegant proposal is the language of *guarded commands* introduced by Edsger W. Dijkstra in [Dij75] and in the book form in [Dij76a]. Dijkstra's original motivation for introducing this language was to simplify programs by delaying some arbitrary choices to the implementation level and to restore symmetry that is not present in the **if-then-else** statement. This naturally led to a specific proposal for nondeterminism.

For the sake of the subsequent discussion it is sufficient to consider two crucial statements of Dijkstra's language:

- the *alternative command*

$$\textbf{if } B_1 \to S_1 \Box \ldots \Box B_n \to S_n \textbf{ fi},$$

- *repetitive command*

$$\textbf{do } B_1 \to S_1 \Box \ldots \Box B_n \to S_n \textbf{ od},$$

where each $B_i$ is a Boolean expression, called a *guard*, and each $S_i$ is a program statement. The symbol $\Box$ represents a nondeterministic choice.

The alternative command is executed by selecting a guard $B_i$ that evaluates to **true** and executing the associated statement $S_i$. If more than one guard $B_i$ evaluates to true any of the corresponding statements $S_i$ may be executed next. If all guards evaluate to **false**, the execution of the alternative command results in a *failure*. The repetitive command is executed in a similar way, with two differences. First, after termination of a selected statement $S_i$ the repetitive command is executed again. Second, if all guards evaluate to **false**, the execution of the repetitive command simply terminates. So it is a natural generalization of the **while** statement.

As an illustration of the use of the alternative command consider the customary program for computing the maximum of two numbers using the conditional statement:

**if** $x \geq y$ **then** $max := x$ **else** $max := y$ **fi**.

A solution using the alternative command is symmetric in the variables $x$ and $y$ and also involves nondeterminism:

$$\textbf{if } x \geq y \to max := x \ \Box \ y \geq x \to max := y \textbf{ fi}.$$

As an illustration of the use of the repetitive command consider the customary **while** program for computing the *greatest common divisor* (*gcd*) of two natural numbers, initially stored in the variables $x$ and $y$:

**while** $x \neq y$ **do**
    **if** $x > y$ **then** $x := x - y$ **else** $y := y - x$ **fi**
**od**.

Using the repetitive command the same algorithm can be written as

$$\textbf{do } x > y \to x := x - y \ \Box \ y > x \to y := y - x \textbf{ od}.$$

Both programs terminate with the *gcd* stored in the variables $x$ and $y$ but the second program is more readable and, unlike the first one, is symmetric in the variables $x$ and $y$.

To reason about the guarded command language Dijkstra introduced in [Dij75] the *weakest precondition calculus* that we briefly discuss in Section 10. But it is easy to conceive the Hoare style proof rules that deal with partial correctness of the alternative and repetitive commands (they were proposed first in [dB80]), though care has be exercised to deal with failures, which is a new concept in this framework. The appropriate rule, introduced in [Apt84], takes the following form:

ALTERNATIVE COMMAND II

$$\frac{P \to \bigvee_{i=1}^{n} B_i, \quad \{P \wedge B_i\}\ S_i\ \{Q\}, i \in \{1, \ldots, n\}}{\{P\}\ \textbf{if } \Box_{i=1}^{n}\ B_i \to S_i \textbf{ fi}\ \{Q\}}$$

In turn, termination of the repetitive command is taken care by the following natural generalization of the WHILE II rule, used in [AdBO09]:

REPETITIVE COMMAND II

$$\frac{\begin{array}{l}\{P \wedge B_i\}\ S_i\ \{P\}, i \in \{1,\ldots,n\},\\ \{P \wedge B_i \wedge t = z\}\ S_i\ \{t < z\}, i \in \{1,\ldots,n\},\\ P \to t \geq 0\end{array}}{\{P\}\ \mathbf{do}\ \Box_{i=1}^{n}\ B_i \to S_i\ \mathbf{od}\ \{P \wedge \bigwedge_{i=1}^{n} \neg B_i\}}$$

where $t$ is a termination function and $z$ is an integer variable not occurring in $P, t, B_i$ or $S_i$ for $i \in \{1,\ldots,n\}$.

This way we obtain a proof system for total correctness of guarded commands.

## 7.2 Reasoning about fairness

*Fairness* is a concept that arises in presence of any form of nondeterministic choice. Suppose that we repeatedly have some choice among a fixed set of alternatives, for instance of going left or going right. If we repeatedly select the alternative of 'going left', then we systematically ignore the other alternative. In such case we can argue that the adopted selection procedure is not fair with respect to the other alternative, 'going right'. To exclude such unfair selection procedures we need to focus on infinite 'runs' of selections of alternatives and make precise when an alternative can be selected.

These matters can be discussed in a more precise way using the guarded commands language. Consider the following program, where $k$ is a fixed natural number:

$$x := 1;\ \mathbf{do}\ x > 0 \to x := x + 1\ \Box\ x > k \to x := 0\ \mathbf{od}.$$

It does not always terminate, since we can repeatedly select the first guard. The resulting computation is considered *unfair* since from some moment on the second guard is always enabled (i.e., evaluates to **true**), but never selected. Once the second guard is selected when it is enabled, the program terminates. More formally, we say that the above program terminates under the *fairness* assumption, which stipulates that each guard that is from some moment on continuously enabled is infinitely often selected.

As another example consider the following program, where $odd(x)$ is a test with the expected meaning:

$$x := 1;\ \mathbf{do}\ x > 0 \to x := x + 1\ \Box\ odd(x) \to x := 0\ \mathbf{od}.$$

Also this program does not always terminate. The only infinite computation repeatedly ignores the second guard. However, in contrast to the previous example, in this infinite computation at no moment the second guard becomes continuously enabled. So even under the fairness assumption this program does not terminate.

On the other hand in this infinite computation the second guard is infinitely often enabled. We say that this computation is *strongly unfair*. If the second guard is selected when it is enabled, the program terminates. In this case we say that the above program terminates under the *strong fairness* assumption, which stipulates that each guard that is infinitely often enabled is infinitely often selected. (To stress the difference the first notion of fairness is usually called *weak fairness*.)

Both forms of fair termination can be established by means appropriate proof rules. In what follows we explain the *transformational approach* proposed in [AO81] and in a journal form in [AO83]. To prove

termination of a repetitive command $S$ under the assumption of weak (or strong) fairness (with respect to a precondition $P$) one can transform it into a repetitive command $T(S)$ the computations of which coincide with the weakly (or strongly) fair computations of $S$. To this end, we need a *random assignment* $x :=?$ that assigns nondeterministically to the variable $x$ an arbitrary natural number. To understand why this command naturally arises when considering fairness, note that under both assumptions of fairness the program

$$b := \mathbf{true};\ x := 0;\ \mathbf{do}\ b \to x := x + 1 \ \Box\ b \to b := \mathbf{false}\ \mathbf{od}$$

mentioned in [Dij76a] is equivalent to

$$b := \mathbf{false};\ x :=?.$$

In what follows we limit ourselves to the presentation of strong fairness and to explain the idea we focus on a repetitive command with just two guards:

$$S \equiv \mathbf{do}\ B_1 \to S_1 \ \Box\ B_2 \to S_2\ \mathbf{od}.$$

The transformed program $T(S)$ uses two auxiliary variables $z_1$ and $z_2$ ranging over natural numbers and has the following form:

$$\begin{array}{l} T(S) \equiv z_1 :=?;\ z_2 :=?; \\ \quad \mathbf{do} \\ \quad\quad B_1 \wedge z_1 \leq z_2 \to S_1;\ z_1 :=?;\ \mathbf{if}\ B_2\ \mathbf{then}\ z_2 := z_2 - 1\mathbf{fi} \\ \quad \Box \\ \quad\quad B_2 \wedge z_2 \leq z_1 \to S_2;\ z_2 :=?;\ \mathbf{if}\ B_1\ \mathbf{then}\ z_1 := z_1 - 1\mathbf{fi} \\ \quad \mathbf{od}. \end{array}$$

Informally, each variable $z_i$ tracks the number of times the corresponding guard is enabled and the augmented guards prevent that an infinitely often enabled guard is never selected.

To reason about the strong fair termination of $S$ it is now equivalent to reason about the termination of the program $T(S)$. To this end, we only need an axiom dealing with the random assignment. Such an axiom was proposed in [Har79]:

RANDOM ASSIGNMENT

$$\{P\}\ x :=?\ \{P\}$$

where $x$ does not appear free in $P$.

But this indirect approach can be avoided by absorbing the transformation $T(\cdot)$ into the proof of termination of the program $T(S)$. This way one obtains a proof rule for establishing termination under the strong fairness assumption that deals directly with the original program $S$ and is similar in shape to the REPETITIVE COMMAND II rule. We omit the details, though need to mention that to reason about fairness it is in general necessary that the termination function takes values from an arbitrary well-founded ordering and not just natural numbers (see [AO83], [APS84], and [AP86], where soundness and relative completeness of a proof system for total correctness of **while** programs augmented with the random assignment was established.).

Different approaches to reason about fairness in the context of nondeterministic programs were independently proposed in [LPS81] and [GFMdR81], that appeared in a journal form as [GFMdR85]. In [APS84] the

method presented here was extended to programs with nested nondeterminism. These approaches and proof rules were discussed in a book form in [Fra86], where also additional versions of fairness were considered.

The transformational approach to fairness was originally developed in [AO81, AO83] for the Dijkstra's guarded command language in which each repetitive command has a fixed finite number of alternatives. The transformations can be seen as implementing a general fair scheduler controlling finitely many processes. These transformations were extended in [OP10, HOP10] to deal with *dynamic control*, where processes can be created dynamically. Then the overall number of processes can be inifinite, but at each step of an execution of the system the number of created processes is finite. In [HP15], J. Hoenicke and A. Podelski go one step further and extend the transformations to deal with fairness with an *infinitary control*, where the number of created processes can be infinite. Both dynamic and inifinitary control were expressed by repetitive commands with infinitely many alternatives. However, these papers did not propose any proof rules derived from the new transformations.

## 7.3 Probabilistic programs

Probabilistic programs are sequential programs with the ability to draw values at random from probabilistic distributions. They have attracted large attention in the research community due to many applications, for example in security to describe randomized encryptions, in machine learning to describe distribution functions, and in randomized algorithms. They have typically just a few lines of code, but are hard to analyze, see, e.g., [KGJ$^+$15]. Properties of interest for such programs include the *expected runtime* and *almost sure termination*.

Most formal modelling takes place in the setting of an extended Dijkstra's guarded command language, called *probabilistic guarded command language*, abbreviated pGCL. Here, both nondeterministic choice and probabilistic choice are admitted. In particular, McIver and Morgan [MM05] carried out their research on probabilistic programs in this setting. They extended the notion of weakest precondition to *weakest pre-expectations*.

J. den Hartog and E.P. de Vink [dHdV02] introduced a Hoare-like proof system for partial correctness of probabilistic programs which are defined as (determininistic) **while** programs extended by the *probabilistic choice* $S_1 \oplus_\rho S_2$ between the statements $S_1$ and $S_2$. This intention is that the statement $S_1$ is chosen with the probability $\rho$ and the statement $S_2$ with the probability $1 - \rho$. This necessitates the introduction of probabilistic predicates in the assertion language. In these predicates, the real-valued expression $\mathbb{P}(R)$ yields the probability that the normal predicate $R$ holds. For example, for an integer variable $x$ the correctness formula

$$\{x = 1\}\ x := x + 1 \oplus_{\frac{1}{3}} x := x + 2\ \{\mathbb{P}(x = 2) = \tfrac{1}{3} \wedge \mathbb{P}(x = 3) = \tfrac{2}{3}\}$$

holds. The proof system of [dHdV02] contains several new rules that go beyond the ones concerning the **while** programs. In particular, for the probabilistic choice the following rule was proposed:

PROBABILISTIC CHOICE

$$\frac{\{P\}\ S_1\ \{Q\}, \{P\}\ S_2\ \{Q'\}}{\{P\}\ S_1 \oplus_\rho S_2\ \{Q \oplus_\rho Q'\}}$$

where the probabilistic choice operator $\oplus_\rho$ is also applied to the probabilistic predicates $Q$ and $Q'$.

Den Hartog and de Vink established soundness of their proof system. They also proved relative completeness for the subset of loop-free probabilistic programs and a restricted set of predicates in the postcondition.

The proof is based on calculating the weakest precondition.

Some other developments concerning verification of probabilistic programs are discussed in Subsection 10.6.

# 8 Parallel and Distributed Programs

By a *concurrent program* we mean a program that has a number of components, the execution of which proceeds in parallel. If the program components share some variables one usually, at least in the context of program correctness and analysis, refers to *parallel programs*. If the program components do not share variables but can communicate by messages, one usually calls them *processes* and refers to the concurrent programs as *distributed programs*.

## 8.1 Reasoning about parallel programs

Verification of parallel programs calls for new insights. To discuss the matters let us first introduce the syntax. By

$$[S_1 \| \ldots \| S_n]$$

we mean a *parallel composition* of $n$ sequential programs, $S_1, \ldots, S_n$, that may share variables.

The main complication in reasoning about parallel programs is the interference caused by the use of shared variables. For example, both

$$\{x = 0\}\ [x := x + 1;\ x := x + 1\ \{x = 2\}$$

and

$$\{x = 0\}\ x := 2\ \{x = 2\}$$

hold but

$$\{x = 0\}\ [x := x + 1;\ x := x + 1 \| x := 2]\ \{x = 2 \vee x = 4\}$$

does not since one possible computation consists of executing the assignments in the following order $x := x + 1;\ x := 2;\ x := x + 1$, which yields the final value 3.

The first limited approach to the verification of parallel programs was proposed by Hoare in [Hoa72c], where he dealt with parallel composition of disjoint program components. More formally, we say that the component programs $S_1, \ldots, S_n$ of $[S_1 \| \ldots \| S_n]$ are *disjoint* if no variable subject to change in one component appears in another component. This led to the following proof rule.

DISJOINT PARALLELISM

$$\frac{\{P_i\}\ S_i\ \{Q_i\}, i \in \{1, \ldots, n\}}{\{\bigwedge_{i=1}^{n}\ P_i\}\ [S_1 \| \ldots \| S_n]\ \{\bigwedge_{i=1}^{n}\ Q_i\}}$$

where $\mathit{free}(P_i, Q_i) \cap \mathit{change}(S_j) = \emptyset$ for $i \neq j$.

As noticed by Hoare in [Hoa75] under the assumption of disjointness parallel and sequential compositions of program components coincide. This brought him to suggest the following rule.

SEQUENTIALIZATION

$$\frac{\{P\}\ S_1;\ \ldots;\ S_n\ \{Q\}}{\{P\}\ [S_1 \| \ldots \| S_n]\ \{Q\}}$$

(In [Hoa72c] and [Hoa75] actually only parallel composition of two components was studied). In 1975 this limited approach to program correctness was extended to arbitrary parallel programs. It was based on a novel idea of *interference freedom*, first introduced in the PhD thesis of S. Owicki [Owi75], and subsequently published in two papers by her and her PhD supervisor D. Gries, [OG76a] and [OG76b]. This approach became known as the *Owicki-Gries* method.

In what follows we explain the ideas behind this extension of Hoare's logic. To deal with the problem of interference mentioned above, Owicki and Gries suggested to compose not the correctness statements about the component programs *but* their proofs, presented in an appropriate form. This form relies on the fact that the correctness proofs of the component programs are syntax-directed, that is they follow in some sense the program structure. As a result the proof can be recorded by retaining the used assertions in the program text. For example, the application of the WHILE rule can be retained in the text of the **while** statement by writing its conclusion as

$$\{P\}\ \textbf{while}\ B\ \textbf{do}\ \{P \wedge B\}\ S\ \{P\}\ \textbf{od}\ \{P \wedge \neg B\},$$

while the conclusion of the CONSEQUENCE rule can be written as

$$\{P\}\{P_1\}\ S\ \{Q_1\}\{Q\},$$

which amounts to interpreting two consecutive assertions as an implication. The crucial point of such a proof representation, called a *proof outline*, is that each statement $S$ is preceded in it by a single assertion $pre(S)$.

Here is for a example a proof outline of the correctness proof of the `DIV` program that we considered in Subsection 3.1:

$\{\textbf{true}\}$
$\{x = x + y \cdot 0\}$
$r := x;$
$\{x = r + y \cdot 0\}$
$q := 0;$
$\{P\}$
**while** $y \leq r$ **do**
  $\{P \wedge y \leq r\}$
  $\{x = (r - y) + y \cdot (1 + q)\}$
  $r := r - y;$
  $\{x = r + y \cdot (1 + q)\}$
  $q := 1 + q$
  $\{P\}$
**od**
$\{\neg y \leq r \wedge P\},$

where

$$P \equiv x = r + y \cdot q.$$

In the Appendix we give another example of a proof outline by representing in such a form Turing's proof discussed in Subsection 2.1. To understand the essence of the approach of Owicki and Gries let us return for a moment to the original Hoare's proof system. It is sound in the following stronger sense (for a proof see, e.g., [AdBO09]).

**Strong Soundness Theorem** Consider a proof outline that corresponds with the correctness formula $\{P\}\ S\ \{Q\}$. Take a computation $\xi$ of $S$ that starts in a state that satisfies $P$. Each time $\xi$ reaches a substatement $T$ of $S$, its precondition $pre(T)$ is satisfied. Further, if $\xi$ terminates, its final state satisfies $Q$.

Armed with this knowledge let us return to parallel programs. Suppose that we established the correctness formulas $\{P_i\}\ S_i\ \{Q_i\}$, where $i \in \{1, \ldots, n\}$ and $S_1, \ldots, S_n$ are the component programs of the parallel program $[S_1 \| \ldots \| S_n]$. Let $\{P_i\}\ S_i^*\ \{Q_i\}$ be the corresponding proof outlines. We call them *interference free* if for all assignments $x := s$ in $[S_1 \| \ldots \| S_n]$ and all assertions $R$ used in a proof outline of another component the correctness formula

$$\{R \wedge pre(x := s)\}\ x := s\ \{R\}$$

holds.

Informally, the proof outlines are interference free if the execution of each assignment statement in the state that satisfies its assertion does not invalidate the assertions used in the proof outlines of other components. Then the following proof rule allows us to reason about parallel programs.

PARALLELISM

$$\frac{\text{The proof outlines } \{P_i\}\ S_i^*\ \{Q_i\}, i \in \{1, \ldots, n\}, \text{ are interference free}}{\{\bigwedge_{i=1}^n\ P_i\}\ [S_1 \| \ldots \| S_n]\ \{\bigwedge_{i=1}^n Q_i\}}$$

Owicki and Gries noted that to reason about parallel programs they needed *auxiliary variables*, so variables that neither influence the control flow nor the data flow of the program, but only record some additional information about the program execution. Formally, a set of simple variables $A$ is called a *set of auxiliary variables* of $S$ if each variable from $A$ occurs in $S$ only in assignments to the variables from $A$. The appropriate proof rule allowing one to delete them is

AUXILIARY VARIABLES

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ S_A\ \{Q\}}$$

where for some set of auxiliary variables $A$ of $S$ the program $S_A$ results from $S$ by deleting all assignments to the variables in $A$, and no variable from $A$ appears free in $Q$.

To allow synchronization between the component programs Owicki and Gries used an *await statement* **await** $B$ **then** $S$ **end** with the following meaning: if $B$ evaluates to true, then the statement $S$ is executed without any interruption by other component programs. The corresponding proof rule is

AWAIT

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \textbf{end}\ \{Q\}}$$

The presence of synchronization statements leads to a possibility of a *deadlock*, an undesired situation in which some components of a parallel program did not terminate while the nonterminated components are all blocked. Owicki and Gries noted that interference free outlines allow one to reason about absence of deadlock in a natural way. To this end, it suffices to identify the possible deadlock situations and to show for each of them that the corresponding conjunction of the $pre(T)$ assertions cannot be simultaneously satisfied.

This approach to verification was successfully applied in [OG76a] to establish correctness of non-trivial parallel programs, in particular two versions of the classical producer/consumer problem, a parallel version of the `FIND` program mentioned in Subsection 3.1, and an implementation of Dijkstra's semaphores. In [OG76b] it was extended to parallel programs that use a more efficient synchronization construct called conditional regions introduced in [Hoa72a], and in [Owi78] to concurrent programs with shared data classes

**Termination** Let us discuss now the issue of termination of parallel programs. Owicki and Gries proposed to establish it by using the WHILE II rule instead of the WHILE rule and by postulating that no component program increases the termination function of a **while** statement used in another component program. However, the authors of [AdBO90] noticed that the termination of parallel programs is subtler than it sounds and came up with an example of simple parallel program that does not terminate, even though its termination can be established using Owicki and Gries method.

The problem has to do with the fact that the assertions used in the proof of the second premise of the WHILE II rule, so $\{P \wedge B \wedge t = z\}\ S\ \{t < z\}$, are not tested for interference freedom. In [AdBO90] two ways of solving this problem are proposed. The first is to retain in the proof outlines the correctness proofs of both the first and second premise of this rule, so that all assertions used in these proofs are subjected to the interference freedom test. Another is by adding additional requirements to the definition of interference freedom that ensure that the used termination functions decrease along all syntactically possible paths through the program. The latter approach suffices to validate the termination proofs presented in [OG76a].

Finally, let us mention that in [OA88] a transformational approach to reason about termination under the fairness assumption, that was presented in Subsection 7.2, was extended to parallel programs by combining it with the Owicki-Gries method.

Independently of the Owicki-Gries method of [OG76a], L. Lamport proposed in [Lam77] an essentially equivalent approach to verification of parallel programs. The difference was that, as in [Flo67], he considered programs presented as flowcharts, now one for each component program. As a result the interference-freedom test referred to the assertions attached to the flowchart nodes and translated into a requirement that these assertions are *monotone*, that is, that they are maintained by the actions of the other components. Lamport used this approach to establish correctness of a solution to the mutual exclusion problem. In the paper also program properties were considered that were not addressed so far. We shall return to this matter in the last section of the paper.

**Auxiliary variables** One of the features of Owicki-Gries method is the use of auxiliary variables. The need for auxiliary variables in correctness proofs was already observed in [Cli73]. In [AdBO09] it was noted that the AUXILIARY VARIABLES rule is already needed to reason about disjoint parallel programs. Indeed, the correctness formula

$$\{x = y\}\ [x := x + 1 \| y := y + 1]\ \{x = y\}$$

cannot be proved using the DISJOINT PARALLELISM rule and the rules of the proof system $\mathcal{H}$.

The most extensive analysis of these matters was provided in [dGR16] that clarified and extended initial results of [Kle98] and [Kle99]. The authors of [dGR16] showed that the AUXILIARY VARIABLES rule is a *derived* rule both in the proof system $\mathcal{H}$ and in a proof system for parameterless recursive procedures, which means that it can be eliminated from any proof that uses it. In the case of disjoint parallel programs the authors showed that this rule can be replaced by the simpler $\exists$-INTRODUCTION rule discussed in Subsection 5.1.

They also showed that for parallel programs with shared variables the AUXILIARY VARIABLES rule turns out to be essential. On the other hand, no other proof rules are needed. Indeed, Owicki proved in [Owi76] that the above presented proof system for parallel programs is complete in the sense of Cook.

**The rely-guarantee approach** A drawback of the Owicki-Gries method is that due to the test of interference freedom, verification of a parallel program $S \equiv [S_1 \| \ldots \| S_n]$ is possible only if all components $S_1, \ldots, S_n$ of $S$ are explicitly given. Further, this method is not compositional, where 'compositionality' means that (partial) correctness of a parallel program is derived directly from the correctness of its components.

This prompted research on alternative formalisms for reasoning about parallel programs. A discussion of these approaches can be found in [dRdBH$^+$01, pp. 479-484]. Here we limit ourselves to an account of one, perhaps most successful proposal, called the *rely-guarantee approach*. It was introduced in the PhD thesis of C.B. Jones, [Jon81], the essence of which appeared in [Jon83]. This approach provides a compositional way reasoning about concurrent programs by incorporating the interference-freedom test into the proof. This is achieved by using more informative correctness formulas.

We follow here the presentation given in the book by W.P. de Roever et al. [dRdBH$^+$01]. In the rely-guarantee approach one assumes that a given program $S$ is executed in some environment and therefore uses an extended specification format

$$\langle R, G \rangle : \{P\}\ S\ \{Q\},$$

where a correctness formula $\{P\}\ S\ \{Q\}$ is extended by an *interface specification* $\langle R, G \rangle$ of the environment consisting of a rely condition $R$ and a guarantee condition $G$. In contrast to the asertions $P$ and $Q$ in the correctness formula, $R$ and $G$ are *predicates on transitions*, i.e., they relate two states, the one before executing a transition and the one after it. As in Turing's flowchart in Subsection 2.1, primed versions of variables are used to refer to the state after executing the transition. For example, $x' < x$ expresses that the value of the variable $x$ decreases. The idea is that $R$ states assumptions that $S$ makes on the transitions of its environment and that $G$ states the guarantees that $S$ provides to the environment.

Informally, $\langle R, G \rangle : \{P\}\ S\ \{Q\}$ expresses that the correctness formula $\{P\}\ S\ \{Q\}$ is true in the sense of partial correctness if

- whenever at some moment during the computation of $S$ all past environmental transitions satisfy $R$
- then all transitions of $S$ up to that moment satisfy $G$.

For parallel composition of programs $S_1$ and $S_2$ the following proof rule is presented in [dRdBH$^+$01]:

PARALLEL COMPOSITION

$$\frac{\begin{array}{l}(R \vee G_1) \to R_2,\\ (R \vee G_2) \to R_1,\\ (G_1 \vee G_2) \to G,\\ \langle R_i, G_i \rangle : \{P_i\}\ S_i\ \{Q_i\}, i = 1, 2,\end{array}}{\langle R, G \rangle : \{P_1 \wedge P_2\}\ [S_1 \| S_2]\ \{Q_1 \wedge Q_2\}}$$

In [dRdBH+01, p. 453] this rule is explained as follows:

- every transition of $S_i$ (which is characterized by $G_i$) and every transition of the common environment of $S_1$ and $S_2$ (characterized by $R$) is seen by $S_j$ with $i \neq j$ as an environment transition which has to satisfy $R_j$,
- every transition by $S_1$ or $S_2$ is a transition of $[S_1 \| S_2]$, and therefore has to satisfy $G$, and
- since the validity of $\langle R_i, G_i \rangle : \{P_i\}\ S_i\ \{Q_i\}$ implies $Q_i$ is invariant under $R_i$, the postcondition $Q_1 \wedge Q_2$ holds upon termination of $[S_1 \| S_2]$; since $R$ implies $R_i$, $i = 1, 2$, this implies that $Q_1 \wedge Q_2$ is invariant under $R$ after $[S_1 \| S_2]$ has terminated, too.

Note that the rule is compositional for the customary correctness formulas. The crux of applying it is to find suitable rely-guarantee conditions for which the implications in the premises hold. In [Jon83] and [dRdBH+01] this approach was illustrated by providing alternative correctness proofs of the mutual exclusion property of a solution to the mutual exclusion problem and of a parallel version of the `FIND` program considered in [OG76a]. In practise, this approach may be as difficult as proving interference freedom.

## 8.2 Reasoning about distributed programs

In [Hoa78] Hoare introduced an elegant approach to distributed programming based on synchronous communication. In an intentional analogy to the title of Dijkstra's seminal paper [Dij68] on parallel programs, this proposal was called Communicating Sequential Processes. In the paper Hoare introduced a simple programming language for distributed programming, called since then CSP, in which Dijkstra's guarded command language was extended by allowing communication primitives for synchronous communication. This focus on synchronous, as opposed to asynchronous, communication had a huge impact on the theory of distributed programming and was also realized in the programming language Occam [INM84]. The idea is that synchronous communication is simpler to reason about as it obviates the discussion of message ordering and buffers and their management. Synchronous communication can be implemented by means of asynchronous one, so it can be viewed as an elegant abstraction.

In [AFdR80] a proof system was proposed to reason about a natural class of CSP programs. The crucial idea of this approach was an introduction of a *cooperation test* that corresponds to the interference test of the Owicki-Gries method.

To explain the matters we first clarify the relevant aspects of CSP. Each CSP program consists of a parallel composition of processes, written as

$$[PR_1 :: S_1 \| \ldots \| PR_n :: S_n]$$

Each $PR_i$ is a label of a process and $S_i$ is its program. These processes share no variables. They communicate by means of synchronous communication that is achieved by means of two matching *input/output commands*, in short i/o commands. An i/o command has the form $PR_i?x$ (an input command) or $PR_i!t$ (an output command), where $x$ is a variable, and $t$ an expression. The i/o commands $PR_i?x$ or $PR_j!t$ *match* if $i \neq j$, $PR_i?x$ appears in the program for process $PR_j$, $PR_j!t$ appears in the program for process $PR_i$, and the types of $x$ and $t$ coincide.

When the control in the programs for processes $PR_i$ and $PR_j$ is just in front of the mentioned i/o commands and they match, they can be executed jointly, with the effect that the value of $t$ is assigned to $x$. So the effect of the joint execution of the commands $PR_i?x$ or $PR_j!t$ is that of an assignment $x := t$.

In the CSP language, Dijkstra's guarded commands are generalized by allowing i/o commands to appear in the guards. So the guards can now also be of the form $B;\alpha$, where $B$ is a Boolean expression and $\alpha$ is an i/o command, or $\alpha$, which is an abbreviation for **true**;$\alpha$. If $B$ evaluates to **true** the i/o command $\alpha$ of the generalized guard behaves the same way as the usual i/o command, though it fails if it addresses a process that terminated. If $B$ evaluates to **false** the generalized guard fails. Further, Dijkstra's **do-od** notation is replaced by using a star '*' and the '[' and ']' brackets, while the **if-fi** notation is replaced by the '[' and ']' brackets, that are also used to enclose the parallel composition.

As an example of a CSP program consider the following transmission problem, taken from [AdBO09], that is a simplified version of a similar problem discussed in [Hoa78]. We wish to transmit from the *SENDER* process to the *RECEIVER* process through a *FILTER* process a sequence of characters in such a way that *FILTER* process deletes from the sequence all blank characters. The following CSP program

$$[\,\mathit{SENDER}\ ::S_1 \parallel \mathit{FILTER}\ ::S_2 \parallel \mathit{RECEIVER}\ ::S_3\,],$$

where

$$S_1 \equiv i := 0;\ *[\,i \neq M; \mathit{FILTER}\,!\,a[i] \rightarrow i := i+1\,],$$

$$\begin{aligned}
S_2 \equiv\ & in := 0;\ out := 0;\ x := \text{`\ '};\\
& *[\,x \neq \text{`}*\text{'}; \mathit{SENDER}\,?\,x \rightarrow\\
& \qquad \textbf{if}\ x = \text{`\ '} \rightarrow\ skip\\
& \qquad \Box\ x \neq \text{`\ '} \rightarrow\ b[in] := x;\\
& \qquad\qquad\qquad in := in+1\\
& \qquad \textbf{fi}\\
& \quad \Box\ out \neq in; \mathit{RECEIVER}\,!\,b[out] \rightarrow out := out+1\\
& \,],
\end{aligned}$$

$$\begin{aligned}
S_3 \equiv\ & j := 0;\ y := \text{`\ '};\\
& *[\,y \neq \text{`}*\text{'}; \mathit{FILTER}\,?\,y \rightarrow c[j] := y; j := j+1\,].
\end{aligned}$$

is a solution to this problem.

Here the sequence of characters is initially stored in the array $a[0 : M-1]$ of characters in the process *SENDER*. The last element of the array is the special character '$*$', i.e., $a[M-1] =$ '$*$'. The process *FILTER* has an array $b$ of characters serving as an intermediate store for processing the character sequence and the process *RECEIVER* has an array $c$ of characters to store the result of the filtering process. For coordinating its activities the process *FILTER* uses two integer variables *in* and *out* pointing to elements in the array $b$.

The process *FILTER* can communicate with both other processes. It can receive characters from process *SENDER* until '$*$' has been received and it can transmit all nonblank characters to the process *RECEIVER*. The Boolean parts of the generalized guards of the *FILTER* process can both evaluate to **true**. In that case the next action can be either a communication between *SENDER* and *FILTER* or between *FILTER* and *RECEIVER*. Consequently this CSP program exhibits a nondeterministic behaviour.

The process *SENDER* terminates once it has sent all its $M$ characters to the *FILTER* process. The process *FILTER* terminates when it has received the character '$*$' and it has transmitted to *RECEIVER* all nonblank characters it has received. Finally, the process *RECEIVER* terminates once it has received from *FILTER* the character '$*$'. Thus the parallel composition of these three processes terminates if *SENDER* sends as the last of its $M$ characters the '$*$'.

The proof system proposed in [AFdR80] extends the one for the guarded commands language by the following axioms and proofs rules that deal with the communication. For simplicity we assume that all variables and expressions are of the same type.

INPUT

$$\{P\}\ PR_i?x\ \{Q\}$$

OUTPUT

$$\{P\}\ PR_i!t\ \{P\}$$

GENERALIZED ALTERNATIVE COMMAND

$$\frac{\{P \wedge B_i\}\ \alpha_i\ \{R_i\}, \{R_i\}\ S_i\ \{Q\}, i \in \{1, \ldots, n\}}{\{P\}\ [\Box_{i=1}^{n}\ B_i; \alpha_i \rightarrow S_i]\ \{Q\}}$$

GENERALIZED REPETITIVE COMMAND

$$\frac{\{P \wedge B_i\}\ \alpha_i\ \{R_i\}, \{R_i\}\ S_i\ \{P\}, i \in \{1, \ldots, n\}}{\{P\}\ *[\Box_{i=1}^{n}\ B_i; \alpha_i \rightarrow S_i]\ \{P \wedge \bigwedge_{i=1}^{n} \neg B_i\}}$$

The INPUT axiom may look strange since it allows us to conclude an arbitrary postcondition. However, the used assertions still have to pass the cooperation test. This test refers to the proof outlines which are defined as in the case of Owicki-Gries approach.

Suppose now that we established the proof outlines $\{P_i\}\ S_i^*\ \{Q_i\}$, where $i \in \{1, \ldots, n\}$ and $S_1, \ldots, S_n$ are respective programs of the processes $PR_1, \ldots, PR_n$. We say that these proof outlines *cooperate* if

- the assertions used in $\{P_i\}\ S_i^*\ \{Q_i\}$ contain no variables subject to change in $S_j$ for $i \neq j$,
- $\{pre_1 \wedge pre_2\}\ PR_j?x \parallel PR_i!t\ \{post_1 \wedge post_2\}$ holds whenever $\{pre_1\}\ PR_j?x\ \{post_1\}$ and $\{pre_2\}\ PR_i!t\ \{post_2\}$ are taken respectively from the proofs outlines $\{P_i\}\ S_i^*\ \{Q_i\}$ and $\{P_j\}\ S_j^*\ \{Q_j\}$.

Intuitively, proof outlines cooperate if they help each other to validate the post conditions of the i/o commands present in these proofs. To establish cooperation the following axiom is needed.

COMMUNICATION

$$\{\mathbf{true}\}\ PR_j?x \| PR_i!t\ \{x = t\}$$

provided $PR_j?x$ and $PR_i!t$ are taken respectively from the programs of $PR_i$ and $PR_j$.

This axiom simply states that, as mentioned before, the effect of the joint execution of a pair of matching i/o commands is that of an assignment.

Then the following proof rule allows one to draw conclusion about the parallel composition of processes:

CSP PARALLELISM

$$\frac{\text{The proof outlines } \{P_i\}\ S_i^*\ \{Q_i\}, i \in \{1, \ldots, n\}, \text{ for the processes } PR_1, \ldots, PR_n \text{ cooperate}}{\{\bigwedge_{i=1}^{n}\ P_i\}\ [PR_1 :: S_1 \| \ldots \| PR_n :: S_n]\ \{\bigwedge_{i=1}^{n} Q_i\}}$$

To reason about the CSP programs, as in the case of the Owicki-Gries approach, the AUXILIARY VARIABLES rule is needed. The presented reasoning about deadlock freedom is analogous as in [OG76a], though the reasons for a deadlock can now be different. In particular, a process can be blocked forever if the control in its program is just before an i/o command that addresses a process that terminated. Using the proposed proof system some example CSP programs were proved correct in this paper. (In [Moi83] it was pointed out that one of the correctness proofs contained an error and a corrected version was presented.) The proof system was subsequently proved in [Apt83] to be complete in the sense of Cook. Neither [AFdR80] nor [Apt83] considered termination.

Independently of [AFdR80] a very similar proof system to reason about CSP programs was proposed in [LG81], in which the *satisfaction property* corresponds to the cooperation test. However, the authors used a different semantics of the generalized repetitive commands than the one stipulated in [Hoa78] and taken care of in the proof system of [AFdR80]. On the other hand, in contrast to [AFdR80], program termination was considered. A number of different approaches to reason about communicating processes was proposed in the literature around that time. They are surveyed in [HdR86].

Subsequently, a simplified proof system for a fragment of CSP was proposed in [Apt86] and used to establish correctness of a solution to the so-called *distributed termination problem*. In this fragment a program for each process consists of a guarded commands program followed by a single generalized repetitive command. The i/o commands can appear only in the guards of this repetitive command and thus not as separate statements. An example of a CSP program written in this fragment is the above-mentioned solution to the transmission problem. This fragment was studied independently in [ABC87] and [Zöb88], where it was shown that each CSP program can be transformed into a program in this subset using some control variables.

The underlying idea of this approach is that such simpler CSP programs can be transformed into Dijkstra's guarded commands language without introducing any additional variables. By absorbing this transformation into a proof rule one obtains a proof rule that uses a global invariant and deals directly with the considered CSP program. This way this approach dispenses with the cooperation test. Termination is naturally dealt with by following the approach used for the guarded commands language. This proof system was adopted in [AdBO09] and its two previous editions, where its soundness was shown to be a direct consequence of the correctness of the abovementioned program transformation.

As explained in [AFK88] in the context of CSP programs, fairness can have various interpretations. One of them states that every pair of matching i/o commands that is infinitely enabled is also infinitely often selected. In [GFK84] a proof rule for dealing with this form of fairness of the CSP programs was proposed.

The approach of [AFdR80] and [LG81] was presented only for CSP programs without nested parallelism. It was extended in [AdB90b] to CSP programs that allowed nested parallelism and also dynamic process creation, a feature not present in CSP. Subsequent work in this direction, [dB91], shifted emphasis to reasoning about objects, a subject that deserves a separate section.

Finally, let us mention that verification parallel and distributed programs in the Hoare-like style was systematically presented in a book form in [dRdBH$^+$01].

# 9 Object-oriented Programs

Object-oriented programming, as exemplified by languages like C++ or Java, builds upon the notion of an object and concepts like inheritance and subtyping. The difficulty in reasoning about such programs is that their execution creates dynamic pointer structures that go beyond the static program structure that has been the backbone of the syntax-directed Hoare-style proof rules discussed so far.

## 9.1 Language characteristics

As object-oriented programming has been realized in many, often incompatible, ways, we clarify first the main characteristics of the objects here considered. These are:

- objects possess (and *encapsulate*) their own local variables,
- objects interact via *method* calls,
- objects can be dynamically *created*.

Each object consists of a set of local variables and a set of methods. In contrast to the formal parameters of procedures and the local variables of the block statements which only exist *temporarily*, the local variables of an object exist *permanently*. To emphasize the difference between these temporary variables and the local variables of an object, the latter ones are called *instance* variables. The *local state* of an object is a mapping that assigns values to its instance variables. Each object represents its local state by a *pointer* to it. *Encapsulation* means that the instance variables of an object cannot be directly accessed from other objects; they can be accessed only by the method calls of the object.

A method call invokes a procedure which is executed by the called object. The execution of a method call thus involves a temporary *transfer* of control from the local state of the caller object to that of the called object (also referred to as *callee*). Upon termination of the method call the control returns to the local state of the caller. The method calls are the *only way* of transferring control from one object to another.

The account of verification of object-oriented programs that follows is based on [AdBO09, Chapter 6]. We distinguish two kinds of variables: the set *Var* of *normal* variables, the ones considered so far, and the set *IVar* of instance variables, which are owned by objects.

We consider a set of *methods*, each defined by means of a declaration

$$m(\mathbf{u}) :: S,$$

where the identifier $m$ denotes a method, $\mathbf{u}$ is the list of formal parameters of type *Var*, and $S$ is the *method body*, which may include recursive calls of $m$.

Methods are invoked by means of the *parameterized method calls* that are of the form

$$s.m(\mathbf{t}).$$

where $s$ is an expression that denotes the *called object* and $\mathbf{t}$ is the list of actual parameters of the method $m$.

A program consists of a main statement and a set of method definitions. In programs we use a basic (i.e., not compound) type **object** that denotes an infinite set of objects. The constant **null** of type **object** represents the *void reference*, a special construct which does not have a local state. The normal variable **self**

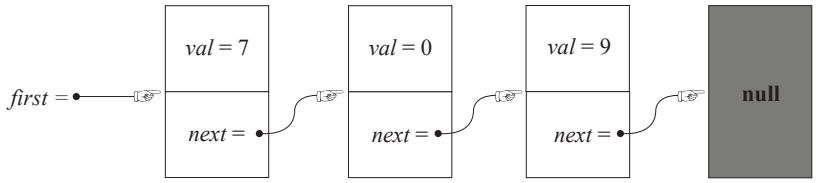


Figure 5: A linked list. This drawing is taken from [AdBO09, p. 190].

of type **object** stores at any moment the currently executing object. Inside a method body no assignments to the variable **self** are allowed, that is, this variable is read-only. Values of type **object** can only be tested for *equality*. Expressions of type **object** are called *object expressions*.

To illustrate the considered programs consider a recursive method used to find a zero in a linked list represented by objects. We represent such lists using the instance object variable *next* that links the objects of the list, and the constant **null** that allows us to identify the last element of the list. We assume that each object stores a value kept in an instance integer variable *val*. Additionally, we use the normal object variable *first* to point to the first object in the list. Figure 5 shows an example of such a list representation.

The desired method is declared as follows.

*find* :: **if** $val = 0$
**then** $return :=$ **self**
**else if** $next \neq$ **null**
**then** *next.find*
**else** $return :=$ **null**
**fi**
**fi**

Then upon termination of the call **self**.*find* the object variable *return* points to the first object in the list that stores zero, if it exists, and otherwise it returns the void reference, represented by the constant **null**.

## 9.2 Reasoning about object-oriented programs

The methods in object-oriented programs use local updates to manipulate a global, dynamically changing pointer structure, the *heap*. Therefore the assertion language must be able to express properties of the heap. To this end, *global expressions* are introduced by extending the (local) expressions that may appear inside methods by *navigation expressions* of the form $e.u$, where $e$ is an object expression and $u$ an instance variable. If $e$ denotes a certain object (for example **self**) and $u$ is an instance variable (for example *next*), then $e.u$ points to the corresponding object (in the example **self**.*next*). This way, one can navigate from object to object along the pointers in the heap. Assertions are then constructed from global Boolean expressions by allowing Boolean combinations and quantification over normal variables in *Var*.

As an example consider the assertion used in [AdBO09, pp. 226-227] to reason about the above method *find*:

$$P \equiv \textbf{self} = a[k] \wedge a[n] = return \wedge \forall\, i \in [k : n-1] : (a[i] \neq \textbf{null} \wedge a[i].val \neq 0 \wedge a[i].next = a[i+1]).$$

In addition to the instance object variables *next* and *return* and the instance integer variable *val* used in the definition of *find*, one uses here the following normal variables: an array $a$ of type $\mathbf{integer} \to \mathbf{object}$, integer variables $i, k$ and $n$, and the object variable **self**.

The assertion $P$ states that the array section $a[k:n]$ stores a linked list of objects which starts with the object **self**, ends with the object *return*, and all of its objects, except possibly the last one, are different from **null** and do not store in *val* zero. Note the use of the navigation expressions $a[i].val$ and $a[i].next$.

The desired behaviour of the above method *find* can then be specified by means of the following correctness formula:

$$\{\mathbf{true}\}\ \mathbf{self}.find\ \{Q\}, \tag{3}$$

where the postcondition $Q$ is defined in terms of the assertion $P$:

$$Q \equiv (return = \mathbf{null} \vee return.val = 0) \wedge \exists\ a : \exists\ k : \exists\ n \geq k : P.$$

So the postcondition states that the returned object is **null** or stores zero and that for some array section $a[k:n]$ the assertion $P$ holds.

To reason about such correctness formulas we need new axioms and proof rules.

Correctness of the customary assignment $x := t$ to normal variables $x$ is captured by the ASSIGNMENT axiom that for a given postcondition $P$ calculates the precondition by applying the substitution $[x := t]$ to $P$. In object-oriented programs one can additionally use assignments to instance variables and use the dereferencing $s.m(\mathbf{t})$ in the method calls. The latter calls for an extension of the assertion language by allowing the corresponding expressions $e.u$.

F.S. de Boer proposed in [dB99] the following textually identical axiom for such assignments:

ASSIGNMENT TO INSTANCE VARIABLES

$$\{P[u := t]\}\ u := t\ \{P\},$$

where $u$ is a (possibly subscripted) instance variable in $IVar$.

So, as in the case of the assignment to subscripted variables, the solution relies on an extension of the definition of substitution, in this case to the instance variables.

As usual, the definition of the substitution $P[u := t]$ proceeds by induction on the structure of $P$. The difference appears at the level of expressions $s$ for which the definition of $s[u := t]$ is more elaborate. If $s$ is of the form $e.u$ for an object expression $e$, the substitution has to take possible aliases of $e.u$ into account. More precisely, if after applying the substitution inductively to $e$, the result refers to the currently active object, i.e., if $e[u := t] = \mathbf{self}$, then the outcome of the substitution is $t$. Otherwise, $u$ is left untouched and the substitution is applied inductively to $e$. This is expressed in the following definition:

$$e.u[u := t] \equiv \mathbf{if}\ e[u := t] = \mathbf{self}\ \mathbf{then}\ t\ \mathbf{else}\ e[u := t].u\ \mathbf{fi}.$$

In case of only one (possibly recursive) method definition, the RECURSION rule for procedure calls can be adapted to method calls as follows, where we use the block statement discussed in Subsection 5.2:

RECURSION IV

$$\frac{\{P\}\ s.m(\mathbf{t})\ \{Q\} \vdash \{P\}\ \mathbf{begin\ var\ self, u} := s, \mathbf{t};\ S\ \mathbf{end}\ \{Q\}}{\{P\}\ s.m(\mathbf{t})\ \{Q\}}$$

where the method $m$ is declared by $m(\mathbf{u}) :: S$.

So, as for recursive procedures we may use the desired conclusion as a hypothesis in the correctness proof of a block statement, where **self** and the list $\mathbf{u}$ of formal parameters of the method are treated as local variables that are respectively initialised by the called object $s$ and the list $\mathbf{t}$ of actual parameters and are accessed in the method body $S$. A rule analogous to the RECURSION III rule is adopted to deal with total correctness.

To adjust correctness formulas that deal with generic method calls $y.m(\mathbf{x})$ to specific objects $s$ and lists of actual parameters $\mathbf{t}$, we modify the SUBSTITUTION rule as follows, where we refer to the given set $D$ of method definitions:

INSTANTIATION

$$\frac{\{P\}\ y.m(\mathbf{x})\ \{Q\}}{\{P[y, \mathbf{x} := s, \mathbf{t}]\}\ s.m(\mathbf{t})\ \{Q[y, \mathbf{x} := s, \mathbf{t}]\}}$$

where $y, \mathbf{x}$ is a list of variables in $Var$ which do not appear in $D$ and $var(s, \mathbf{t}) \cap change(D) = \emptyset$.

In [AdBO09] these axioms and proof rules were used to establish total correctness of the above example program expressed by the correctness formula (3) and of an object-oriented a program that inserts an element into a linked list.

## 9.3 Object creation

An object can be dynamically created by the assignment statement $u :=$ **new**, where $u$ is an object variable and **new** is a keyword in the considered programming language. The execution of this statement creates a *new* object and assigns its identity to the variable $u$. This new object comes with a default initialisation of all its instance variables. This can be modelled by some bookkeeping of the set of objects that are currently created, for example by maintaining a counter referring to an unbounded array. This allows one to reason about this assignment using a small program transformation. A drawback of this approach is that it refers to an explicit implementation.

An alternative is to use a substitution of $[x :=$ **new**$]$ for object variables and define its application to a limited class of assertions (we call them *pure*) that take into account that object variables can only be compared for equality or be dereferenced in the method calls $s.m(\mathbf{t})$, and in which one does not quantify over them. This leads to the following axiom proposed in [dB99]:

OBJECT CREATION

$$\{P[x := \mathbf{new}]\}\ x := \mathbf{new}\ \{P\},$$

where $x$ is a simple (so not subscripted) object variable and $P$ is a pure assertion.

We omit the details of the definition of the substitution $P[x :=$ **new**$]$ and only remark that one cannot simply replace $x$ in $P$ by the keyword **new** because it is *not* an expression of the assertion language. The details are given in [AdBO09, Chapter 6].

C. Pierik and F.S. de Boer showed in [PdB03] how the approach of [dB99] can be extended to deal with inheritance and subtyping. The same authors introduced in [dBP03] a general methodology for obtaining relatively complete Hoare's logics for object-oriented programs. A key issue is the extension of Gorelick's most general formulas [Gor75] (see Subsection 6.1) to deal with the states of object-oriented programs.

As a final contribution to this line of research on verification of object-oriented programs let us mention [ÁdBdRS05] in which a proof system for partial correctness and deadlock freedom was developed for a subset of Java. The considered subset comprised the object-oriented core of Java, as well as concurrency via thread classes, allowing for a multithreaded flow of control. The Java concurrency model includes synchronous message passing, dynamic thread creation, shared-variable concurrency via instance variables, and coordination via reentrant synchronization monitors. The verification method was formulated in terms of proof outlines, where the assertions were layered into local ones specifying the behavior of a single instance, and global ones taking care of the connections between objects. The proof outlines were tested both for interference freedom of shared-variable concurrency as in [OG76a] and for cooperation of synchronous message passing as in [AFdR80]. The authors establish the soundness and the relative completeness of their proof system. From an annotated program, a number of verification conditions were generated and discharged using the interactive theorem prover PVS [ORS92].

### 9.4 Dynamic typing

B. Engelmann [EO16, Eng17] considered object-oriented programs in the context of *dynamic typing*. This means that variables do not have an a priori declared static type but may assume any value during the execution of the program. As an example consider the following method from [Eng17], where it is assumed that $b$ is a Boolean variable:

$$num\_or\_string(b) :: \textbf{if } b \textbf{ then } x := y := 5 \textbf{ else } x := \texttt{``foo''};\ y := \texttt{``bar''} \textbf{ fi};\ z := x + y$$

If $b$ is true, the method call $num\_or\_string(b)$ yields the numeric value $z = 10$, but if $b$ is false, it yields the string value $z =$ ``foobar''. So the type of $x$ and $y$ is determined dynamically during runtime, and it is either numeric or string, with the operation + being either addition or string concatentation. Such dynamically typed programs are present in the list processing language LISP and in the widespread programming languages like Python and JavaScript.

Engelmann [Eng17] developed a Hoare-like proof system for a model language **dyn** of object-oriented programs with dynamic typing. He proved soundness and relative completeness of his system, thereby extending the arguments of [Coo78, Gor75] and [dBP03] for the object-oriented part.

## 10 Alternative Approaches

### 10.1 Weakest precondition semantics and systematic program development

In [Dij75] Dijkstra suggested an alternative approach to program verification, called *weakest precondition semantics*. The idea is that given a program $S$ and a desired postcondition $P$ we would like to find the weakest precondition $wp(S, Q)$ such that $\{wp(S, Q)\}\ S\ \{Q\}$ holds in the sense of total correctness. 'Weakest' means here that for any precondition $P$ such that $\{P\}\ S\ \{Q\}$ holds in the sense of total correctness, the implication $P \to wp(S, Q)$ holds.

This approach then differs from Hoare's original approach by

- insisting on total correctness instead of on partial correctness,
- assuming that initially only the postcondition is given.

Additionally, Dijkstra insisted that the program should be developed *together* with its correctness proof. In [Dij75] he advanced this approach for his guarded command language that we briefly discussed in Subsection 7.1. Its notable feature was use of programming constructs that support nondeterminism. Another interesting feature of the language was *parallel assignment* $\mathbf{x} := \mathbf{t}$, where $\mathbf{x}$ is a list of different variables and $\mathbf{t}$ is a list of expressions of the same length. This construct is for example useful to swap the values of variables without additional variables:

$$x, y := y, x.$$

The weakest precondition $wp(S, Q)$ is defined by induction on the structure of the program, with

- $wp(\mathbf{x} := \mathbf{t}, Q) \equiv Q[\mathbf{u} := \mathbf{t}]$,
- $wp(S_1;\ S_2, Q) \equiv wp(S_1, wp(S_2, Q))$,

as typical clauses.

The main problem is how to deal with the loops. In [Dij75] the weakest precondition for guarded commands was defined as follows. Denote the alternative command **if** $B_1 \to S_1 \Box \ldots \Box B_n \to S_n$ **fi** by $IF$, the repetitive command **do** $B_1 \to S_1 \Box \ldots \Box B_n \to S_n$ **od** by $DO$ and abbreviate $\bigvee_{i=1}^{n} B_i$ to $BB$. Then

$$wp(DO, Q) \equiv \exists k : k \geq 0 : H_k(Q),$$

where

$$H_0(Q) \equiv (Q \wedge \neg BB)$$

and for $k > 0$

$$H_k(Q) \equiv wp(IF, H_{k-1}(Q)) \vee H_0(Q).$$

Intuitively, $H_k(Q)$ is the weakest precondition guaranteeing proper termination in a state satisfying $Q$, after at most $k$ guard selections.

Since $k$ is used as a subscript of $H_k$, '$\exists k : k \geq 0$' is here not a customary quantification but a shorthand for an infinite disjunction, with additionally the formulas $H_k(Q)$ defined by induction. In other words, so defined weakest precondition of a repetitive command is not an assertion in a first-order language. This underlies the difficulty of finding loop invariants, the problem we already mentioned in Subsection 3.1, and clarifies why finding loop invariants is an important problem in developing correct programs or in establishing their correctness.

The idea of developing programs together with their correctness proofs was subsequently presented in a book form in [Dij76a], where the weakest precondition semantics was extended to blocks and procedures. A systematic development of provably correct programs was further advanced by David Gries in his book [Gri81], notably by proposing a number of heuristics for finding loop invariants. Other books devoted to this subject are [Bac86, DF88, Kal90, BvW08]. Various aspects of the weakest precondition semantics were further discussed in a book form in [DS90]. The problem of finding loop invariants became central in the subsequent study of program verification and development. [FMV14] surveys various ways of constructing loop invariants, and provides their classification and analysis.

Another matter relevant for a systematic program development is termination. Both the WHILE II and RECURSION III rules provide only a kind of template for an actual termination proof, without explaining how the termination functions are to be found. Some sophisticated techniques were developed to establish termination. They go beyond the framework of Hoare's logic or the weakest precondition semantics and

rely on various methods developed in other areas, notably term rewriting systems. The authors of [CPR11] provide an accessible account of the recent developments. In particular, they explain a recent alternative approach to proving termination called *disjunctive termination argument*: only one of a disjunction of the termination functions needs to decrease, but this has to be the case after any number of iterations of the loop. It is argued that disjunctive termination arguments are easier to find than the classical termination argument dating back to Turing [Tur49] (see Subsection 2.1), where a single termination function has to decrease its values taken from a well-founded set with each iteration of the loop.

One difficulty of this approach for program verification is that it is not clear how to extend the weakest precondition semantics to deal with advanced programming constructs, for instance arbitrary procedures as considered in Section 6. In contrast, Hoare's approach is more flexible. In particular, as we have seen in Subsection 5.1, the reasoning can be supported by various adaptation rules. This cannot be done in the framework of the weakest precondition semantics which requires computing a single assertion. On the other hand, as we saw in Subsection 7.3, the weakest precondition semantics turned out to be helpful for the verification of probabilistic programs.

## 10.2 Programming from specifications

R.-J. Back [Bac80] and C. Morgan [Mor94] extended Dijkstra's approach to a methodology for "programming from specifications" by a rule-based step-by-step development of specifications into programs. To this end, they introduced a *specification* statement. In [Mor94] it takes the form $\mathbf{x} : [P, Q]$, with the meaning

> "*If* the initial state satisfies the precondition $P$ *then* change only the variables listed in $\mathbf{x}$ so that the resulting final state satisfies the postcondition $Q$."

While the variables that are changed by a specification $S \equiv \mathbf{x} : [P, Q]$ are explicitly mentioned, namely $change(S) = \{\mathbf{x}\}$, there is no information which variables are accessed, i.e., $var(S)$ is not defined. Recall that various Hoare-style proof rules for programs $S$ have application conditions concerning $var(S)$. Also, Gorelicks's most general formulas require fresh variables outside of $var(S)$ to freeze their initial values. To overcome this weakness, Morgan considers *logical constants* that by definition appear only in assertions and are thus never changed by any program or specification. With a logical constant $X$ the initial values of a variable $x$ can be frozen. For example, $x : [x = X, x > X]$ specifies that $x$ should be increased. The amount of the increment is left unspecified.

The idea is that specifications and programs are handled on the same footing, so that programming operators like sequential composition or loops can be applied to specifications, as well. Constructs $S_1$ and $S_2$ in this extended syntax can be compared by a *refinement relation*: $S_1 \sqsubseteq S_2$ denotes that $S_1$ is refined by $S_2$.

Semantically, specifications and programs are considered as *predicate transformers* that transform given postconditions into the corresponding weakest preconditions. Refinement $S_1 \sqsubseteq S_2$ means that for all postconditions $Q$ the implication

$$wp(S_1, Q) \to wp(S_2, Q)$$

holds, i.e., $S_2$ establishes the postcondition $Q$ in at least all states where $S_1$ establishes $Q$. For example, $x : [x = X, x > X] \sqsubseteq x := x + 42$. In this approach specifications are stepwise *refined* to programs by the applications of refinement rules, described in [Mor94].

Morgan [Mor94] also considers a number of advanced programming concepts like recursive procedures with parameters, modules with local declarations of variables and procedures, and data refinement.

Subtly different from Morgan's specification statement is the *generic commend* of J. Schwarz [Sch77]. It is written as $[P \Rightarrow Q][X]$, where $P$ is a precondition, $Q$ a postcondition, and $X$ a set of variables. Semantically, a generic command denotes a certain state transformer. Assume an interpretation $I$. A *state transformer $T$ based on* a finite set $X$ of variables is a binary relation on the set $\Sigma$ of states that has only read or write access to the states via the variables in the set $X$. For each program $S$ its meaning is a state transformer based on $var(S)$. Let $\mathcal{T}(X)$ be the set of all state transformers based on $X$, ordered by the set inclusion. Now the semantics $\mathcal{M}_I$ relative to $I$ of a generic command $[P \Rightarrow Q][X]$ is given by

$$\mathcal{M}_I[\![[P \Rightarrow Q][X]]\!] = \bigcup\{S \mid S \in \mathcal{T}(X) \text{ and } \{P\}\ S\ \{Q\} \text{ is true in } I \text{ in the sense of partial correctness}\}.$$

So given an interpretation $I$, $[P \Rightarrow Q][X]$ denotes the largest w.r.t. the set inclusion state transformer $S$ based on $X$ that satisfies $\{P\}\ S\ \{Q\}$ in the sense of partial correctness. Schwarz used generic commands for stating an alternative version of a RECURSION rule, but did not embark on program development. Morgan's specification statement $\mathbf{x} : [P, Q]$ is similar to the generic command $[P \Rightarrow Q][X]$, except that the list $\mathbf{x}$ does not record the variables that are only read, which are also covered by $X$.

## 10.3 Algorithmic logic and dynamic logic

Hoare's logic is geared towards establishing program correctness. However, from the point of view of mathematical logic it has a very rigid syntax: the correctness formulas cannot be negated or combined, for example by disjunction. As a result one cannot view Hoare's logic as an extension or a modification of some existing logics, even though it crucially relies on first-order logic and its extensions, for example by allowing subscripted variables. In some alternative approaches one could view reasoning about programs as an extension of reasoning about existing logics. We discuss now briefly two most prominent examples.

*Algorithmic logic* was originally proposed in [Sal70] and presented in a book form in [MS87]. It extends first-order language by expressions that can be interpreted as programs and constructs that allow one to mix formulas and programs. Interestingly, substitutions for a sequences of variables are viewed as atomic programs. This is equivalent to the mentioned earlier parallel assignment of Dijkstra, and also shows a close connection with the ASSIGNMENT axiom. Programs are built by allowing formulas as tests and using program composition, conditionals, and loops, all written in a compact notation. For example, **if** $B$ **then** $S$ **else** $T$ is written as $\underline{\mathrm{v}}[BST]$. In turn, the construct $*[BST]$ corresponds to **while** $B$ **do** $S$ **od**; $T$.

Expressions of the form $S\phi$, where $S$ is a program and $\phi$ a formula, correspond to the strongest postcondition introduced in Subsection 4.2. Further, it is shown how termination of the considered programs can be expressed as a formula that admits countable disjunction. This is analogous to the definition of $wp(DO, Q)$ given earlier.

The relation between Hoare's logic and algorithmic logic becomes clear when one realizes that the correctness formula $\{P\}\ S\ \{Q\}$ can be expressed as the implication $SP \to Q$. Consequently, rules used in Hoare's logic can be readily reproduced as rules in algorithmic logic, in particular various forms of adaption rules. This straightforward modelling, however, does not yield new insights concerning program verification.

Research on algorithmic logic focused mostly on such matters as studies of consistency and the infinitary completeness of selected theories, derivation of the normal forms of programs, and axiomatization of various data structures, rather than on (relatively) complete axiomatizations of fragments concerned with specific features of programming languages, a direction Hoare's logic took.

*Dynamic logic* was originally proposed in [Pra76] and presented in a book form in [Har79] and more extensively in [HKT00]. It is very similar to algorithmic logic introduced six years earlier, though it was de-

veloped independently. It enriches first-order logic by constructs reminiscent of modal logic. First, programs are defined starting from atomic actions and tests, using the sequential composition (;), nondeterministic composition ($\cup$) (absent in algorithmic logic), and iteration ($*$) that corresponds to Kleene's star. In the dynamic logic syntax the **while** $B$ **do** $S$ **od** statement can be expressed as $(B;S)^* \cup \neg B$.

Further, one admits formulas of the form $[S]\phi$, where $S$ is a program and $\phi$ is a formula, with the intended interpretation "every execution of the program $S$ from the current state leads to a state in which $\phi$ is true". The formulas and programs are defined by simultaneous induction, allowing the usual Boolean connectives. A dual formula to $[S]\phi$ is $\langle S\rangle\phi$, defined by

$$\langle S\rangle\phi \equiv \neg[S]\neg\phi.$$

So its intended interpretation is "some execution of the program $S$ from the current state leads to a state in which $\phi$ is true". The $[S]$ and $\langle S\rangle$ operators can thus be viewed as the counterparts of the $\Box$ and $\Diamond$ operators in the propositional modal logic, but parameterized with a program $S$.

Research on dynamic logic mainly focused on a study of various fragments or extensions, with the corresponding sound and complete axiomatizations, and the corresponding decidability and computational complexity results. In particular, dynamic logic was extended in [Har79] to deal with recursive procedures.

Typical axioms are:

$$[S;T]\phi \leftrightarrow [S][T]\phi,$$

which corresponds to one of the mentioned clauses that define the weakest precondition, and

$$[S^*]\phi \leftrightarrow \phi \wedge [S][S^*]\phi,$$

that captures the idea that $^*$ stands for the infinite iteration.

The relation between Hoare's logic and dynamic logic is easily established by noticing that the correctness formula $\{P\}\ S\ \{Q\}$ can be expressed as the implication $P \rightarrow [S]Q$. So $[S]Q$ models what is called the weakest liberal precondition introduced in Subsection 4.2. Consequently, as in the case of algorithmic logic, proof rules of the proof system $\mathcal{H}$ of Subsection 3.1 can be translated into proof rules of dynamic logic. Moreover, these translated rules are derivable from the adopted axioms and proof rules. Further, thanks to the richer syntax, it is possible to express other program properties and discuss such properties like program equivalence. For example, the following formula states that the program $S$ is deterministic:

$$\langle S\rangle\mathbf{true} \rightarrow [S]\mathbf{true}.$$

This translation of the proof rules of the proof system $\mathcal{H}$ presented in Subsection 3.1 results in proof rules that are derivable in dynamic logic.

## 10.4 Temporal logic and model checking

Research on verification of parallel programs carried out in the seventies showed limitations of Hoare's logic in reasoning about concurrent programs. We discussed three properties of such programs: partial correctness, termination, and absence of deadlock. However, in contrast to the sequential programs, concurrent programs are often supposed to operate repeatedly, in a cyclic fashion. For instance, a solution to the mutual exclusion problem deals with infinite executions of the program components operating in parallel. This calls for a study of properties (such as an *eventual access*) that cannot be expressed in Hoare's logic.

To express such concepts and to systematically reason about them Amir Pnueli proposed in [Pnu77] to use *temporal logic*. Using it one can express in a natural way various program properties that do not necessarily deal with the input/output behaviour of a program. In contrast to Hoare's logic the reasoning about programs is not syntax-directed. Instead, one usually reasons about specific control points in a program and a relation between them.

For example, the following formula states that a process $P$ infinitely often enters its critical section $CS$:

$$\Box\Diamond \; in\, CS$$

where $in\, CS$ is a formula that states that the control in the process $P$ is within $CS$.

In turn, to express the strong fairness assumption for the repetitive command

$$S \equiv \textbf{do}\ B_1 \to S_1 \Box \ldots \Box B_n \to S_n\ \textbf{od}$$

we can use following formula:

$$\bigwedge_{i=1}^{n} (\Box\Diamond\, (at\, S \wedge B_i) \to \Box\Diamond\ at\, S_i),$$

where $at\, T$ holds when the control in the considered program is just in front of $T$.

Appropriate axioms and proof rules were then developed to reason about such formulas. Temporal logic, applied to concurrent programs, grew into an impressive research area, see in particular the books [MP91, MP95] of Z. Manna and A. Pnueli. This line of research should be viewed as complementary to Hoare's logic, that is why we do not devote more space to it.

In that context a distinction, first advanced in [Lam77], is useful. A *safety property* states 'nothing bad will happen' during a program execution, while a *liveness property* states that eventually 'something good will happen' during a program execution. According to this distinction partial correctness, absence of errors, and deadlock freedom are safety properties. In contrast, program termination, fairness, eventual access, infinite access, etc. are liveness properties. Hoare's logic is a natural vehicle to prove safety properties. In temporal logic these properties can be formulated by means of invariants that become formulas of the form $\Box\phi$, while liveness properties are formulas of the form $\Diamond\phi$.

Temporal logic led in turn to *model checking*. Independently, E.M. Clarke and A. Emerson [EC82], as well as J.P. Queille and J. Sifakis [QS81], discovered that the problem of checking whether a finite-state system satisfies (is a *model* of) a propositional temporal logic formula is decidable, providing efficient algorithms for it. This was the start of enormous research activities extending the scope of model checking so that even industrial-size problems could be tackled [CGH$^+$93]. Interestingly, model checking is often used to debug a system because in case the system does not satisfy the temporal logic specification, model checkers can provide a counterexample that is helpful for understanding the mismatch between a system and its specification. In the recent years model checking has been extended to infinite-state systems, mostly by automatically constructing and refining abstractions of the system, a method known as *counterexample-guided abstraction refinement*, see [CGJ$^+$03]. The state of the art of model checking is represented in the handbook [CHVB18].

## 10.5 Separation logic

This approach to program verification was originally developed as an extension of Hoare's logic to reason locally about about pointer structures, see [ORY01, Rey02, OYR04, O'H19]. To cope with pointers, separation logic builds upon a semantic model, in which a state is a pair $(s, h)$ consisting of a *store* $s$ and a *heap*

$h$. A store $s$ is a mapping from variables to values (so a state in the sense of Subsection 4.1), which may be data values such as integers, or pointer values such as addresses. A heap $h$ is a finite partial mapping from addresses (or cells) to values, which again can be data or pointers. It is assumed that the addresses are integers, which in turn are values.

Several low-level statements for manipulating the heap are considered [ORY01, O'H19]. Let $x$ stand for a variable and $e$ for an integer expression, which can denote an address in the heap. One can explicitly distinguish between an address and its contents: $[e]$ denotes the contents of the heap at address $e$. *Assignments* affect only the store. Besides the normal assignments $x := e$ there are *lookups* $x := [e]$, where $e$ is interpreted as an address in the heap and the contents of $e$ is assigned to the variable $x$ in the store. *Mutations* affect the heap. An *update* $[e] := e'$ expresses that the contents of the address $e$ in the heap becomes value of the expression $e'$. Further, $x := \mathbf{alloc}()$ expresses that the address $x$ is newly allocated to the heap, and $\mathbf{free}(x)$ expresses that the address $x$ is deallocated from the heap.

Separation logic extends the usual assertion language of Hoare's logic to specify properties of the heap: **emp** asserts that the heap is empty, $e \mapsto e'$ asserts that the heap consists of one cell, with address $e$ and contents $e'$, and $e \mapsto -$ asserts that the heap consists of one cell, with address $e$ but unknown content. The main new operator in separation logic is the *separation conjunction*, written * and pronounced "and separately". The assertion $P * Q$ expresses that the heap can be split into two disjoint parts in which $P$ and $Q$ hold, respectively. Using separation conjunction, one can specify larger parts of the heap. For instance, the assertion $(x \mapsto 21) * (y \mapsto 42)$ describes two separate cells with addresses $x$ and $y$ and contents 21 and 42, respectively. As abbreviation one uses $e \mapsto e_1, \ldots, e_n$ to stand for $e \mapsto e_1 * e+1 \mapsto e_2 * \cdots * e+n-1 \mapsto e_n$ and thus asserting that the heap consists of $n$ adjacent cells with addresses $e, \ldots, e+n-1$ and contents $e_1, \ldots, e_n$, respectively. For example, the assertion $(x \mapsto 21, y) * (y \mapsto 42, x)$ concisely specifies a heap with a cyclic pointer structure, in which $x \mapsto 21, y$ stands for $x \mapsto 21 * x+1 \mapsto y$ and similarly with $y \mapsto 42, x$. It consists of two separate parts of the heap at the addresses $x$ and $y$ that contain 21 and 42, respectively, and a pointer to the other address.

For mutations and lookups the following axioms are stated in [Bro07, O'H19]:

ALLOCATION

$$\{\mathbf{emp}\}\ x := \mathbf{alloc}()\ \{x \mapsto -\}$$

DE-ALLOCATION

$$\{x \mapsto -\}\ \mathbf{free}(x)\ \{\mathbf{emp}\}$$

UPDATE

$$\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}$$

LOOKUP

$$\{P[x := e'] \wedge e \mapsto e'\}\ x := [e]\ \{P \wedge e \mapsto e'\}$$

where $x$ does not occur in $e$ or $e'$.

Separation conjunction enables the formulation of proof rules for local reasoning about components of parallel programs. Crucial is the following rule, which is essentially the INVARIANCE rule in a semantic setting of heaps:

FRAME

$$\frac{\{P\}\ S\ \{Q\}}{\{P * R\}\ S\ \{P * R\}}$$

where $\mathit{free}(R) \cap \mathit{change}(S) = \emptyset$.

In separation logic, this rule serves to extend a local specification involving only the variables and parts of the heap that is used by $S$ by adding assertions about variables and other parts of the heap not modified by $S$. Thus the FRAME rule is considered as the key to local reasoning about the heap [Rey02].

Separation logic was originally used for the verification of sequential programs manipulationg pointer structures. Exploiting its ability to reason explicitly about the heap, the approach was later extended in [O'H07] to reason in a modular way about concurrent programs. The idea is that two threads that operate on disjoint parts of the heap do not interfere, and thus can be verified in isolation. This is captured by the following rule, which is the DISJOINT PARALELISM rule in a semantic setting with heaps:

CONCURRENCY

$$\frac{\{P_1\}\ S_1\ \{Q_2\},\ \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 * P_2\}\ [S_1 \| S_2]\ \{Q_1 * Q_2\}}$$

where $\mathit{free}(P_1, Q_1) \cap \mathit{change}(S_2) = \mathit{free}(P_2, Q_2) \cap \mathit{change}(S_1) = \emptyset$.

In this rule, the separation conjunction $P_1 * P_2$ in the precondition of the parallel composition $[S_1 \| S_2]$ is true if the heap can be partitioned into sub-heaps making the local preconditions of the components $S_1$ and $S_2$ true. The components establish then in their local heap local postconditions which are then combined into the gobal postcondition $Q_1 * Q_2$.

A major issue was developing a coherent semantical model for this *concurrent separation logic*. This was solved by S. Brookes in [Bro07]. His semantics evaluates resource-sensitive partial correctness formulas of the form $\Gamma \vdash \{P\}\ S\ \{Q\}$, where $\Gamma$ is a *resource context* that specifies for each resource name occuring in the program $S$ a finite set of variables, a protection list, and a resource invariant, and a proof system allows one to reason about these formulas.

## 10.6 Relational Hoare logic

We conclude this overview of alternative approaches by discussing a line of research that begins with the work of N. Benton, [Ben04]. In it Hoare's logic was modified to verify correctness of various optimizing program transformations. This was achieved by proposing a proof system in which one reasons at the same time about a pair of programs the variables of which are related by some precondition and postcondition. To this end the customary correctness formulas were replaced by *judgments* concerning two **while** programs, $S_1$ and $S_2$. These are statements of the form

$$S_1 \sim S_2 : \Psi \Rightarrow \Phi,$$

where $\Psi$ and $\Phi$ are relations on program states. Informally, such a judgment states that if initially the relation $\Psi$ between the variables of the programs $S_1$ and $S_2$ holds, then after their independent executions the relation $\Phi$ between their variables holds. (So a notation $\{\Psi\}\ S_1 \sim S_2\ \{\Phi\}$ would have been more intuitive.) Typically both programs use the same variables, so to indicate from which program the variable is taken the indices $\langle 1 \rangle$ and $\langle 2 \rangle$ are used in $\Psi$ and $\Phi$.

As an example consider the technique of *invariant hoisting*. The program

$$S_1 \equiv \textbf{while}\ i < n\ \textbf{do}\ x := y + 1;\ i := i + x\ \textbf{od}$$

can be optimized to

$$S_2 \equiv x := y + 1;\ \textbf{while}\ i < n\ \textbf{do}\ i := i + x\ \textbf{od}.$$

The resulting programs are equivalent if we ignore the variable $x$. This can be expressed as the judgment

$$S_1 \sim S_2 : \Phi \Rightarrow \Phi,$$

with $\Phi \equiv i\langle 1\rangle = i\langle 2\rangle \wedge n\langle 1\rangle = n\langle 2\rangle \wedge y\langle 1\rangle = y\langle 2\rangle$. So the first occurrence of $\Phi$ expresses the information that prior to the programs executions the values of their variables $i, n, y$ respectively equal, while the second occurrence states that the values of these variables respectively equal after the executions of both programs.

In general, if we rename $S_1$ to $S_1'$ by appending to each of its variables the index $\langle 1\rangle$, and similarly with $S_2$ for which we use the index $\langle 2\rangle$, then the judgment $S_1 \sim S_2 : \Psi \Rightarrow \Phi$ can be interpreted as the correctness formula $\{\Psi\}\ S_1';\ S_2'\ \{\Phi\}$.

The resulting logic was called *Relational Hoare Logic* (RHL), where the qualification 'Relational' referred to the fact that instead of assertions relations were used. An example rule is the following one dealing with the conditional statement:

$$\frac{S_1 \sim S_2 : \Psi \Rightarrow \Phi}{\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi} \sim S_1 : \Psi \Rightarrow \Phi}$$

Appropriate proof rules formulated in this framework made it possible to verify various known compiler optimization techniques, including the above example of invariant hoisting.

In [BGB09] this approach was extended to study equivalence of probabilistic programs. The considered programs extend **while** programs by allowing a standard feature of probabilistic programs, called *random sampling*, which is an assignment of the form $x := \mathcal{D}$, where $\mathcal{D}$ is a probability distribution over the values of the type of $x$. The semantics of such programs, as in [dHdV02], is then a mapping from the set of states to the set of distributions over the states.

The semantics of the judgments is defined in such a way that it respects the fact that in both considered programs the same probability distributions are used. In particular the judgment $S \sim S : \textbf{true} \Rightarrow x\langle 1\rangle = x\langle 2\rangle$, where $S \equiv x := \mathcal{D}$, is true.

The resulting logic was called *probabilistic Relational Hoare Logic* (pRHL). pRHL is not a simple extension of RHL because in presence of probabilities some rules of RHL become unsound. For example the rule

$$\frac{S_1 \sim S_2 : \Psi \Rightarrow \Phi_1,\ S_1 \sim S_2 : \Psi \Rightarrow \Phi_2}{S_1 \sim S_2 : \Psi \Rightarrow \Phi_1 \wedge \Phi_2}$$

is sound in RHL but not in pRHL.

This work was carried out in the context of `CertiCrypt`, a system built on top of the theorem prover `Coq`, that makes possible to provide machine-checked correctness proofs of cryptographic algorithms.

In [BKOB13] this framework was further generalized to allow for a probabilistic reasoning about *differential privacy*, a parametrized by two parameters notion of privacy that guarantees that the behaviour of an algorithm taking values from a dataset hardly changes when the dataset is slightly modified. Differential privacy in particular provides a formal guarantee that information about specific participants in a database is not revealed by the algorithm.

The resulting formalism is called *approximate probabilistic Relational Hoare Logic* (apRHL). In apRHL the judgments of [BGB09] were generalized to parametrized judgments concerning two probabilistic programs, $S_1$ and $S_2$. They are of the form

$$S_1 \sim_{\alpha,\delta} S_2 : \Psi \Rightarrow \Phi,$$

where $\alpha \geq 1$, $\delta \in [0,1]$, and $\Psi$ and $\Phi$ are relations on program states.

The appropriate proof rules generalize those of [BGB09]. This framework was implemented in a system called `CertiPriv` built on top of `Coq`, which was in particular used to provide machine-checked proofs of soundness of the considered rules.

This work was further pursued in [BGA+14], where differential privacy was dealt with by means of a transformation of a probabilistic program $S$ into a non-probabilistic one $\lceil S \rceil$ that simulates two executions of the original program. This made it possible to model the judgment $S \sim_{\alpha,\delta} S : \Psi \Rightarrow \Phi$ as $\{\Psi\}\ \lceil S \rceil\ \{\Phi \wedge v_\alpha \leq \alpha \wedge v_\delta \leq \delta\}$, where the relevant parameters $\alpha$ and $\delta$ are maintained in the designated variables $v_\alpha$ and $v_\delta$. As a result differential privacy of a single program could be established using the original proof system $\mathcal{H}$ presented in Subsection 3.1.

# 11 Final Remarks: a Summary and an Assessment

Hoare's logic had a huge impact on program verification, notably by allowing one to approach it in a systematic way, using the logical apparatus of formal proofs. Combining it with the research on program semantics made it possible to argue about the soundness and relative completeness of the underlying proof systems. The syntax-directed form of Hoare's logic suggested a natural research agenda, which —as we have seen— allowed one to deal with several programming constructs and forms of program construction, including higher-order procedures, nondeterminism, concurrency, and object-orientation.

This survey aimed at a systematic exposition of these developments. Because of space considerations we had to omit an account of various Hoare-style proof rules for such concepts as program jumps ([CH72]), go-to statement ([dB80, Chapter 10], written by A. de Bruijn), or several forms of abrupt loop termination present in Java ([HJ00]).

In the seventies Hoare's logic was used to define programming languages. In [Hoa72a] and [HW73] an axiomatic definition of the programming language Pascal was given. These paper provided proof rules for simple constructs such as the **case** statement and the **repeat** statement. However, the presentation was incomplete. For example, no account of reasoning about recursive procedures or pointers was given. This work was pursued in [LGH+78] where axioms and proof rules in Hoare's logic for the programming language Euclid were presented.

From the current perspective one can see that such axiomatic presentations were not rigorous since no soundness proofs were provided to justify the introduced axioms and rules, notably the recursion rule. To see that such soundness proofs are not superfluous recall from Subsection 5.1 the observation of [Old83b] that the adaptation rule for Euclid is not sound. Also, termination was not dealt with in these papers and the reasoning about it can be another source of possible, subtle, errors (see for example the discussion at the end of Subsection 6.1). In fact, as we saw, reasoning about soundness within the framework of Hoare's logic started only after these two papers were published.

But even if such soundness proofs were presented, given the size of the considered programming languages, there would be a non-trivial chance of errors. In fact, we mentioned a number of times that arguments about correctness of various proof rules in Hoare's logic or about soundness or relative completeness of some specific proof systems have led to various, occasionally, pretty subtle errors.

A natural remedy is to use automated reasoning to argue about various Hoare's logics. The first contribution in this direction was [Sok87], where soundness of the original proof system of Hoare from [Hoa69] was established in the LCF system. Next, in [Kle98], [Nip02a] and [Nip02b] soundness and relative completeness

of Hoare's logics for partial and total correctness of **while** programs and programs with recursive procedures was established in the LEGO and Isabelle/HoL interactive theorem provers.

Further, in [NN99] semantics and proof system of [OG76a] for partial correctness of parallel programs, that we discussed in Subsection 8.1, was formalized in the Isabelle/HoL system. Subsequently the authors proved soundness of this proof system and verified a number of correctness proof examples. An analogous formalization was carried out in [Bal06] on the basis of dynamic logic in the KIV system, described in [BRS$^+$00]. A most recent contribution relevant to Hoare's logic is [HN18] in which three Hoare's logics for reasoning about time bounds, including the original logic due to [Nie87], were formalized and shown to be sound and relatively complete.

One should also mention here recent work aimed at supporting teaching of Hoare's logic. In [SS14] an account is given of a tool called HAHA (Hoare Advanced Homework Assistant) that was specifically designed to teach Hoare's logic. The tool supports reasoning about the **while** programs with integer variables and arrays. Further, in [BH16] a tool called KeY-Hoare is discussed. It is built on top of a KeY system, an extensive software development system that supports in particular specification and formal verification of object-oriented software, based on dynamic logic (see [ABB$^+$16]). The KeY-Hoare tool allows one to reason about partial and total correctness of **while** programs in an extension of Hoare's logic with explicit state updates.

In recent years research on Hoare's logic visibly slowed down, probably due to the fact that through hundreds of publications it achieved its main goal of creating a comprehensive formal framework to reason about various classes of programs.

The versatility of Hoare's logic as an approach to program verification can be appreciated by the fact that it was applied also to probabilistic programs and, more recently, to quantum programs, a research direction that originated with the work of M. Ying [Yin11]. For a recent overview of the developments on this subject, that successfully parallel the developments of the customary Hoare's logic, see [Yin19]. A related approach of [Unr19] follows the line of research started with [Ben04] and [BGB09] and introduces *quantum Relational Hoare Logic* (qRHL) that allows one to reason about how the outputs of two quantum programs relate given a relation between their inputs.

# References


[ABB$^+$16] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[ABC87] K. R. Apt, L. Bougé, and P. Clermont. Two normal form theorems for CSP programs. *Information Processing Letters*, 26:165–171, 1987.

[AdB77] K. R. Apt and J. W. de Bakker. Semantics and proof theory of pascal procedures. In A. Salomaa and M. Steinby, editors, *Automata, Languages and Programming: Proceedings of the Fourth Colloquium*, volume 52 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 1977.

[AdB90a] P. America and F. S. de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–162, 1990.

[AdB90b] P. America and F. S. de Boer. A proof system for process creation. In M. Broy, editor, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*, pages 303–332. North-Holland, 1990.

[ÁdBdRS05] E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded java. *Theor. Comput. Sci.*, 331(2-3):251–290, 2005.

[AdBO90] K. R. Apt, F. S. de Boer, and E.-R. Olderog. Proving termination of parallel programs. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business, A Birthday Salute to Edsger W. Dijkstra*, pages 0–6, New York, 1990. Springer-Verlag.

[AdBO09] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York, third edition, 2009.

[AFdR80] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[AFK88] K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. *Distributed Computing*, 2(4):226–241, August 1988.

[AFPdS11] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa. *Rigorous Software Development - An Introduction to Program Verification*. Undergraduate Topics in Computer Science. Springer, 2011.

[AO81] K. R. Apt and E.-R. Olderog. Proof rules dealing with fairness. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 1981.

[AO83] K. R. Apt and E.-R. Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3:65–100, 1983.

[AO91] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York, 1991.

[AP86] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986.

[APS84] K. R. Apt, A. Pnueli, and J. Stavi. Fair termination revisited with delay. *Theoretical Computer Science*, 33:65–84, 1984.

[Apt81] K. R. Apt. Ten years of Hoare's logic, a survey, part I. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.

[Apt83] K. R. Apt. Formal justification of a proof system for Communicating Sequential Processes. *Journal of the ACM*, 30:197–216, 1983.

[Apt84] K. R. Apt. Ten years of Hoare's logic, a survey, part II: nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[Apt86] K. R. Apt. Correctness proofs of distributed termination algorithms. *ACM Transactions on Programming Languages and Systems*, 8:388–405, 1986.

[Bac80] R.-J. R. Back. Correctness preserving program refinements: Proof theory and applications. Technical Report 131, Mathematisch Centrum, Amsterdam, 1980.

[Bac86] R. C. Backhouse. *Program Construction and Verification.* Prentice-Hall International, Englewood Cliffs, N.J., 1986.

[Bal06] M. Balser. *Verifying Concurrent Systems with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction.* PhD thesis, University of Augsburg. Shaker Verlag, 2006.

[Ben04] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 14–25. ACM, 2004.

[Ben12] M. Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition.* Springer, 2012.

[BG87] A. Blass and Y. Gurevich. Existential fixed-point logic. In E. Börger, editor, *Computation Theory and Logic*, pages 20–36. Springer, 1987.

[BGA+14] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P. Strub. Proving differential privacy in Hoare logic. In *IEEE 27th Computer Security Foundations Symposium (CSF 2014)*, pages 411–424. IEEE Computer Society, 2014.

[BGB09] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[BH16] R. Bubel and R. Hähnle. KeY-Hoare. In *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, pages 571–589. Springer, 2016.

[BKOB13] G. Barthe, B. Köpf, F. Olmedo, and S. Z. Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9:1–9:49, 2013.

[Bro07] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development in KIV. In T. Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 2000.

[BT82] J. A. Bergstra and J. V. Tucker. Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs. *Theor. Comput. Sci.*, 17:303–315, 1982.

[BvW08] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, New York, 2008.

[CDE+16] R. Certezeanu, S. Drossopoulou, B. Egelund-Müller, K. R. M. Leino, S. Sivarajan, and M. J. Wheelhouse. Quicksort revisited - verifying alternative versions of quicksort. In E. Ábrahám, M. M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 2016.

[CGH83] E. M. Clarke, S. M. German, and J. Y. Halpern. Effective axiomatizations of Hoare logics. *J. ACM*, 30(3):612–636, 1983.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In D. Agnew, L. J. M. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications (CHDL '93)*, volume A-32 of *IFIP Transactions*, pages 15–30. North-Holland, 1993.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CH72] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Inf.*, 1:214–224, 1972.

[CHVB18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[Cla76] E. M. Clarke. *Completeness and incompleteness theorems for Hoare-like axiom systems*. PhD thesis, Computer Science Department, Cornell University, USA, 1976.

[Cla79] E. M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM*, 26(1):129–147, January 1979.

[Cla85] E. M. Clarke. The characterization problem for Hoare logics. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 89–106, Englewood Cliffs, N.J., 1985. Prentice-Hall International.

[Cli73] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2:50–63, 1973.

[CO81] R. Cartwright and D. C. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[Coo81] S. A. Cook. Corrigendum: Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 10(3):612, 1981.

[Cou90] P. Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 841–994. Elsevier, 1990.

[CPR11] B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.

[dB75] J. W. de Bakker. Inleiding bewijsmethoden. In *Colloquium Programmcorrectheid*, MC Syllabus 21, pages 3–17. Mathematisch Centrum, Amsterdam, 1975.

[dB80] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, N.J., 1980.

[dB91] F. S. de Boer. A compositional proof system for dynamic process creation. In *LICS*, pages 399–405. IEEE Computer Society, 1991.

[dB99] F. S. de Boer. A WP-calculus for OO. In *FoSSaCS*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999.

[dBP03] F. S. de Boer and C. Pierik. How to cook a complete Hoare logic for your pet OO language. In *FMCO*, volume 3188 of *Lecture Notes in Computer Science*, pages 111–133. Springer, 2003.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., 1972.

[DF88] E. W. Dijkstra and W. H. J. Feijen. *A method of programming*. Addison-Wesley, 1988.

[dGR16] S. de Gouw and J. Rot. Effectively eliminating auxiliaries. In E. Ábrahám, M. M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2016.

[dHdV02] J. den Hartog and E. P. de Vink. Verifying probabilistic programs using a Hoare like logic. *Int. J. Found. Comput. Sci.*, 13(3):315–340, 2002.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112, London, 1968. Academic Press.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[Dij76a] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Dij76b] E. W. Dijkstra. A great improvement. Available at `http://www.cs.utexas.edu/users/EWD/ewd05xx/EWD573.PDF`, published as [Dij82], July 1976.

[Dij82] E. W. Dijkstra. A great improvement. In *Selected Writings on Computing: A Personal Perspective*, pages 217–219. Springer-Verlag, 1982.

[DJ83] W. Damm and B. Josko. A sound and relatively complete Hoare-logic for a language with higher type procedures. *Acta Informatica*, 20:59–101, 1983.

[dMB08] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dRdBH+01] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York, 1990.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[Eng17] B. Engelmann. *Techniques for the verification of dynamically typed programs*. PhD thesis, University of Oldenburg, Germany, 2017.

[EO16] B. Engelmann and E.-R. Olderog. A sound and complete Hoare logic for dynamically-typed, object-oriented programs. In E. Ábrahám, M. M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *Lecture Notes in Computer Science*, pages 173–193. Springer, 2016.

[FH71] M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *Computer Journal*, 14(4):391–395, 1971.

[Fil07] J.-C. Filliâtre. Formal proof of a program: Find. *Sci. Comput. Program.*, 64(3):332–340, 2007.

[Flo67] R. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science*, pages 19–32, American Mathematical Society, New York, 1967.

[FMV14] C. A. Furia, B. Meyer, and S. Velder. Loop invariants: Analysis, classification, and examples. *ACM Comput. Surv.*, 46(3):34:1–34:51, 2014.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, New York, 1986.

[Fra92] N. Francez. *Program Verification*. Addison-Wesley, Reading, MA, 1992.

[GCH89] S. M. German, E. M. Clarke, and J. Y. Halpern. Reasoning about procedures as parameters in the language L4. *Inf. Comput.*, 83(3):265–359, 1989.

[GFK84] O. Grumberg, N. Francez, and S. Katz. Fair termination of communicating processes. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*, pages 254–265. ACM, 1984.

[GFMdR81] O. Grumberg, N. Francez, J. A. Makowsky, and W. P. de Roever. A proof rule for fair termination of guarded commands. In J. v. V. J.W. de Bakker, editor, *Proceedings of the International Symposium on Algorithmic Languages*, pages 339–416, 1981.

[GFMdR85] O. Grumberg, N. Francez, J. A. Makowsky, and W. P. de Roever. A proof rule for fair termination of guarded commands. *Information and Control*, 66(1/2):83–102, July/August 1985.

[GL80] D. Gries and G. Levin. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, 2(4):564–579, 1980.

[Gor75] G. A. Gorelick. A complete axiomatic system for proving assertions about recursive and nonrecursive programs. Technical Report 75, Department of Computer Science, University of Toronto, 1975. Available at `https://archive.org/details/ACompleteAxiomaticSystemForProvingAssertionsAboutRecursiveAnd`.

[Gri78] D. Gries. The multiple assignment statement. *IEEE Transactions on Software Engineering*, SE-4:89–93, March 1978.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Har79] D. Harel. *First-Order Dynamic Logic*. Lecture Notes in Computer Science 68, Springer-Verlag, New York, 1979.

[HdR86] J. Hooman and W. P. de Roever. The quest goes on: a survey of proofsystems for partial correctness of CSP. In *Current Trends in Concurrency*, pages 343–395, New York, 1986. Lecture Notes in Computer Science 224, Springer-Verlag.

[HJ00] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering, Third Internationsl Conference, FASE 2000*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic logic*. MIT Press, Cambridge, MA, 2000.

[HL74] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Inf.*, 3:135–153, 1974.

[HN18] M. P. L. Haslbeck and T. Nipkow. Hoare logics for time bounds. *Archive of Formal Proofs*, February 2018. `http://isa-afp.org/entries/Hoare_Time.html`, Formal proof development.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, 1969.

[Hoa71a] C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In E. Engeler, editor, *Proceedings of Symposium on the Semantics of Algorithmic Languages*, pages 102–116, New York, 1971. Lecture Notes in Mathematics 188, Springer-Verlag.

[Hoa71b] C. A. R. Hoare. Proof of a program: FIND. *Commun. ACM*, 14(1):39–45, 1971.

[Hoa72a] C. A. R. Hoare. An axiomatic definition of the programming language PASCAL. In *International Sympoisum on Theoretical Programming*, volume 5 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1972.

[Hoa72b] C. A. R. Hoare. Proof of a structured program: 'the sieve of Eratosthenes'. *Comput. J.*, 15(4):321–325, 1972.

[Hoa72c] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrot, editors, *Operating Systems Techniques*, pages 61–71, London, 1972. Academic Press.

[Hoa75] C. A. R. Hoare. Parallel programming: An axiomatic approach. *Comput. Lang.*, 1(2):151–160, 1975.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[HOP10] J. Hoenicke, E.-R. Olderog, and A. Podelski. Fairness for dynamic control. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, (TACAS 2010)*, volume 6015 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2010.

[HP15] J. Hoenicke and A. Podelski. Fairness for infinitary control. In R. Meyer, A. Platzer, and H. Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday.*, volume 9360 of *Lecture Notes in Computer Science*, pages 33–43. Springer, 2015.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[INM84] INMOS Limited. *Occam Programming Manual*. Prentice-Hall International, Englewood Cliffs, N.J., 1984.

[Jon81] C. B. Jones. *Developing methods for computer programs including a notion of interference*. PhD thesis, University of Oxford, UK, 1981.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[JR10] C. B. Jones and A. W. Roscoe. Insight, inspiration and collaboration. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare.*, pages 1–32. Springer, 2010.

[JW75] K. Jensen and N. Wirth. *PASCAL User Manual and Report*. Springer, 1975.

[Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall International, Englewood Cliffs, N.J., 1990.

[KGJ+15] J. Katoen, F. Gretz, N. Jansen, B. L. Kaminski, and F. Olmedo. Understanding probabilistic programs. In R. Meyer, A. Platzer, and H. Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday.*, volume 9360 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2015.

[Kin69] J. King. *Developing methods for computer programs including a notion of interference*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, USA, 1969.

[Kle98] T. Kleymann. *Hoare logic and VDM : machine-checked soundness and completeness proofs*. PhD thesis, University of Edinburgh, UK, 1998.

[Kle99] T. Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:2:125–143, 1977.

[Lan79] H. Langmaack. A proof of a theorem of Lipton on Hoare logic and applications. Technical report, Ber. 8003, Inst. Inf. Prakt. Math., University of Kiel, Germany, 1979.

[Lan82] H. Langmaack. On the termination problem for finitely interpreted ALGOL-like programs. *Acta Inf.*, 18:79–108, 1982.

[Lau71] P. E. Lauer. Consistent formal theories of the semantics of programming languages. Technical Report 25. 121, IBM Laboratory Vienna, 1971.

[Lei10] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[LG81] G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[LGH+78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Inf.*, 10:1–26, 1978.

[Lip77] R. J. Lipton. A necessary and sufficient condition for the existence of Hoare logics. In *18th Annual Symposium on Foundations of Computer Science*, pages 1–6. IEEE Computer Society, 1977.

[LO80] H. Langmaack and E.-R. Olderog. Present-day Hoare-like systems for programming languages with procedures: Power, limits and most likely expressions. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium*, volume 85 of *Lecture Notes in Computer Science*, pages 363–373. Springer, 1980.

[LPS81] D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness: the ethics of concurrent termination. In O. Kariv and S. Even, editors, *Proceedings of International Colloquium on Automata Languages and Programming (ICALP '81)*, pages 264–277, New York, 1981. Lecture Notes in Computer Science 115, Springer-Verlag.

[LS87] J. Loeckx and K. Sieber. *The Foundation of Program Verification.* Teubner-Wiley, Stuttgart, second edition, 1987.

[Man74] Z. Manna. *Mathematical Theory of Computation.* Mc Graw-Hill, 1974.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.

[MM05] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Monographs in Computer Science. Springer, 2005.

[Moi83] A. Moitra. On Apt, Francez, and de Roever's "A Proof System for Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems*, 5(3):500–501, 1983.

[Mor94] C. Morgan. *Programming from Specifications*. Prentice-Hall International, London, second edition, 1994.

[MP74] Z. Manna and A. Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:253–263, 1974.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer-Verlag, New York, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems – Safety*. Springer-Verlag, New York, 1995.

[MS87] C. Mirkowska and A. Salwicki. *Algorithmic Logic*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[NBB+63] P. Naur, J. Backus, F. Bauer, J. Green, C. Katz, J. McCarthy, A. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Numerische Mathematik*, 4:420–453, 1963.

[Nie87] H. R. Nielson. A Hoare-like proof system for analysing the computation time of programs. *Sci. Comput. Program.*, 9(2):107–136, 1987.

[Nip02a] T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, volume 62 of *NATO Science Series*, pages 341–367. Springer, 2002.

[Nip02b] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Proceedings of the Computer Science Logic, 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2002.

[NN99] T. Nipkow and L. P. Nieto. Owicki/Gries in Isabelle/HOL. In J. P. Finance, editor, *Fundamental Approaches in Software Enginering (FASE)*, volume 1577 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 1999.

[OA88] E.-R. Olderog and K. R. Apt. Fairness in parallel programs, the transformational approach. *ACM Transactions on Programming Languages and Systems*, 10:420–455, 1988.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, 1976.

[O'H07] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[O'H19] P. W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019.

[Old81] E.-R. Olderog. Sound and complete Hoare-like calculi based on copy rules. *Acta Informatica*, 16:161–197, 1981.

[Old83a] E.-R. Olderog. A characterization of Hoare's logic for programs with Pascal-like procedures. In *Proc. of the 15th ACM Symp. on Theory of Computing (STOC)*, pages 320–329. ACM, April 1983.

[Old83b] E.-R. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 30:337–347, 1983.

[Old84] E.-R. Olderog. Correctness of programs with Pascal-like procedures without global variables. *Theoretical Computer Science*, 30:49–90, 1984.

[OP10] E.-R. Olderog and A. Podelski. Explicit fair scheduling for dynamic control. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 96–117. Springer, 2010.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE-11), 11th International Conference on Automated Deduction, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[ORY01] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic, 15th International Workshop (CSL 2001)*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1975.

[Owi76] S. Owicki. A consistent and complete deductive system for the verification of parallel programs. In *STOC*, pages 73–86. ACM, 1976.

[Owi78] S. Owicki. Verifying concurrent programs with shared data classes. In E. J. Neuhold, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, pages 279–298, Amsterdam, 1978. North-Holland.

[OYR04] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In N. D. Jones and X. Leroy, editors, *Proc. 31st Symp. on Principles of Programming Languages (POPL 2004)*, pages 268–280. ACM, 2004.

[PdB03] C. Pierik and F. S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2003.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pra76] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science (FoCS 1976*, pages 109–121. IEEE Computer Society, 1976.

[QS81] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, Paris, 1981.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[Sal70] A. Salwicki. Formalized algorithmic languages. *Bulletin de l'Academie Polonaise des Sciences*, 18:227–232, 1970.

[Sch77] J. Schwarz. Generic commands – a tool for partial correctness formalisms. *The Computer Journal*, 20:151–155, 1977.

[Sok77] S. Sokołowski. Total correctness for procedures. In *6th Symposium on Mathematical Foundations of Computer Science*, volume 53 of *Lecture Notes in Computer Science*, pages 475–483. Springer, 1977.

[Sok87] S. Sokołowski. Soundness of Hoare's logic: An automated proof using LCF. *ACM Transactions on Programming Languages and Systems*, 9(1):100–120, 1987.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[SS14] T. Sznuk and A. Schubert. Tool support for teaching Hoare logic. In D. Giannakopoulou and G. Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference (SEFM 2014)*, volume 8702 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2014.

[Tar36] A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1(3):261—-405, 1936.

[Tur49] A. M. Turing. On checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949. Univ. Math. Laboratory, Cambridge, 1949. (See also: F. L. Morris and C. B. Jones, *An early program proof by Alan Turing*, *Annals of the History of Computing 6* pages 139–143, 1984).

[TZ88] J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*. North-Holland and CWI Monographs, Amsterdam, 1988.

[Unr19] D. Unruh. Quantum relational Hoare logic. *Proc. ACM Program. Lang.*, 3(POPL):33:1–33:31, 2019.

[vWMP+75] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Inf.*, 5:1–236, 1975.

[Wan78] M. Wand. A new incompleteness result for Hoare's system. *J. ACM*, 25(1):168–175, 1978.

[WH66] N. Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, 1966.

[Win93] G. Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993.

[Yin11] M. Ying. Floyd–Hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems*, 33(6):19:1–19:49, 2011.

[Yin19] M. Ying. Toward automatic verification of quantum programs. *Formal Asp. Comput.*, 31(1):3–25, 2019.

[Zöb88] D. Zöbel. Normalform-Transformationen für CSP-Programme. *Informatik: Forschung und Entwicklung*, 3:64–76, 1988.

# A Turing's example

We present here in Figure 6 a proof outline for a **while** program corresponding to Turing's example given in Figure 3 of Subsection 2.1. The qualification **inv** is used to annotate the loop invariants and **bf** to annotate the termination functions. So this is a proof outline establishing total correctness of the program w.r.t. the precondition $n \geq 1$ and postcondition $v = n!$.

$\{n \geq 1\}$
$r := 1;$
$u := 1;$
$v := u;$
$\{\mathbf{inv} : P_1 \equiv v = r! \wedge u = r! \wedge 1 \leq r \leq n\}$
$\{\mathbf{bd} : t_1 \equiv n - r\}$
**while** $r < n$ **do**
$\{P_1 \wedge r < n\}$
$\{v = r! \wedge u + v = 2 \cdot v \wedge 1 \leq 2 \leq r + 1 \leq n\}$
$s := 1;$
$u := u + v;$
$s := s + 1;$
$\{\mathbf{inv} : P_2 \equiv v = r! \wedge u = s \cdot v \wedge 1 \leq s \leq r + 1 \leq n\}$
$\{\mathbf{bd} : t_2 \equiv r + 1 - s\}$
**while** $s \leq r$ **do**
$\{P_2 \wedge s \leq r\}$
$\{v = r! \wedge u + v = (s + 1) \cdot v \wedge 1 \leq s + 1 \leq r + 1 \leq n\}$
$u := u + v;$
$s := s + 1$
$\{P_2\}$
**od**
$\{P_2 \wedge s > r\}$
$\{v = r! \wedge u = s \cdot v \wedge 1 \leq s = r + 1 \leq n\}$
$\{u = (r + 1) \cdot r! \wedge 1 \leq r + 1 \leq n\}$
$\{u = (r + 1)! \wedge 1 \leq r + 1 \leq n\}$
$r := r + 1;$
$v := u$
$\{P_1\}$
**od**
$\{P_1 \wedge r \geq n\}$
$\{v = r! \wedge r \leq n \wedge r \geq n\}$
$\{v = n!\}$

Figure 6: Turing's example as a proof outline of a corresponding **while** program.