Dijkstra, Hoare and Parnas

12

Key Topics

Algol 60
Axiomatic semantics
Calculus of weakest preconditions
Communicating sequential processes
Graph algorithms
Operating systems
Predicate calculus
Tabular expressions
Normal table

12.1 Introduction

Edsger W. Dijkstra, C.A.R. Hoare and David Parnas are famous names in computer science, and they have received numerous awards for their contribution to the discipline. Their work has provided a scientific basis for computer software development and a rigorous approach to the development of software. We present a selection of their contributions in this chapter, including Dijkstra's calculus of weakest preconditions; Hoare's axiomatic semantics and Parnas's tabular expressions. There is more detailed information on the contributions of these pioneers in [1].

Mathematics and Computer Science were regarded as two completely separate disciplines in the 1960s, and software development was based on the assumption that the completed code would always contain defects. It was therefore better and more productive to write the code as quickly as possible and to then perform debugging to find the defects. Programmers then corrected the defects, made

G. O'Regan, *Concise Guide to Formal Methods*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-64021-1_12

Fig. 12.1 Edsger Dijkstra. Courtesy of Brian Randell

patches and retested and found more defects. This continued until they could no longer find defects. Of course, there was always the danger that defects remained in the code that could give rise to software failures.

John McCarthy argued at the IFIP congress in 1962 that the focus should instead be to prove that the programs have the desired properties, rather than testing the program *ad nauseum*. Robert Floyd believed that there was a way to construct a rigorous proof of the correctness of the programs using mathematics, and he demonstrated techniques (based on assertions) in a famous paper in 1967 that mathematics could be used for program verification. The NATO conference on software engineering in 1968 highlighted the extent of the problems that existed with software, and the term "*software crisis"* was coined to describe this. The problems included cost and schedule overruns and problems with the reliability of the software.

Dijkstra (Fig. 12.1) was born in Rotterdam in Holland, and he studied mathematics and physics at the University of Leyden. He obtained a PhD in Computer Science from the University of Amsterdam in 1959. He decided not to become a theoretical physicist, as he believed that programming offered a greater intellectual challenge.

Table 12.1 Dijkstra's achievements

Area	Description
Go to statement	Dijkstra argued against the use of the goto statement in programming. This eventually led to its abolition in programming
Graph algorithms	Dijkstra developed several efficient graph algorithms to determine the *shortest* or *longest* paths from a vertex u to vertex v in a graph
Operating systems	Dijkstra introduced ideas such as semaphores and deadly embrace, and that operating systems can be built as synchronized sequential processes
Algol 60	Dijkstra contributed to the definition of the language, and he designed and coded the first Algol 60 compiler
Formal program development (guarded commands and predicate transformers)	Dijkstra introduced guarded commands and predicate transformers as a means of defining the semantics of a programming language. He showed how weakest preconditions can be used as a calculus (*wp*-calculus) to develop reliable programs. This led to a science of programming using mathematical logic as a methodology for formal program construction His approach involves the development of programs from mathematical axioms

He commenced his programming career at the Mathematics Centre in Amsterdam in the early 1950s, and he invented the shortest path algorithm in the mid-1950s. He contributed to the definition of Algol 60, and he designed and coded the first Algol 60 compiler.

Dijkstra has made many contributions to computer science, including contributions to language development, operating systems, formal program development and to the vocabulary of Computer Science. He received the Turing award in 1972, and some of his achievements are listed in Table 12.1.

Dijkstra advocated simplicity, precision and mathematical integrity in his formal approach to program development. He insisted that programs should be composed correctly using mathematical techniques and not debugged into correctness. He considered testing to be an inappropriate means of building quality into software, and his statement on software testing is well known:

> Testing a program shows that it contains errors never that it is correct.[1]

Dijkstra corresponded with other academics through an informal distribution network known as the EWD series. These contain his various personal papers including trip reports and technical papers.

[1]Software testing is an essential part of the software process, and various types of testing are described in [2]. Modern software testing is quite rigorous and can provide a high degree of confidence that the software is fit for use. It cannot, of course, build quality in; rather, it can provide confidence that quality has been built in. The analysis of the defects identified during testing may be useful in improving the software development process.

Fig. 12.2 C.A.R Hoare

Charles Anthony Richard (C.A.R or Tony) Hoare studied philosophy (including Latin and Greek) at Oxford University (Fig. 12.2). He studied Russian at the Royal Navy during his National Service in the late 1950s. He then studied statistics and went to Moscow University as a graduate student to study machine translation of languages and probability theory. He discovered the well-known sorting algorithm "*Quicksort*", while investigating efficient ways to look up words in a dictionary.

He returned to England in 1960 and worked as a programmer for Elliot Brothers (a company that manufactured scientific computers). He led a team to produce the first commercial compiler for Algol 60, and it was a very successful project. He then led a team to implement an operating system, and the project was a disaster. He managed a recovery from the disaster and then moved into the research division of the company.

He took a position at Queens University in Belfast in 1968, and his research goals included examining techniques to assist with the implementation of operating systems, especially to see if advances in programming methodologies could assist with the problems of concurrency. He also published material on the use of assertions to prove program correctness.

He moved to Oxford University in 1977 following the death of Christopher Strachey (well known for his work in denotational semantics) and built up the programming research group. This group later developed the Z specification language and CSP, and Hoare received the ACM Turing award in 1980. Following his

Table 12.2 Hoare's achievements

Area	**Description**
Quicksort	Quicksort is a highly efficient sorting algorithm
Axiomatic semantics	Hoare defined a small programming language in terms of axioms and logical inference rules for proving partial correctness of programs
Communicating Sequential Processes (CSP)	CSP is a mathematical approach to the study of communication and concurrency. It is applicable to the specification and design of computer systems that continuously interact with their environment

retirement from Oxford, he took up a position as senior researcher at Microsoft Research in the UK.

Hoare has made many contributions to computer science and these include the quicksort algorithm, the axiomatic approach to program semantics, and programming constructs for concurrency (Table 12.2). He remarked on the direction of the Algol programming language:

> Algol 60 was a great achievement in that it was a significant advance over most of its successors.

Hoare has made fundamental contributions to programming languages, and his 1980 ACM Lecture on the "*Emperors Old Clothes*" is well known. He stresses the importance of communicating ideas (as well as having ideas) and enjoys writing (and rewriting).

David L. Parnas (Fig. 12.3) has been influential in the computing field, and his ideas on the specification, design, implementation remain important. He has won numerous awards (including ACM best paper award in 1979); influential paper awards from ICSE; the ACM SigSoft outstanding researcher award and honorary doctorates for his contribution to Computer Science.

He studied at Carnegie Mellon University and was awarded B.S., M.S., and PhD degrees in Electrical Engineering by the university. He has worked in both industry and academia, and his approach aims to achieve a middle way between theory and practice. His research has focused on real industrial problems that engineers face and on finding solutions to these practical problems. Several organizations such as Phillips in the Netherlands; the Naval Research Laboratory (NRL) in Washington; IBM Federal Systems Division and the Atomic Energy Board of Canada have benefited from his advice and expertise.

He advocates a solid engineering approach to the development of high-quality software and argues that software engineers[2] today do not have the right engineering education to perform their roles effectively. The role of engineers is to

[2]Parnas argues that the term engineer' should be used only in its classical sense as a person who is qualified and educated in science and mathematics to design and inspect products. The evolution of language that has led to a debasement of the term 'engineer' with various groups who do not have the appropriate background to be considered 'engineers' in the classical sense applying this title.

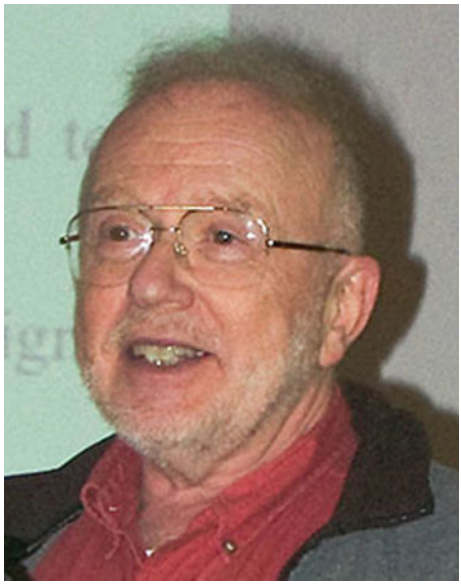

Fig. 12.3 David Parnas

apply scientific principles and mathematics to design and develop useful products. He argues that the level of mathematics taught in most Computer Science courses is significantly less than that taught to traditional engineers. In fact, computer science graduates often enter the work place with knowledge of the latest popular technologies, but with only a limited knowledge of the foundations needed to be successful in producing safe and useful products. Consequently, he argues that it should not be surprising that the quality of software produced today falls below the desired standard, as the current approach to software development is informal and based on intuition rather than sound engineering principles. He argues that computer scientists should be educated as engineers and provided with the right scientific and mathematical background to do their work effectively.

Parnas has made a strong contribution to software engineering, including contributions to requirements specification, software design, software inspections, testing, tabular expressions, predicate logic and ethics for software engineers (Table 12.3). His reflections on software engineering remain valuable and contain the insight gained over a long career.

12.2 Calculus of Weakest Preconditions

The weakest precondition calculus was developed by Dijkstra [4] and applied to the formal development of programs. This section is based on material from [5], and a programming notation is introduced and defined in terms of the weakest precondition. The weakest precondition $wp(S, R)$ is a predicate that describes a set of

Table 12.3 Parnas's achievements

Area	Description
Tabular expressions	Tabular expressions are mathematical tables that are employed for specifying requirements. They enable complex predicate logic expressions to be represented in a simpler form
Mathematical documentation	He advocates the use of mathematical documents for software engineering that are precise and complete. These documents are for system requirements, system design, software requirements, module interface specification and module internal design
Requirements specification	His approach to requirements specification (developed with Kathryn Heninger and others) involves the use of mathematical relations to specify the requirements precisely
Software design	His contribution to software design was revolutionary. A module is characterized by its knowledge of a design decision (secret) that it hides from all others. This is known as the information hiding principle, and it allows software to be designed for changeability. Every information-hiding module has an interface that provides the only means to access the services provided by the modules. The interface hides the module's implementation. Information hiding is used in object-oriented programming
Software inspections	His approach to software inspections is quite distinct from the well-known Fagan inspection methodology. The reviewers are required to take an active part in the inspection and are provided with a list of questions by the author. The reviewers are required to provide documentation of their analysis to justify the answers to the individual questions. This involves the production of mathematical tables
Predicate logic	He introduced a novel approach to deal with undefined values[a] in predicate logic expressions which preserves the two-valued logic. His approach is quite distinct from the logic of partial functions developed by Cliff Jones [3]
Industry contributions	His industrial contribution is impressive including work on defining the requirements of the A7 aircraft and the inspection of safety critical software for the automated shutdown of the nuclear power plant at Darlington
Ethics for software engineers	He has argued that software engineers have a professional responsibility to build safe products, to accept individual responsibility for their design decisions, and to be honest about current software engineering capabilities. He applied these principles in arguing against the strategic defence initiative (SDI) of the Reagan administration in the mid 1980s

[a]His approach allows undefinedness to be addressed in predicate calculus while maintaining the two-valued logic. A primitive predicate logic expression that contains an undefined term is considered false in the calculus, and this avoids the three-valued logics developed by Jones and Dijkstra

states, and it is a function with two arguments that results in a predicate. The function has two arguments (a command and a predicate), where the predicate argument describes the set of states satisfying R after the execution of the command. It is defined as follows:

Definition (*Weakest Precondition*)
The predicate $wp(S, R)$ represents the set of all states such that, if execution of S commences in any one of them, then it is guaranteed to terminate in a state satisfying R.

Let S be the assignment command $i := i + 5$, and let R be $i \leq 3$ then

$$wp(i := i + 5;\ i \leq 3) = (i \leq -2)$$

The weakest precondition $wp(S, T)$ represents the set of all states such that if execution of S commences in any one of them, then it is guaranteed to terminate.

$$wp(i := i + 5;\ T) = T$$

The weakest precondition $wp(S, R)$ is a precondition of S with respect to R, and it is also the weakest such precondition. Given another precondition P of S with respect to R, then $P \Rightarrow wp(S, R)$.

For a fixed command S then $wp(S, R)$ can be written as a function of one argument: $wp_S(R)$, and the function wp_S transforms the predicate R to another predicate $wp_S(R)$. In other words, the function wp_S acts as a *predicate transformer*.

An imperative program may be regarded as a predicate transformer. This is since a predicate P characterizes the set of states in which the predicate P is true, and an imperative program may be regarded as a binary relation on states, leading to the Hoare triple $P\{F\}Q$. That is, the program F acts as a predicate transformer. The predicate P may be regarded as an input assertion, i.e. a predicate that must be true before the program F is executed. The predicate Q is the output assertion, and is true if the program F terminates, having commenced in a state satisfying P.

12.2.1 Properties of WP

The weakest precondition $wp(S, R)$ has several well-behaved properties as described in Table 12.4.

12.2.2 WP of Commands

The weakest precondition can be used to provide the definition of commands in a programming language. The commands considered are taken from [5].

- **Skip Command**

$$wp(skip, R) = R$$

Table 12.4 Properties of WP

Property	Description
Law of the excluded miracle $wp(S, F) = F$	This describes the set of states such that if execution commences in one of them, then it is guaranteed to terminate in a state satisfying false. However, no state ever satisfies false, and therefore $wp(S, F) = F$. The name of this law derives from the fact that it would be a miracle if execution could terminate in no state
Distributivity of conjunction $wp(S, Q) \wedge wp(S, R) = wp(S, Q \wedge R)$	This property stipulates that the set of states such that if execution commences in one of them, then it is guaranteed to terminate in a state satisfying $Q \wedge R$ is precisely the set of states such that if execution commences in one of them then execution terminates with both Q and R satisfied
Law of monotonicity $Q \Rightarrow R$ then $wp(S, Q) \Rightarrow wp(S, R)$	This property states that if a postcondition Q is stronger than a postcondition R, then the weakest precondition of S with respect to Q is stronger than the weakest precondition of S with respect to R
Distributivity of disjunction $wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q \vee R)$	This property states that the set of states corresponding to the weakest precondition of S with respect to Q or the set of states corresponding to the weakest precondition of S with respect to R is stronger than the weakest precondition of S with respect to $Q \vee R$ Equality holds for distributivity of disjunction only when the execution of the command is deterministic

The *skip* command does nothing and is used to explicitly say that nothing should be done. The predicate transformer wp_{skip} is the identity function.

- **Abort Command**

$$wp(abort, R) = F$$

The *abort* command is executed in a state satisfying false (i.e. no state). This command should never be executed. If program execution reaches a point where *abort* is to be executed then the program is in error and abortion is called for.

- **Sequential Composition**

$$wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$$

The sequential composition command composes two commands S_1 and S_2 by first executing S_1 and then executing S_2. Sequential composition is expressed by $S_1; S_2$.

Sequential composition is associative:

$$wp(S_1; (S_2; S_3), R) = wp((S_1; S_2); S_3, R)$$

- **Simple Assignment Command**

$$wp(x := e, R) = dom(e)\ \textbf{cand}\ R_e^x$$

The execution of the *assignment* command consists of evaluating the value of the expression e and storing its value in the variable x. However, the command may be executed only in a state where e may be evaluated.

The expression R_e^x denotes the expression obtained by substituting e for all free occurrences of x in R. For example,

$$(x + y > 2)_v^x = v + y > 2$$

The **cand** operator is used to deal with undefined values, and it was discussed in Chap. 7. It is a non-commutative operator and the expression a **cand** b is equivalent to:

$$a\ \textbf{cand}\ b \cong \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ F$$

The explanation of the definition of the weakest precondition of the assignment statement $wp(x := e, R)$ is that R will be true after execution if and only if the predicate R with the value of x replaced by e is true before execution (since x will contain the value of e after execution).

Often, the domain predicate $dom(e)$ that describes the set of states that e may be evaluated is omitted as assignments are usually written in a context in which the expressions are defined.

$$wp(x := e, R) = R_e^x$$

The simple assignment can be extended to a multiple assignment to simple variables. The assignment is of the form $x_1,x_2,..x_n := e_1,e_2,..e_n$ and is described in [5].

- **Assignment to Array Element Command**

$$wp(b[i] := e, R) = inrange(b, i)\ \textbf{cand}\ dom\,(e)\ \textbf{cand}\ R_{(b;i:e)}^b$$

The execution of the *assignment to an array element* command consists of evaluating the expression e and storing its value in the array element subscripted by i. The *inrange* (b, i) and *dom*(e) are usually omitted in practice as assignments are usually written in a context in which the expressions are defined and the subscripts are in range. Therefore, the weakest precondition is given by:

$$wp(b[i] := e, R) = R^{b}_{(b;i:e)}$$

The notation ($b;i:e$) denotes an array identical to array b except that the array element subscripted by i contains the value e. The explanation of the definition of the weakest precondition of the assignment statement to an array element ($wp(b[i] := e, R)$ is that R will be true after execution if and only if the value of b replaced by ($b;i:e$) is true before execution (since b will become ($b;i:e$) after execution).

- **Alternate Command**

$$wp(IF, R) = dom(B_1 \vee B_2 \vee .. \vee B_n) \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \\ \wedge (B_1 \Rightarrow wp(S_1, R)) \wedge (B_2 \Rightarrow wp(S_2, R)) \wedge \ldots \wedge (B_n \Rightarrow wp(S_n, R))$$

The *alternate* command is the familiar **if** statement of programming languages. The general form of the alternate command is:

If $B_1 \rightarrow S_1$
$\square$ $B_2 \rightarrow S_2$
$\ldots$
$\square$ $B_n \rightarrow S_n$
fi

Each $B_i \rightarrow S_i$ is a guarded command (S_i is any command). The guards must be well defined in the state where execution begins, and at least one of the guards must be true or execution aborts. If at least one guard is true, then one guarded command $B_i \rightarrow S_i$ with true guard B_i is chosen and S_i is executed.

For example, in the if statement below, the statement $z := x + 1$ is executed if $x > 2$, and the statement $z := x + 2$ is executed if $x < 2$. For $x = 2$ either (but not both) statements are executed. This is an example of non-determinism.

if $x \geq 2 \rightarrow z := x + 1$
$\square\, x \leq 2 \rightarrow z := x + 2$
fi

- **Iterative Command**

The *iterate* command is the familiar while loop statement of programming languages. The general form of the iterate command is:

$$\begin{array}{ll} \textbf{do} & B_1 \rightarrow S_1 \\ \square & B_1 \rightarrow S_1 \\ & \ldots \\ \square & B_n \rightarrow S_n \\ \textbf{od} & \end{array}$$

The meaning of the iterate command is that a guard B_{i} is chosen that is true, and the corresponding command S_i is executed. The process is repeated until there are no more true guards. Each choice of a guard and execution of the corresponding statement is an iteration of the loop. On termination of the iteration command all of the guards are false.

The meaning of the DO command $wp(DO, R)$ is the set of states in which execution of DO terminates in a bounded number of iterations with R true.

$$wp(DO, R) = (\exists k : 0 \leq k : H_k(R))$$

where $H_k(R)$ is defined as:

$$H_k(R) = H_0(R) \vee wp(\mathrm{IF}, H_{k-1}(R))$$

A more detailed explanation of loops is in [5]. The definition of procedure call may be given in weakest preconditions also.

12.2.3 Formal Program Development with WP

The use of weakest preconditions for formal program development is described in [5]. The approach is a radical departure from current software engineering, and it involves developing the program and a formal proof of its correctness together. A program P is correct with respect to a precondition Q and a postcondition R if $\{Q\}P\{R\}$, and the idea is that the program and its proof should be developed together. The proof involves weakest preconditions and uses the formal definition of the programming constructs (e.g. assignment and iteration) as discussed earlier.

Programming is viewed as a goal-oriented activity in that the desired result (i.e. the postcondition R) plays a more important role in the development of the program than the precondition Q. Programming is employed to solve a problem, and the problem needs to be clearly stated with precise preconditions and postconditions.

The example of a program[3] P to determine the maximum of two integers x and y is discussed in [5]. A program P is required that satisfies:

$$\{\mathrm{T}\}P\{R : z = max(x, y)\}$$

The postcondition R is then refined by replacing max with its definition:

$$\{R : (z \geq x \wedge z \geq y) \wedge (z = x \vee z = y)\}$$

The next step is to identify a command that could be executed in order to establish the postcondition R. One possibility is $z := x$ and the conditions under which this assignment establishes R is given by:

$$\begin{aligned} \mathrm{wp}(z := x, \mathrm{R}) &= x \geq x \wedge x \geq y \wedge (x = x \vee x = y) \\ &= x \geq y \end{aligned}$$

Another possibility is $z := y$ and the conditions under which this assignment establishes R is given by:

$$\mathrm{wp}(z := y\,\mathrm{R}) = y \geq x$$

The desired program is then given by:

if $x \geq y \rightarrow z := x$
$\square\, y \geq x \rightarrow z := y$
fi

There are many more examples of formal program development in [5].

12.3 Axiomatic Definition of Programming Languages

An assertion is a property of the program's objects: e.g. the assertion $(x - y > 5)$ is an assertion that may or may not be satisfied by a state of the program during execution. For example, the state in which the values of the variables x and y are 7 and 1, respectively, satisfies the assertion; whereas a state in which x and y have values 4 and 2, respectively, does not.

Robert Floyd (Fig. 12.4) did pioneering work on software engineering from the 1960s, including important contributions to the theory of parsing; the semantics of programming languages and methodologies for the creation of efficient and reliable software.

[3]Many of these examples are considered "toy programs" when compared to real-world industrial software development, but they illustrate the concepts involved in developing software rigorously using the weakest precondition calculus.

Fig. 12.4 Robert Floyd

Floyd believed that there was a way to construct a rigorous proof of the correctness of the programs using mathematics. He showed that mathematics could be used for program verification, and he introduced the concept of *assertions* that provided a way to verify the correctness of programs. His first article on program proving techniques based on assertions was in 1967 [6].

Floyd's 1967 paper was concerned with assigning meaning to programs, and he also introduced the idea of a loop invariant. His approach was based on programs expressed by flowcharts, and an assertion was attached to the edge of the flowchart. The meaning was that the assertion would be true during execution of the corresponding program whenever execution reached that edge. For a loop, Floyd placed an assertion P on a fixed position of the cycle, and proved that if execution commenced at the fixed position with P true, and reached the fixed position again, then P would still be true.

Flowcharts were employed in the 1960s to explain the sequence of basic steps for computer programs. Floyd's insight was to build upon flowcharts and to apply *an invariant assertion to each branch* in the flowchart. These assertions state the essential relations that exist between the variables at that point in the flowchart. An example relation is "$R = Z > 0,\ X = 1,\ Y = 0$". He devised a general flowchart language to apply his method to programming languages. The language essentially contains boxes linked by flow of control arrows.

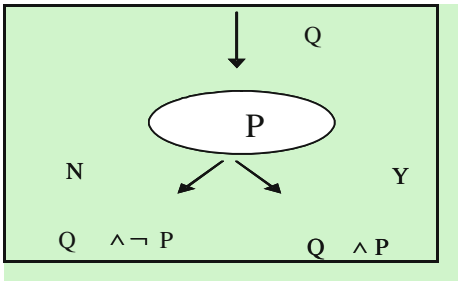

Fig. 12.5 Branch assertions in flowcharts

Consider the assertion Q that is true on entry to a branch where the condition at the branch is P. Then, the assertion on exit from the branch is $Q \wedge \neg P$ if P is false and $Q \wedge P$ otherwise (Fig. 12.5).

The use of assertions may be employed in an assignment statement. Suppose x represents a variable and v represents a vector consisting of all the variables in the program. Suppose $f(x, v)$ represents a function or expression of x and the other program variables represented by the vector v. Suppose the assertion $S(f(x, v), v)$ is true before the assignment $x = f(x, v)$. Then the assertion $S(x, v)$ is true after the assignment (Fig. 12.6). This is given by:

Floyd used flowchart symbols to represent entry and exit to the flowchart. This included entry and exit assertions to describe the program's entry and exit conditions.

Floyd's technique showed how a computer program is a sequence of logical assertions. Each assertion is true whenever control passes to it, and statements appear between the assertions. The initial assertion states the conditions that must be true for execution of the program to take place, and the exit assertion essentially describes what must be true when the program terminates.

He recognized that if it can be shown that the assertion immediately following each step is a consequence of the assertion immediately preceding it, then the assertion at the end of the program will be true, provided the appropriate assertion was true at the beginning of the program.

His influential 1967 paper, "*Assigning Meanings to Programs*" influenced Hoare's work on preconditions and postconditions leading to Hoare logic [7]. Hoare recognized that Floyd's approach provided an effective method for proving the correctness of programs, and he built upon Floyd's work to cover the familiar

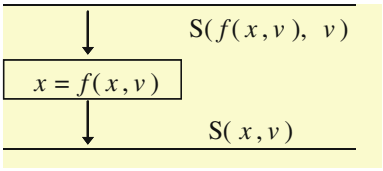

Fig. 12.6 Assignment assertions in flowcharts

constructs of high-level programming languages. Floyd's paper also presented a formal grammar for flowcharts, together with rigorous methods for verifying the effects of basic actions like assignments.

Hoare logic is a formal system of logic for programming semantics and program verification, and it was originally published in Hoare's 1969 paper "*An axiomatic basis for computer programming*" [7]. Hoare and others have subsequently refined it, and it provides a logical methodology for precise reasoning about the correctness of computer programs. The well-formed formulae of the logical system are of the form:

$$P\{a\}Q$$

where P is the precondition; a is the program fragment and Q is the postcondition. The precondition P is a predicate (or input assertion), and the postcondition R is a predicate (output assertion). The braces separate the assertions from the program fragment. The well-formed formula $P\{a\}Q$ is itself a predicate that is either true or false. This notation expresses the partial correctness of a with respect to P and Q, where *partial correctness* and *total correctness* are defined as follows:

Definition (*Partial Correctness*)
A program fragment a is partially correct for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied and the execution terminates, then the resulting state satisfies Q.

The proof of partial correctness requires proof that the postcondition Q is satisfied if the program terminates. Partial correctness is a useless property unless termination is proved, as any non-terminating program is partially correct with respect to any specification.

Definition (*Total Correctness*)
A program fragment a is totally correct for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied then execution terminates and the resulting state satisfies Q.

The proof of total correctness requires proof that the postcondition Q is satisfied and that the program terminates. Total correctness is expressed by $\{P\}\ a\ \{Q\}$. The calculus of weakest preconditions developed by Dijkstra (discussed in the previous section) is based on total correctness, whereas Hoare's approach is based on partial correctness.

Hoare's axiomatic theory of programming languages consists of axioms and rules of inference to derive certain pre-post formulae. The meaning of several constructs in programming languages is presented here in terms of pre-post semantics.

- **Skip**

The meaning of the skip command is:

$$P\{skip\}P$$

Skip does nothing and it's this instruction guarantees that whatever condition is true on entry to the command is true on exit from the command.

- **Assignment**

The meaning of the assignment statement is given by the axiom:

$$P_e^x\{x := e\}P$$

The notation P_e^x has been discussed previously and denotes the expression obtained by substituting e for all free occurrences of x in P.

The meaning of the assignment statement is that P will be true after execution if and only if the predicate P_e^x with the value of x replaced by e in P is true before execution (since x will contain the value of e after execution).

- **Compound**

The meaning of the conditional command is:

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

The execution of the **compound** statement involves the execution of S_1 followed by S_2. The correctness of the compound statement with respect to P and R is established by proving that the correctness of S_1 with respect to P and Q, and the correctness of S_2 with respect to Q and R.

- **Conditional**

The meaning of the conditional command is:

$$\frac{P \wedge B\{S_1\}Q, \quad P \wedge \neg B\{S_2\}Q}{P\{\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\}Q}$$

The execution of the **if** statement involves the execution of S_1 or S_2. The execution of S_1 takes place only when B is true, and the execution of S_2 takes place only when $\neg B$ is true. The correctness of the **if** statement with respect to P and Q is established by proving that S_1 and S_2 are correct with respect to P and Q.

However, S_1 is executed only when B is true, and therefore it is required to prove the correctness of S_1 with respect to $P \wedge B$ and Q, and the correctness of S_2 with respect to $P \wedge \neg B$ and Q.

- **While Loop**

The meaning of the while loop is given by:

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

The property P is termed the loop invariant as it remains true throughout the execution of the loop. The invariant is satisfied before the loop begins and each iterations of the loop preserves the invariant.

The execution of the **while loop** is such that if the truth of P is maintained by one execution of S, then it is maintained by any number of executions of S. The execution of S takes place only when B is true, and upon termination of the loop $P \wedge \neg B$ is true.

Loops may fail to terminate and therefore there is a need to prove termination. The loop invariant needs to be determined for formal program development.

12.4 Tabular Expressions

Tables of constants have used for millennia to define mathematical functions. The tables allow the data to be presented in an organized form that is easy to reference and use. The data presented in tables is well-organized and provides an explicit definition of a mathematical function. This allows the computation of the function for a particular value to be easily done. The use of tables is prevalent in schools where primary school children are taught multiplication tables and high school students refer to sine or cosine tables. The invention of electronic calculators may lead to a reduction in the use of tables as students may compute the values of functions immediately.

Tabular expressions are a generalization of tables in which constants may be replaced by more general mathematical expressions. Conventional mathematical expressions are a special case of tabular expressions. In fact, everything that can be expressed as a tabular expression can be represented by a conventional expression. Tabular expressions can represent sets, relations, functions and predicates and conventional expressions. A tabular expression may be represented by a conventional expression, but its advantage is that the tabular expression is easier to read and use, since a complex conventional expression is replaced by a set of simpler expressions.

Tabular expressions are invaluable in defining a piecewise continuous function, as it is relatively easy to demonstrate that all cases have been considered in the definition. It is easy to miss a case or to give an inconsistent definition in the conventional definition of a piecewise continuous function. The evaluation of a tabular expression is easy once the type of tabular expression is known. Tabular expressions have been applied to practical problems including the precise documentation of the system requirements of the A7 aircraft [8].

Tabular expressions have been applied to precisely document the system requirements and to solve practical industrial problems. A collection of tabular expressions are employed to document the system requirements. The meaning of these tabular expressions in terms of their component expressions was done by Parnas [9]. He identified several types of tabular expressions and provided a formal meaning for each type. A more general model of tabular expressions was proposed by Janicki [10], although this approach was based on diagrams using an artificial cell connection graph to explain the meaning of the tabular expressions. Parnas and others have proposed a general mathematical foundation for tabular expressions.

The function $f(x, y)$ is defined in the tabular expression below. The tabular expressions consist of headers and a main grid. The headers define the domain of the function and the main grid gives the definition. It is easy to see that the function is defined for all values on its domain as the headers are complete. It is also easy to see that the definition is consistent as the headers partition the domain of the function.

The evaluation of the function for a particular value (x, y) involves determining the appropriate row and column from the headers of the table and computing the grid element for that row and column (Fig. 12.7).

For example, the evaluation of $f(2, 3)$ involves the selection of row 1 of the grid (as $x = 2 \geq 0$ in H_1) and the selection of column 3 (as $y = 3 < 5$ in H_2). Hence, the value of $f(2, 3)$ is given by the expression in row 1 and column 3 of the grid, i.e. $-y^2$ evaluated with $y = 3$ resulting in -9. The table simplifies the definition of the function. Tabular expressions have several applications (Table 12.5).

Examples of Tabular Expressions

The objective of this section is to illustrate the power of tabular expressions by considering a number of examples. The more general definition of tabular expressions allows for multidimensional tables, including multiple headers, and supports rectangular and non-rectangular tables. However, the examples presented here will be limited to two-dimensional rectangular tables, and will usually include two headers and one grid, with the meaning of the tables given informally.

	$y = 5$	$y > 5$	$y < 5$	H_2
$x \geq 0$	0	y^2	$-y^2$	G
$x < 0$	x	$x+y$	$x-y$	

H_1

Fig. 12.7 Tabular expressions (normal table)

Table 12.5 Applications of tabular expressions

Applications of tabular expressions
Specify requirements
Specify module interface design
Description of implementation of module
Mathematical software inspections

The role of the headers and grid will become clearer in the examples, and usually, the headers contain predicate expressions, whereas the grid usually contains terms. However, the role of the grid and the headers change depending on the type of table being considered.

Normal Function Table

The first table that we discuss is termed the normal function table, and this table consists of two headers (H_1 and H_2) and one grid G. The headers are predicate expressions that partition the domain of the function; header H_1 partitions the domain of y, whereas header H_2 partitions the domain of x. The grid consists of terms. The function $f(x, y)$ is defined by the following table (Fig. 12.8):

The evaluation of the function $f(x, y)$ for a particular value of x, y is given by:

1. Determine the row i in header H_1 that is true.
2. Determine the column j in header H_2 that is true.
3. The evaluation of $f(x, y)$ is given by $G(i, j)$.

For example, the evaluation of $f(-2, 5)$ involves row 3 of H_1 as y is 5 (>0) and column 1 of header H_2 as x is −2 (<0). Hence, the element in row 3 and column 1 of the grid is selected (i.e. the element $x + y$). The evaluation of $f(-2, 5)$ is $-2 + 5 = 3$.

The usual definition of the function $f(x, y)$ defined piecewise is:

$$
\begin{aligned}
f(x,y) &= x^2 - y^2 \quad \text{where } x \le 0 \wedge y < 0;\\
f(x,y) &= x^2 + y^2 \quad \text{where } x > 0 \wedge y < 0;\\
f(x,y) &= x + y \quad \text{where } x \ge 0 \wedge y = 0;\\
f(x,y) &= x - y \quad \text{where } x < 0 \wedge y = 0;\\
f(x,y) &= x + y \quad \text{where } x \le 0 \wedge y > 0;\\
f(x,y) &= x^2 + y^2 \quad \text{where } x > 0 \wedge y > 0;
\end{aligned}
$$

		$x<0$	$x=0$	$x>0$	H_2
H_1	$y<0$	x^2-y^2	x^2-y^2	x^2+y^2	
	$y=0$	x-y	x+y	x+y	G
	$y>0$	x+y	x+y	x^2+y^2	

Fig. 12.8 Normal table

The danger with the usual definition of the piecewise function is that it is more difficult to be sure that every case has been considered, as it is easy to miss a case or for the cases to be overlap. Care needs to be taken with the value of the function on the boundary, as it is easy to introduce inconsistencies. It is straightforward to check that the tabular expression has covered all cases, and that there are no overlapping cases. This is done by examination of the headers to check for consistency and completeness. The headers for the tabular representation of *f(x, y)* must partition the values that *x* and *y* may take, and this is clear from an examination of the headers.

Normal relation tables and predicate expression tables are interpreted similarly to normal function tables except that the grid entries are predicate expressions rather than terms as in the normal function table. The result of the evaluation of a predicate expression table is a Boolean value of true or false, whereas the result of the evaluation of the normal relation table is a relation. A characteristic predicate table is similar except that it is interpreted as a relation whose domain and range consist of tuples of fixed length. Each element of the tuple is a variable and the tuples are of the form (('x_1,'x_2,....'x_n), (x_1',x_2',....x_n')).

Inverted Function Table

The inverted function table is different from the normal table in that the grid contains predicates, and the header H_2 contains terms. The function *f(x, y)* is defined by the following inverted table (Fig. 12.9):

The evaluation of the function *f(x, y)* for a particular value of *x, y* is given by:

1. Determine the row *i* in header H_1 that is true.
2. Select row *i* of the grid and determine the column *j* of row *i* that is true.
3. The evaluation of *f(x, y)* is given by $H_2(j)$.

For example, the evaluation of *f*(−2,5) involves the selection of row 3 of H_1 as *y* is 5 (>0). This means that row 3 of the grid is then examined and as *x* is −2 (<0) column 2 of the grid is selected. Hence, the element in column 2 of H_2 is selected as the evaluation of *f(x, y)* (i.e. the element *x* − *y*). The evaluation of *f*(−2, 5) is therefore −2 − 5 = −7.

		$x+y$	$x-y$	xy	H_2
H_1	$y<0$	$x<0$	$x=0$	$x>0$	
	$y=0$	$x>0$	$x<0$	$x=0$	G
	$y>0$	$x=0$	$x<0$	$x>0$	

Fig. 12.9 Inverted table

The usual definition of the function *f(x, y)* defined piecewise is:

$$
\begin{array}{ll}
f(x,y) = x+y & \text{where } x<0 \wedge y<0; \\
f(x,y) = x-y & \text{where } x=0 \wedge y<0; \\
f(x,y) = xy & \text{where } x>0 \wedge y<0; \\
f(x,y) = x+y & \text{where } x>0 \wedge y=0; \\
f(x,y) = x-y & \text{where } x<0 \wedge y=0; \\
f(x,y) = xy & \text{where } x=0 \wedge y=0; \\
f(x,y) = x+y & \text{where } x=0 \wedge y>0; \\
f(x,y) = x-y & \text{where } x<0 \wedge y>0; \\
f(x,y) = xy & \text{where } x>0 \wedge y>0;
\end{array}
$$

Clearly, the tabular expression provides a more concise representation of the function. The inverted table arises naturally when there are many cases to consider, but only a few distinct values of the function. The function *f(x, y)* can also be represented in an equivalent normal function table. In fact, any function that can be represented by an inverted function table may be represented in a normal function table and vice versa.

Inverted predicate expression tables and inverted relation tables are interpreted similarly to inverted function tables except that the header H_1 consists of predicate expressions rather than terms. The result of the evaluation of an inverted predicate expression table is the Boolean value true or false, whereas the evaluation of an inverted relation table is a relation.

There is more detailed information on Parnas's contributions to software engineering, including software requirements, software design and software inspections in [8].

12.5 Review Questions

1. What are Dijkstra's main achievements in computer science?
2. Describe Dijkstra's weakest precondition calculus and its application to formal program development.
3. What are Hoare's main achievements in computer science?
4. Describe Hoare's axiomatic semantics and its application to the correctness of computer programs.
5. What are Parnas's main achievements in computer science?

6. Describe tabular expressions and their applications.
7. What is a normal function table? What is an inverted function table?
8. Investigate Floyd's contributions to the computing field.

12.6 Summary

Dijkstra, Hoare and Parnas have made important contributions to computer science, and they have received numerous awards in recognition of their achievements. Their work has provided a scientific basis for computer software development, and we presented a selection of their contributions in this chapter.

Dijkstra has made contributions to language development, operating systems, formal program development and to the vocabulary of Computer Science. His calculus of weakest preconditions is used for the formal development of computer programs, where a program and its proof of correctness are developed together.

Hoare has developed the quicksort algorithm, the axiomatic approach to program semantics, and programming constructs for concurrency. He was responsible for producing the first commercial compiler for Algol 60 at Elliot Brothers.

Parnas has made a strong contribution to software engineering, including contributions to requirements specification, software design, software inspections, testing, tabular expressions, predicate logic and ethics for software engineers. His reflections on software engineering remain valuable and contain the insight gained over a long career. His tabular expressions are useful in defining piecewise continuous functions, where tabular expressions are a generalization of tables in which constants can be replaced by more general mathematical expressions.

Reference

1. G. O'Regan, *Mathematical Approaches to Software Quality*, vol 26 (Springer, London)
2. G. O'Regan, *A Practical Approach to Software Quality* (Springer Verlag, New York, 2002)
3. C. Jones, *Systematic Software Development using VDM* (Prentice Hall International, 1986)
4. E.W. Dijkstra, *A Disciple of Programming* (Prentice Hall, Englewood Cliffs, NJ, 1976)
5. D. Gries, *The Science of Programming* (Springer, Berlin, 1981)
6. R. Floyd, Assigning Meanings to Programs, in Proc. Symp. Appl. Math. (19), 19–32 (1967)
7. C.A.R. Hoare, An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–585 (1969)
8. D. Hoffman, D.L. Parnas, in *Software Fundamentals*, ed. by D. Weiss. Collected Papers by D.L. Parnas (Addison Wesley, Reading, 21)
9. D.L. Parnas, *Tabular Representation of Relations*. CRL Report 260.McMaster University, Canada (1992)
10. R. Janicki, On a Formal Semantics of Tabular Expressions. Technical Report CRL 355. Communications Research Laboratory, McMaster University, Ontario (1997)