Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

K.R. Apt, E.-R. Olderog

Introduction to program verification

Computer Science/Department of Software Technology Report CS-R9036 August

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Introduction to Program Verification

Krzysztof R. Apt
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712–1188, U.S.A.

Ernst-Rüdiger Olderog
Department of Computer Science
University of Oldenburg
2900 Oldenburg, West Germany

We provide a systematic introduction to program verification based on the assertional method. We study here deterministic, nondeterministic, parallel and distributed programs and deal with such properties as partial correctness, termination, absence of failures, interference freedom and deadlock freedom.
Note. This paper will appear in: E.J. Neuhold, M. Paul, Eds., Formal Description of Programming Concepts, IFIP State-of-the-Art Report, Springer-Verlag.

Keywords and Phrases: program verification, assertional method, sequential, parallel and distributed programs, partial correctness, termination, interference freedom, deadlock freedom.
1985 Mathematics Subject Classification: 68N05, 68Q55, 68Q60.

1 Introduction

During the last three decades we witnessed the development of new programming languages and new styles of programming. Our understanding of the whole programming process increased significantly. It also became increasingly clear that the only way to ensure program correctness is by providing a rigorous proof that the program meets its specification.

Program verification is a systematic approach to proving the absence of program errors. The idea is to compare the program with a specification expressing the desired program properties. A number of approaches to program verification have been proposed and used in the literature.

The most common of them is based on an *operational reasoning*, i.e. on an analysis in terms of the execution sequences of the given program.

Report CS-R9036
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

To this end, an informal understanding of the program semantics is used. While this analysis is often successful in the case of sequential programs, it is much less so in the case of concurrent programs. The number of possible execution sequences is then most often forbiddingly large and it is all too easy to overlook a possible execution sequence.

A different approach is based on an *axiomatic reasoning*. According to this approach, we first need a formalism which makes it possible to express the relevant program properties. In other words, using the terminology of logic, we first need an appropriate *language* defined by syntactic rules that allows us to construct *well-formed formulas*. Next, we need a *proof system* consisting of axioms and rules which allows us to construct formal *proofs* of certain relevant formulas. And this proof system should be such that only true properties can be proved in it.

The origins of this approach to program correctness can be traced back to Turing [1949], but the first constructive effort should be attributed to Floyd [1967] where proving correctness of flowchart programs by means of assertions was proposed. This method was subsequently presented in a syntax directed manner in the seminal paper of Hoare [1969] which opened the way to a proof-theoretic approach for other classes of programs. His approach has received a great deal of attention and several Hoare-style proof systems dealing with various programming constructs have been proposed since then.

In 1976 and 1977, this approach was extended to parallel programs by Owicki and Gries [1976] and Lamport [1977], and in 1980 and 1981 to distributed programs by Apt, Francez and de Roever [1980] and Levin and Gries [1981].

The aim of this article is to provide a systematic exposition of this method. We study sequential, parallel and distributed programs and deal with several program properties.

Sequential programs are programs in which at each time instance at most one instruction can be executed. A special case of sequential programs are *deterministic programs*, those in which there is always at most one next instruction to be executed. Their correctness is studied in Chapter 3. A more general class of sequential programs consists of *nondeterministic programs*, those in which the choice of the next instruction to be executed is not fully determined. Their correctness is studied in Chapter 4. We also discuss there a systematic development of programs together with their correctness proofs.

Concurrent programs are programs in which more than one component can be active at a time. We study here two types of concurrent programs - parallel programs and distributed programs. *Parallel programs* are programs in which the components can communicate by means of shared variables. Their correctness is studied in Chapters 5,6 and 7, in which successively more sophisticated classes of programs are considered.

Distributed programs are programs in which the components, sometimes

called *processes*, do not share variables and can communicate instead by messages. Correctness of distributed programs is studied in Chapter 8.

Depending on the type of program, correctness refers to different program properties and hence requires different methods of reasoning. For sequential programs we study

- partial correctness,
- termination, and
- absence of failures.

For concurrent programs we additionally study

- interference freedom, and
- deadlock freedom

When dealing with concurrent programs, we heavily rely on the concepts and techniques developed for sequential programs. Our presentation does not aim at completeness and should serve merely as an introduction to the basic concepts and techniques of program verification by means of the assertional method.

2 Preliminaries

In this chapter we explain the notation and syntax we shall use throughout this article and explain some elementary notions from mathematical logic we shall need in the sequel.

First we define an assertion language in which assertions about programs will be written. This language extends first order logic in that it uses types and array variables. The assertion language consists of the types, the alphabet, the expressions and the formulas.

2.1 Types and Alphabets

First, we define types. We assume at least two *basic* types:

- **integer**,
- **Boolean**

and for each $n \geq 1$ one *higher type*:

- $T_1 \times \ldots \times T_n \rightarrow T_{n+1}$,
  where each T_i is a basic type. Here $T_1, \ldots, T_n$ are called *argument* types and T_{n+1} the *value* type. Some other basic types (like **character**) will be occasionally used.

A type provides us with information about the intended set of values. The type **integer** consists of all integers, the type **Boolean** consists of two values — **true** and **false**, and a type $T_1 \times \ldots \times T_n \rightarrow T_{n+1}$ consists of all functions from the Cartesian product of [the sets described by] $T_1, \ldots, T_n$ to [the set described by] T_{n+1}.

Next, we introduce the alphabet of an assertion language. It consists of the following classes of symbols:

- *variables*,
- *constants*,
- *quantifiers*, which are: $\exists$ (there exists) and $\forall$ (for all),
- *parentheses*, which are: (,), [and],
- *punctuation symbols*, which are , and ..

2.2 Variables and Constants

We assume three sorts of variables:

- simple,
- subscripted,
- array.

Each variable has a type associated with it and can assume as values only elements of this type. Simple and subscripted variables are of a basic type and array variables are of a higher type.

Simple variables of a type **integer** are called *integer variable* and usually denoted by i, j, k, x, y, z. Simple variables of a type **Boolean** are called *Boolean variables*. In programs simple variables will be usually denoted by more suggestive names like *turn* or *found*.

Subcripted variables will be explained in the next section. Array variables (or just *arrays*) are usually denoted by a, b, c. They range over functions of an appropriate type. In particular, when all argument types are of a type **integer**, there are no bounds associated with it. On the other hand, we occasionally use finite *sections* of an arrays. When an array a has one argument type which is **integer**, then for any k, l with $k \leq l$ the section $a[k : l]$ stands for the restriction of a. to the interval $\{i \mid k \leq i \leq l\}$. The number of arguments of the higher type associated with the array a is called its *dimension*.

Each constant has a type associated with it. Its value belongs to this type and is fixed. We assume two sorts of constants:

- of basic type

- of higher type

Among constants of basic type we distinguish *integer constants* and *Boolean constants*. We assume infinitely many integer constants: 0, -1, 1, -2, 2, ... and two Boolean constants: **true**, **false**.

Among constants of a higher type $T_1 \times \ldots \times T_n \rightarrow T_{n+1}$ we distinguish two sorts. When the value type T_{n+1} is **Boolean**, the constant is called a *relation symbol* and otherwise the constant is called a *function symbol*; n is called the *arity* of the constant.

We do not wish to be overly specific but we assume existence of at least the following function and relation symbols:

- $+, -, \cdot, mod$ of type **integer** $\times$ **integer** $\rightarrow$ **integer**,
- $<$ of type **integer** $\times$ **integer** $\rightarrow$ **Boolean**,
- $=_{int}$ of type **integer** $\times$ **integer** $\rightarrow$ **Boolean**,
- $=_{Bool}$ of type **Boolean** $\times$ **Boolean** $\rightarrow$ **Boolean**,
- $\neg$ of type **Boolean** $\rightarrow$ **Boolean**,
- $\vee, \wedge, \rightarrow, \leftrightarrow$ of type **Boolean** $\times$ **Boolean** $\rightarrow$ **Boolean**

The value of each of these constants is well known and therefore not further explained. In the sequel we shall drop the subscript when using $=$ as from the context it will be always clear which interpretation is meant.

The 2-ary (or *binary* constants) are written in an *infix form*, i.e. between the arguments. The relation symbols $\neg, \wedge, \vee, \rightarrow$ and $\leftrightarrow$ are usually called *connectives*.

The value of variables can be changed during the execution of a program, whereas the value of constants remains fixed forever.

2.3 Expressions and Assertions

Out of variables and constants we build up *expressions*. Each expression has a type associated with it. We consider here only expressions of a basic type. Thus we distinguish *integer expressions* and *Boolean expressions*. Expressions are defined by induction as follows:

- a simple variable of type T is an expression of type T,
- a constant of a basic type T is an expression of type T,
- if $s_1, \ldots, s_n$ are expressions of type $T_1, \ldots, T_n$, respectively and c a constant of type $T_1 \times \ldots \times T_n \rightarrow T$, then $c(s_1, \ldots, s_n)$ is an expression of type T,

- if $s_1, \ldots, s_n$ are expressions of type $T_1, \ldots, T_n$, respectively, and a an array variable of type $T_1 \times \ldots \times T_n \rightarrow T$, then $a[s_1, \ldots, s_n]$ is an expression of type T,
- if B is an expression of type **Boolean** and s_1 and s_2 expressions of type T, then **if** B **then** s_1 **else** s_2 **fi** is an expression of type T.

Expressions of the form $a[s_1, \ldots, s_n]$ are called *subscripted expressions.* Expressions are usually denoted by letters s, t and Boolean expressions by the letter B. Simple and subscripted variables are usually denoted by letters u, v. They can be assigned a value in the programs by means of an assignment statement, which will be discussed in the next chapter. Assignments to a subscripted variable $a[s_1, \ldots, s_n]$ model a selected update of the array a at the argument tuple $[s_1, \ldots, s_n]$.

Assertions are formulas in the assertion language. They are defined by induction as follows:

- a Boolean expression is an assertion,
- if p, q are assertions, then $\neg p, (p \vee q), (p \wedge q), (p \rightarrow q)$ and $(p \leftrightarrow q)$ are assertions,
- if x is a simple variable and p an assertion, then $\exists x p$ and $\forall x p$ are assertions.

By a *subassertion* we mean a substring of p which is again an assertion.

In this definition brackets are introduced around binary connectives to avoid ambiguities. Note that in the last clauses only quantifiers with simple variables are allowed. Also, note that assertions built up without the use of quantifiers are just Boolean expressions.

2.4 Notational Conventions

The definition of expressions and assertions is rigorous at the expense of excessive use of parentheses. We now introduce a number of conventions which allow us to eliminate some of the brackets. This will enhance the readability of expressions and assertions.

First we introduce a *binding order.* We assume that all relation symbols, in particular $<$ and $=$, bind stronger than $\neg, \exists$ and $\forall$, which bind stronger than $\vee$ and $\wedge$ which in turn bind stronger than $\rightarrow$ and $\leftrightarrow$. Moreover, we assume that both $\vee$ and $\wedge$ associate to the right. Also we abbreviate

$$\begin{array}{rcl} p_1 \vee \ldots \vee p_n & \text{to} & \vee_{i=1}^{n}\, p_i, \\ p_1 \wedge \ldots \wedge p_n & \text{to} & \wedge_{i=1}^{n}\, p_i, \\ (s < t \vee s = t) & \text{to} & s \leq t, \\ (s < t \wedge t \leq w) & \text{to} & s < t \leq w, \end{array}$$

and similarly with other combinations of $<$ and $=$. Occasionally we write $s > t$ instead of $t < s$, and similarly with $s \geq t$. We assume that $\leq$ and $\geq$ bind stronger than all connectives and quantifiers.

Next we abbreviate

$$\begin{array}{rcl} \exists x(x \geq t \wedge p) & \text{to} & \exists x \geq t.p, \\ \exists x(s \leq x < t \wedge p) & \text{to} & \exists x(s \leq x < t).p, \end{array}$$

and similarly with other combinations of $<, \leq, \geq$, and $>$, and abbreviate

$$\begin{array}{rcl} \forall x(x \geq t \rightarrow p) & \text{to} & \forall x \geq t.p, \\ \forall x(s \leq x < t \rightarrow p) & \text{to} & \forall x(s \leq x < t).p, \end{array}$$

and similarly with other combinations of $<, \leq, \geq$, and $>$.

Finally, once an assertion of the form $(p \vee q), (p \wedge q), (p \rightarrow q)$ or $(p \leftrightarrow q)$ has been constructed, we omit the outer brackets.

The following example illustrates the use of these conventions. Consider the assertion

$$(\forall x((x = t \vee x < t) \rightarrow \exists y((y < s) \wedge (p \wedge q))) \rightarrow r).$$

Thanks to the binding order applied to $<$ it can be simplified to

$$(\forall x((x = t \vee x < t) \rightarrow \exists y(y < s \wedge (p \wedge q))) \rightarrow r),$$

which thanks to the convention of associating $\wedge$ to the right further simplifies to

$$(\forall x((x = t \vee x < t) \rightarrow \exists y(y < s \wedge p \wedge q)) \rightarrow r).$$

Introducing $\leq$ we obtain

$$(\forall x(x \leq t \rightarrow \exists y(y < s \wedge p \wedge q)) \rightarrow r)$$

Finally, applying the last three abbreviation conventions we obtain the assertion

$$\forall x \leq t.\exists y < s.(p \wedge q) \rightarrow r$$

which is much more readable than its original form.

Note that in the last step we also reintroduced brackets around $p \wedge q$ to avoid ambiguities. This step can be formally defined when discussing the abbreviation $\exists x \leq t.p$.

2.5 Substitution

An important concept is that of a *substitution* of an expression t for the free occurrences of the simple or subscripted variable u in an assertion p, written as $p[t/u]$. Its definition presupposes that t and u are of the same type. To explain it we first define the substitution of an expression t for a simple or subscripted variable u in an expression s, written as $s[t/u]$. It is defined by induction on the structure of the expression s.

We put for a simple variable x

$$x[t/u] \equiv \begin{cases} t & \text{if } x \equiv u \\ x & \text{otherwise,} \end{cases}$$

and, following De Bakker [1980], for a subscripted variable $a[s_1, \ldots, s_n]$

$$a[s_1, \ldots, s_n][t/b[t_1, \ldots, t_m]] \equiv a[s_1, \ldots, s_n] \text{ if } a \not\equiv b$$

and otherwise

$$a[s_1, \ldots, s_n][t/a[t_1, \ldots, t_n]] \equiv \textbf{if } \wedge_{i=1}^{n} s_i' = t_i \textbf{ then } t \textbf{ else } a[s_1', \ldots, s_n'] \textbf{ fi}$$

where $s_i' \equiv s_i[t/a[t_1, \ldots, t_n]]$.

Other cases are straightforward and omitted. The last clause motivated the introduction of conditional expressions in our syntax. Intuitively, it represents a statement "the array a assigns t to the argument tuple $[t_1, \ldots, t_n]$."

Now we define the substitution $p[t/u]$ for an assertion p. The definition is again by induction. The base case of a Boolean expression $c(s_1, \ldots, s_n)$ is handled using the definition of substitution for exressions:

$$c(s_1, \ldots, s_n)[t/u] \equiv c(s_1[t/u], \ldots, s_n[t/u]).$$

The inductive clauses are straightforward with the exception of the case of assertions of the form $\exists x p$ and $\forall x p$.

We put

$$(\exists x p)[t/u] \equiv \exists y p[y/x][t/u],$$

where y does not appear in p, t or u and is of the same type as x, and similarly for $\forall x p$.

Substitution will be used in the next chapter when dealing with the assignment statement.

By a *bound occurrence* of a simple variable in an assertion p we mean an occurrence within a subassertion of p of the form $\exists x r$ or $\forall x r$. An occurrence of a simple variable in an assertion p is called *free* if it is not a bound one.

Given an assertion p by $free(p)$ we denote the set of simple variables which have a free occurrence (or *occur free*) in p augmented with the set of array variables which occur in p.

2.6 Formal Proof Systems

Our main interest here is in proving program correctness. To this purpose we shall investigate *correctness formulas*, i.e. statements of the form $\{p\}\ S\ \{q\}$ where p, q are assertions and S is a program under consideration. All program properties we wish to verify will be expressed in the form of correctness formulas and, occasionally, assertions. Therefore, we shall present various proof systems allowing us to prove correctness formulas.

A proof system consists of a language in which formulas are defined and of *axioms* and *proof rules*. Axioms are formulas assumed to be given. Proof rules allow us to prove from the already established formulas some new formulas. They have the form

$$\frac{\varphi_1, \ldots, \varphi_k}{\varphi_{k+1}} \quad \text{where } \ldots,$$

where $\varphi_1, \ldots, \varphi_{k+1}$ are formulas and "..." stands for a condition describing when the rule can be applied. It should be read as "deduce φ_{k+1} from $\varphi_1, \ldots, \varphi_k$ provided "..." ". The formulas $\varphi_1, \ldots, \varphi_k$ are called the *premises* of the rule and the formula φ_{k+1} is called the *conclusion* of the rule.

A *proof* in a given proof system is a sequence of formulas in which each formula is either an axiom or follows from the previous ones by a rule of the system. The last formula in a proof is called a *theorem*. Or, to put it the other way around: a theorem is a formula which has a proof or can be *proved* in the given proof system. The length of this sequence is called the *length of the proof*.

Given a proof system P and a formula ϕ we shall write $\vdash_P \phi$ to denote that ϕ is a theorem of P. On the other hand, for proof systems allowing us to prove correctness formulas we shall write $\vdash_P \phi$ to denote that the correctness formula ϕ is a theorem of P augmented by the set of all true assertions. This means that in the correctness proofs we shall use all true assertions as axioms.

We refer here to an informal definition of truth. The assertions claimed true will always be very simple and their truth will rely on some elementary facts about integers. Informally, an assertion is true if it holds for all possible values of the variables which occur free in it. A formal definition of truth can be given but is omitted here.

3 Deterministic Programs

In a deterministic program there is at most one next instruction to be executed so that from a given initial state only one execution sequence is generated. In classical programming languages like Pascal only deterministic programs could be written. Here we consider a very small set of deterministic programs, usually called **while**-programs.

3.1 Syntax

A while-program is a string of symbols including the keywords **if, then, else, fi, while, do** and **od** which is generated by the following grammar:

$$S ::= \mathit{skip} \mid u := t \mid S_1;\ S_2 \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } B \textbf{ do } S_1 \textbf{ od}.$$

Following the conventions of the previous chapter, the letter u stands for a simple or subscripted variable and t for an expression. We require that in an assignment $u := t$ the variable u and the expression t are of the same type. Since types are implied by the notational conventions of the previous chapter, we do not declare variables in the programs. As an abbreviation we introduce

$$\textbf{if } B \textbf{ then } S \textbf{ fi} \equiv \textbf{if } B \textbf{ then } S \textbf{ else } \mathit{skip} \textbf{ fi}.$$

As usual, spaces and indentation will be used to make programs more readable, but these additions are not part of the formal syntax. Here and elsewhere programs will be denoted by letters R, S, T.

Though we assume that the reader is familiar with **while**-programs, we would like to recall how they are executed. The program *skip* changes nothing and just terminates. An *assignment* $u := t$ assigns the value of the expression t to the (possibly subscripted) variable u and then terminates. A *sequential composition* $S_1;\ S_2$ is executed by first executing the statement S_1 followed upon its termination by an execution of S_2. Since this interpretation of sequential composition is associative, we need not introduce brackets enclosing $S_1;\ S_2$. Execution of a *conditional statement* **if** B **then** S_1 **else** S_2 **fi** starts by evaluating the Boolean expression B. If B is true, the statement S_1 is executed, otherwise (i.e., if B is false), S_2 is executed. Execution of a *loop* **while** B **do** S **od** starts with the evaluation of the Boolean expression B. If B is false, the loop terminates immediately, otherwise S is executed. If S terminates, the whole procedure is repeated.

Given a **while**-program S we denote by $\mathit{var}(S)$ the set of all simple and array variables which appear in it and by $\mathit{change}(S)$ the set of all simple and array variables which appear in it on a left-hand side of an assignment; $\mathit{change}(S)$ is the set of variables which can be modified by S. Both notions will be used later in the chapters on parallel programs.

By a *subprogram* S of a **while**-program R we mean a substring S of R which again is a **while**-program. For example,

$$S \equiv x := x - 1$$

is a subprogram of

$$R \equiv \textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } y := y - x;\ x := x - 1 \textbf{ fi}.$$

3.2 Proof Theory

In this section we consider correctness formulas of the form $\{p\}\ S\ \{q\}$ where S is a **while**-program and p and q are assertions. The assertion p, usually called a *precondition*, specifies the initial, or input, conditions to be satisfied by the variables of S. The assertion q, usually called a *postcondition*, specifies the final, or output, conditions satisfied by the variables of S. Thus the pair p, q can be viewed as an *input-output specification* of the program S.

More precisely, we are interested in two interpretations of correctness formulas. We say that $\{p\}\ S\ \{q\}$ is true in the sense of *partial correctness* if every terminating computation of S starting in a state satisfying p terminates in a state satisfying q. And $\{p\}\ S\ \{q\}$ is true in the sense of *total correctness* if every computation of S starting in a state satisfying p terminates and its final state satisfies q.

Partial correctness

We now present a proof system, called PD, for deriving partial correctness formulas about deterministic programs. It was introduced in Hoare [1969]. It consists of the following axioms and proof rules.

AXIOM 1: SKIP

$$\{p\}\ skip\ \{p\}$$

AXIOM 2: ASSIGNMENT

$$\{p[t/u]\}\ u := t\ \{p\}$$

RULE 3: COMPOSITION

$$\frac{\{p\}\ S_1\ \{r\}, \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

RULE 4: CONDITIONAL

$$\frac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$$

RULE 5: LOOP

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

RULE 6: CONSEQUENCE

$$\frac{p \rightarrow p_1, \{p_1\}\ S\ \{q_1\}, q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

The *skip* axiom should be obvious. On the other hand, the first reaction to the assignment axiom is usually astonishment. The axiom encourages reading the assignment "backwards" and initially we have no intuition associated with such a view. So before we proceed further let us first analyze a simple program consisting only of assignments.

Example 3.1 Consider the following program S:

$$x := 1;\ a[1] := 2;\ a[x] := x.$$

We now prove that after the execution of S the value of $a[1]$ is 1, that is, we prove in PD the correctness formula

$$\{\mathbf{true}\}\ S\ \{a[1] = 1\}.$$

To this purpose we repeatedly apply the assignment axiom while proceeding "backwards". We start with

$$\{(a[1] = 1)[x/a[x]]\}\ a[x] := x\ \{a[1] = 1\},$$

that is, by the definition of substitution,

$$\{\mathbf{if}\ 1 = x\ \mathbf{then}\ x\ \mathbf{else}\ a[1]\ \mathbf{fi}\ = 1\}\ a[x] := x\ \{a[1] = 1\},$$

which after simplifications (formally justified by the consequence rule) gives

$$\{x \neq 1 \rightarrow a[1] = 1\}\ a[x] := x\ \{a[1] = 1\}.$$

By the same token

$$\{x \neq 1 \rightarrow 2 = 1\}\ a[1] := 2\ \{x \neq 1 \rightarrow a[1] = 1\},$$

that is, by the consequence rule,

$$\{x = 1\}\ a[1] := 2\ \{x \neq 1 \rightarrow a[1] = 1\}.$$

Finally,

$$\{\mathbf{true}\}\ x := 1\ \{x = 1\}.$$

Putting the last correctness formulas together using the composition rule twice we get the desired result. □

Another stumbling block in the understanding of the above proof system might be the loop rule. This rule states that given the program **while** B **do** S **od**, if p is preserved with each iteration of its loop, then p is true upon exit of the program. Therefore p is called a *loop invariant.*

Let us see how this rule can be used. We choose here as an example the first program (written in a textual form) which was formally verified. This historic event was duly documented in Hoare [1969].

Example 3.2 Let S be the following program computing the integer quotient and remainder of two natural numbers x and y:

$$S \equiv quo := 0;\ rem := x;\ S_0,$$

where S_0 is

$$\textbf{while } rem \geq y \textbf{ do } rem := rem - y;\ quo := quo + 1 \textbf{ od}.$$

We wish to prove

$$\{x \geq 0 \wedge y \geq 0\}\ S\ \{quo \cdot y + rem = x \wedge 0 \leq rem < y\}, \tag{1}$$

that is,

$$\begin{array}{l}\text{if, } x, y \text{ are nonnegative integers and } S \text{ terminates,}\\ \text{then } quo \text{ is the integer quotient}\\ \text{of } x \text{ divided by } y \text{ and } rem \text{ is the remainder.}\end{array} \tag{2}$$

Note that the interpretation (2) of (1) is only true because S does not change the variables x and y. Programs that may change x and y can trivially achieve (1) without satisfying (2). An example is the program

$$S \equiv x := 0;\ y := 1;\ q := 0;\ r := 0.$$

To prove (1), we choose the assertion

$$p \equiv quo \cdot y + rem = x \wedge rem \geq 0$$

as the loop invariant of S_0. It is obtained from the postcondition of (1) by dropping the formula $rem < y$.

We now prove the following three facts:

1°. $\{x \geq 0 \wedge y \geq 0\}\ quo := 0;\ rem := x\ \{p\}$,

i.e., the program $quo := 0;\ rem := x$ establishes p.

2°. $\{p \wedge rem \geq y\}\ rem := rem - y;\ quo := quo + 1\ \{p\}$,

i.e., p is indeed a loop invariant of S_0;

3°. $p \wedge \neg(rem \geq y) \rightarrow quo \cdot y + rem = x \wedge 0 \leq rem < y$,

i.e., upon exit of the loop S_0, p implies the desired assertion.

Observe first that we can prove (1) from 1°, 2° and 3°. Indeed, 2° implies, by the loop rule,

$$\{p\}\ S_0\ \{p \wedge \neg(rem \geq y)\}.$$

This, together with 1°, implies, by the composition rule,

$$\{x \geq 0 \wedge y \geq 0\}\ S\ \{p \wedge rem < y\}.$$

Now, by 3°, (1) holds by an application of the consequence rule.
So let us prove now 1°, 2° and 3°.

ad 1°.

We have

$$\{quo \cdot y + x = x \wedge x \geq 0\}\ rem := x\ \{p\}$$

by the assignment axiom. Once more by the assignment axiom

$$\{0 \cdot y + x = x \wedge x \geq 0\}\ quo := 0\ \{quo \cdot y + x = x \wedge x \geq 0\},$$

so by the composition rule

$$\{0 \cdot y + x = x \wedge x \geq 0\}\ quo := 0;\ rem := x\ \{p\}.$$

On the other hand

$$x \geq 0 \wedge y \geq 0 \rightarrow 0 \cdot y + x = x \wedge x \geq 0$$

so 1° holds by the consequence rule.

ad 2°.

We have

$$\{(quo + 1) \cdot y + rem = x \wedge rem \geq 0\}\ quo := quo + 1\ \{p\}$$

by the assignment axiom. Once more by the assignment axiom

$$\begin{array}{l}\{(quo + 1) \cdot y + (rem - y) = x \wedge rem - y \geq 0\} \\ rem := rem - y \\ \{(quo + 1) \cdot y + rem = x \wedge rem \geq 0\},\end{array}$$

so by the composition rule

$$\{(quo+1)\cdot y+(rem-y) = x \wedge rem-y \geq 0\}\ rem := rem-y;\ quo := quo+1\ \{p\}.$$

On the other hand

$$p \wedge rem \geq y \rightarrow (quo + 1) \cdot y + (rem - y) = x \wedge rem - y \geq 0,$$

so 2° holds by the consequence rule.

ad 3°.

Clear.

This completes the proof of (1). □

The only step in the above proof which required some creativity was finding the appropriate loop invariant. The remaining steps were straightforward applications of the corresponding axioms and proof rules. The form of the assignment axiom makes it easier to deduce a pre-assertion from a post-assertion than the other way around, so we proceeded in the proofs of 1° and 2° "backwards". Finally, we did not provide any formal proof of the implications used as premises of the consequence rule. Formal proofs of such assertions in some proof system which includes arithmetic will always be omitted; we shall simply rely on an intuitive understanding of their truth.

Total correctness

It is important to note that the proof system PD does not allow us to establish the termination of programs, i.e., it is not appropriate for proofs of total correctness. Even though we proved in Example 3.2 the correctness formula (1), we cannot infer from this fact that the program S studied there terminates. In fact, S diverges when started in a state in which the value of y is 0.

Clearly the only proof rule of PD which introduces possibility of nontermination is the loop rule, so to deal with total correctness this rule has to be strengthened.

We now introduce the following refinement of the loop rule.

RULE 7: LOOP II

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

where t is an integer expression and z is an integer variable which does not appear in p, B, t or S.

The two additional premises of the rule guarantee termination of the loop. By the second premise t is decreased with each iteration and by the third premise t is non-negative if another iteration can be performed. Thus no infinite computation is possible. The expression t is called a *bound function* of the program **while** B **do** S **od**. The purpose of z is to retain the initial value of t.

Let TD denote the proof system obtained from PD by replacing the loop rule by the loop II rule. TD is an appropriate proof system for proving total correctness of **while**-programs. To see an application of the loop II rule let us reconsider the program S studied in Example 3.2.

Example 3.3 We now prove

$$\{x \geq 0 \wedge y > 0\}\ S\ \{quo \cdot y + rem = x \wedge 0 \leq rem < y\} \tag{3}$$

in the sense of total correctness, that is that

> if x is nonnegative and y is a positive integer, then
> S terminates with *quo* being the integer quotient
> of x divided by y and *rem* being the remainder. (4)

Note that (3) differs from the correctness formula (1) from Example 3.2 by requiring that initially $y > 0$. For this purpose it is sufficient to modify appropriately the proof of (1). Let

$$p' \equiv p \wedge y > 0$$

be a new loop invariant and let

$$t \equiv rem$$

be the bound function. As in the proof given in Example 3.2, to prove (3) in the sense of total correctness it is sufficient to establish the following facts:

1°. $\{x \geq 0 \wedge y > 0\}\ quo := 0;\ rem := x\ \{p'\}$,

2°. $\{p' \wedge rem \geq y\}\ rem := rem - y;\ quo := quo + 1\ \{p'\}$,

3°. $\{p' \wedge rem \geq y \wedge rem = z\}\ rem := rem - y;\ quo := quo + 1\ \{rem < z\}$,

4°. $p' \rightarrow rem \geq 0$,

5°. $p' \wedge \neg(rem \geq y) \rightarrow quo \cdot y + rem = x \wedge 0 \leq rem < y$.

Indeed, 2° 3° and 4° imply by the loop II rule $\{p'\}\ S_0\ \{p' \wedge \neg(rem \geq y)\}$ and the rest of the argument is the same as in Example 3.2. Proofs of 1°, 2° and 5° are analogous to the proofs of 1°, 2° and 3° in Example 3.2.

To prove 3° observe that by the assignment axiom

$$\{rem < z\}\ quo := quo + 1\ \{rem < z\}$$

and

$$\{(rem - y) < z\}\ rem := rem - y\ \{rem < z\}.$$

But

$$p \wedge y > 0 \wedge rem \geq y \wedge rem = z \rightarrow (rem - y) < z,$$

so 3° holds by the consequence rule.

Finally, 4° clearly holds. This concludes the proof. □

3.3 Proof Outlines

Formal proofs are tedious to follow. We are not accustomed to following a line of reasoning presented in small, formal steps. A better solution consists of a logical organization of the proof with the main steps isolated. The proof can then be seen on a different level.

In the case of correctness proofs of **while**-programs a possible strategy lies in using the fact that they are structured. The proof rules we introduced follow the syntax of the programs, so the structure of the program can be used to structure the correctness proof. We can simply present the proof by giving a program with assertions interleaved at appropriate places.

Partial correctness

Example 3.4 Let us reconsider the integer division program studied in Example 3.2. We present facts 1°, 2° and 3° in the following form:

$\{x \geq 0 \wedge y \geq 0\}$
$quo := 0;\ rem := x;$
$\{\mathbf{inv} : p\}$
while $rem \geq y$ **do**
    $\{p \wedge rem \geq y\}$
    $rem := rem - y;\ quo := quo + 1$
od
$\{p \wedge rem < y\}$
$\{quo \cdot y + rem = x \wedge 0 \leq rem < y\},$

where

$$p \equiv quo \cdot y + rem = x \wedge rem \geq 0.$$

The keyword **inv** is used here to label the loop invariant. Two adjacent assertions $\{q_1\}\{q_2\}$ stand for the fact that the implication $q_1 \rightarrow q_2$ is true.

The proofs of the facts can also be presented in such a form. For example, here is the proof of fact 1°:

$\{x \geq 0 \wedge y \geq 0\}$
$\{0 \cdot y + x = x \wedge x \geq 0\}$
$quo := 0$
$\{quo \cdot y + x = x \wedge x \geq 0\}$
$rem := x$
$\{p\}.$

□

Such a proof presentation is much simpler to study and analyze. It was introduced in Owicki and Gries [1976] and is called a *proof outline*. It is

formally defined as follows.

Definition 3.5 (Proof outline: partial correctness) Let S^* stand for the program S interspersed, or as we shall say *annotated*, with assertions, some of them labelled by the keyword **inv**. We define the notion of a *proof outline for partial correctness* inductively by the following axioms and rules.

An axiom φ should be read here as a statement: φ is a proof outline (for partial correctness) and a rule

$$\frac{\varphi_1, \ldots, \varphi_n}{\varphi}$$

should be read as a statement: if $\varphi_1, \ldots, \varphi_n$ are proof outlines, then φ is a proof outline.

(i)

$$\{p\}\ skip\ \{p\}$$

(ii)

$$\{p[t/u]\}\ u := t\ \{p\}$$

(iii)

$$\frac{\{p\}\ S_1^*\ \{r\}, \{r\}\ S_2^*\ \{q\}}{\{p\}\ S_1^*\ ;\ \{r\}\ S_2^* \{q\}}$$

(iv)

$$\frac{\{p \wedge B\}\ S_1^* \{q\}, \{p \wedge \neg B\}\ S_2^* \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ \{p \wedge B\}\ S_1^*\ \{q\}\ \textbf{else}\ \{p \wedge \neg B\}\ S_2^*\ \{q\}\ \textbf{fi}\ \{q\}}$$

(v)

$$\frac{\{p \wedge B\}\ S^*\ \{p\}}{\{\textbf{inv} : p\}\ \textbf{while}\ B\ \textbf{do}\ \{p \wedge B\}\ S^*\ \{p\}\ \textbf{od}\ \{p \wedge \neg B\}}$$

(vi)

$$\frac{p \to p_1, \{p_1\}\ S^* \{q_1\}, q_1 \to q}{\{p\}\{p_1\}\ S^* \{q_1\}\{q\}}$$

(vii)

$$\frac{\{p\}\ S^* \{q\}}{\{p\}\ S^{**}\ \{q\}}$$

where S^{**} results from S^* by omitting some of the intermediate assertions not labelled by the keyword **inv**.

Thus in a proof outline some of the intermediate assertions used in the correctness proof are retained; loop invariants are always kept. A proof outline $\{p\}\ S^*\ \{q\}$ for partial correctness is called *standard* if every substatement T of S is preceded by exactly one assertion in S^*, called $pre(T)$, and there are no other assertions in S^*. □

Thus every standard proof outline $\{p\}\ S^*\ \{q\}$ starts with exactly 2 assertions, namely p and $pre(S)$. If $p \equiv pre(S)$, then we drop p from this proof outline.

Note that a standard proof outline is not minimal in the sense that some assertions used in it can be removed. For example, the assertion $\{p \wedge B\}$ in the context $\{\mathbf{inv} : p\}$ **while** B **do** $\{p \wedge B\}\ S$ **od** $\{q\}$ can be deduced. Standard proof outlines will be needed in the chapters on parallel programs.

By studying proofs of partial correctness in the form of proof outlines we do not lose any generality in the sense of the following lemma.

Lemma 3.6 Let $\{p\}\ S^*\ \{q\}$ be a proof outline for partial correctness. Then there exists a proof of $\{p\}\ S\ \{q\}$ in the proof system PD. □

Also, the proof outlines $\{p\}\ S^*\ \{q\}$ enjoy the following useful and intuitive property: whenever the control of S in a given computation starting in a state satisfying p reaches a point annotated by an assertion, this assertion is true. Thus the assertions of a proof outline are true at the appropriate moments.

Total correctness

So far we have only discussed proof outlines for partial correctness. To complete the picture we should take care of the termination of loops.

Consider a loop **while** B **do** S **od**. The loop II rule suggests a rule for a proof outline for total correctness of loops whose premises are of the form $\{p \wedge B\}\ S^*\ \{p\}$, $\{p \wedge B \wedge t = z\}\ S^{**}\ \{t < z\}$, $p \rightarrow t \geq 0$ with the first two being proof outlines for total correctness. However, there is no obvious way to record both proof outlines in the conclusion of such a rule.

One solution is to start with a modification of the loop II rule whose first two premises are replaced by

$$\{p \wedge B \wedge t = z\}\ S\ \{p \wedge t < z\}$$

and introduce the following rule for a proof outline for total correctness

$$\frac{\{p \wedge B \wedge t = z\}\ S^*\ \{p \wedge t < z\},\quad p \rightarrow t \geq 0}{\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}\ \mathbf{while}\ B\ \mathbf{do}\ \{p \wedge B \wedge t = z\}\ S^*\ \{p \wedge t < z\}\ \mathbf{od}\ \{p \wedge \neg B\}}$$

where t is an integer expression and z is an integer variable not occuring in p, t, B or S^*.

This rule, however, forces us to mix the proofs of the invariance of p and of the decrease of t.

Another solution, which we adopt here, is to assume that the proof of decrease of t is of a particularly simple form, namely that

(i) all assignments inside S decrease t or leave it unchanged,

(ii) on each syntactically possible path through S at least one assignment decreases t.

By a *path* we mean here a possibly empty finite sequence of assignments. Sequential composition $\pi_1;\ \pi_2$ of paths π_1 and π_2 is lifted to sets Π_1 and Π_2 of paths by putting

$$\Pi_1;\ \Pi_2 = \{\pi_1;\ \pi_2 \mid \pi_1 \in \Pi_1 \text{ and } \pi_2 \in \Pi_2\}.$$

By ε we denote the empty sequence. For any path π we have $\pi;\ \varepsilon = \varepsilon;\ \pi = \pi$.

Next we define the path set for a while-program.

Definition 3.7 Let S be a while-program. We define $path(S)$ by induction on the structure of S:

- $path(skip) = \{\varepsilon\}$,
- $path(u := t) = \{u := t\}$,
- $path(S_1;\ S_2) = path(S_1);\ path(S_2)$,
- $path(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = path(S_1) \cup path(S_2)$,
- $path(\textbf{while } B \textbf{ do } S_1 \textbf{ od}) = \{\varepsilon\}$.

□

Intuitively, $path(S)$ is a set of all paths through S. Each path through S is identified with the sequence of assignments lying on it. Note that in the last clause we only take into account the case when the loop is exited immediately. This is sufficient for establishing the condition (ii) above.

Definition 3.8 (Proof outline: total correctness) The notion of a *(standard) proof outline for total correctness* is defined as for partial correctness, except for rule (v) dealing with loops. It is to be replaced by:

(v′)

$$\frac{\begin{array}{l}\{p \wedge B\}\ S^*\ \{p\},\\ \{pre(T) \wedge t = z\}\ T\ \{t \leq z\} \text{ for each}\\ \quad \text{assignment } T \text{ within } S,\\ \text{for each } \pi \in path(S) \text{ there exists an}\\ \text{assignment } T \text{ in } \pi \text{ such that}\\ \{pre(T) \wedge t = z\}\ T\ \{t < z\},\\ p \rightarrow t \geq 0\end{array}}{\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}\ \mathbf{while}\ B\ \mathbf{do}\ \{p \wedge B\}\ S^*\ \{p\}\ \mathbf{od}\ \{p \wedge \neg B\}}$$

where t is an integer expression and z is an integer variable not occuring in p, t, B and S^*. Here $\{p\}\ S^*\ \{q\}$ is a standard proof outline for total correctness and $pre(T)$ stands for the assertion preceding T in this proof outline. □

The annotation $\{\mathbf{bd} : t\}$ represents the bound function of the loop **while** B **do** S **od**. Note that clause (vii) in the definition of a proof outline for total correctness does not allow us to delete the bound functions.

Example 3.9 The following is a proof outline for total correctness of the integer division program studied in Example 3.3:

$\{x \geq 0 \wedge y > 0\}$
$quo := 0;\ rem := x;$
$\{\mathbf{inv} : p'\}\{\mathbf{bd} : rem\}$
while $rem \geq y$ **do**
  $\{p' \wedge rem \geq y\}$
  $rem := rem - y;\ quo := quo + 1$
  $\{p'\}$
od
$\{p' \wedge rem < y\}$
$\{quo \cdot y + rem = x \wedge 0 \leq rem < y\},$

where

$$p' \equiv quo \cdot y + rem = x \wedge rem \geq 0 \wedge y > 0.$$

Note that, due to the precondition $p' \wedge rem \geq y$, the assignment $rem := rem - y$ decreases the bound function rem, whereas the assignment $quo := quo + 1$ leaves rem unchanged. □

Note that when the empty path ε is an element of $path(S)$, we cannot verify the pre-last condition of the above rule (v′). Thus it may happen that we can prove total correctness of a **while**-program but we shall not be able

to present this proof in the form of a proof outline for total correctness. An example is the program

```
b := true;
while b do
    if b then b := false
od
```

whose termination can be easily established. This shows some limitations of the above approach to proof outlines for total correctness. However, all the programs discussed in this article can be handled in this way.

3.4 Derived Rules

The presentation of correctness proofs can be simplified in another way — by means of *derived rules*. They allow us to prove certain correctness formulas about the same program separately and then combine them. This can lead to a different organization of the correctness proof.

These rules for combining correctness formulas are not necessary, in the sense that their use in the correctness proof can be eliminated by applying other rules. That is why they are called derived rules. These rules are appropriate both for partial correctness and total correctness and can be used for all classes of programs considered in this paper. We shall use them in Chapter 7 when studying parallel programs.

RULE D1: DISJUNCTION

$$\frac{\{p\}\ S\ \{q\}, \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

RULE D2: $\exists$-INTRODUCTION

$$\frac{\{p\}\ S\ \{q\}}{\{\exists x p\}\ S\ \{q\}}$$

where x does not appear in S or q.

3.5 Conclusions

In what sense are the proof systems PD and TD natural for the correctness proofs of while-programs? Their important feature is that they are *syntax directed*, that is, their proof rules follow the syntax of the programming constructs. This allowed us to organize the proofs in a form that follows the program structure. This, in turn, makes them easier to understand and allows us to be less formal in their presentation.

None of this would be possible if the proofs were presented in a formalism not referring to the programs. Consider, for example, a natural translation of the correctness formulas into, say, Peano arithmetic. Even though one can, in principle, consider proofs of the translated formulas in Peano arithmetic, it is clear that they will not be easy to construct and understand. The reason is that in Peano arithmetic, or in any other proof system studied in mathematical logic, the formulas expressing program correctness do not naturally reflect the meaning of the program and are consequently difficult to study.

However, once a program is already written, it is usually *too late* to prove its correctness because all helpful intuitions present in its development process have disappeared, and only the final product is available! A reconstruction of these intuitions is a very tedious, if not impossible, process. Moreover, the proof has nothing to do with the process of the development of the program — it only documents the final result. We thus deal with two disjoint activities, namely development and proving, addressing the same intuitions.

This problem was recognized and addressed in Dijkstra [1976] who proposed to develop the program *together* with its correctness proof with the intention of simplifying both tasks. This approach will be discussed in the next chapter.

4 Nondeterministic Programs

Activating a deterministic program in a certain state will generate exactly one computation sequence. Often such a level of detail is unnecessary, for example when two different computation sequences yield the same final state. The phenomenon that a program may generate more than one computation sequence from a given state is called *nondeterminism.* In this chapter we will study a toy programming language due to Dijkstra [1975, 1976] which allows us to write programs with such a behaviour.

In Chapter 8 this class of programs will allow us to study distributed programs.

4.1 Syntax

We expand the grammar for **while**-programs by adding:

- *alternative commands*

$$S ::= \textbf{if } B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n \textbf{ fi},$$

- *repetitive commands*

$$S ::= \textbf{do } B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n \textbf{ od}.$$

These new commands will also be written as

$$\textbf{if } \Box_{i=1}^{n}\ B_i \to S_i \textbf{ fi and do } \Box_{i=1}^{n}\ B_i \to S_i \textbf{ od},$$

respectively. A command S_i within S is said to be *guarded* by the Boolean expression B_i. The construct $B_i \to S_i$ is therefore called a *guarded command.*

The symbol $\Box$ represents a nondeterministic choice between guarded commands $B_i \to S_i$. More precisely, in the context of an alternative command

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

a guarded command $B_i \to S_i$ can be chosen only if its guard B_i evaluates to true; then S_i remains to be executed. If more than one guard B_i evaluates to true *any* of the corresponding statements S_i may be executed next. There is no rule saying which statement should be selected. If all guards evaluate to false, the alternative command will signal a *failure*, also called *abortion.*

The selection of guarded commands in the context of a repetitive command

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

is performed in a similar way. The difference is that after termination of a selected statement S_i the whole command is repeated starting with a new evaluation of the guards B_i. Moreover, contrary to the alternative command, the repetitive command properly terminates when all guards evaluate to false.

Conventional conditionals and loops can be modelled by alternative and repetitive commands because

$$\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

is equivalent to

$$\textbf{if } B \to S_1 \ \Box\ \neg B \to S_2 \textbf{ fi}$$

and

$$\textbf{while } B \textbf{ do } S \textbf{ od}$$

is equivalent to

$$\textbf{do } B \to S \textbf{ od}.$$

The notion of a subprogram of a nondeterministic program is defined as in Chapter 3. Let us discuss now the main features of guarded commands.

Symmetry

Guarded commands allow us to present Boolean tests in a symmetric manner. This often enhances the clarity of programs. For example, instead of writing

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

the well-known algorithm for finding the *g*reatest common *d*ivisor (gcd) of two natural numbers can now be expressed as

$$\textbf{do } x > y \to x := x - y \;\square\; x < y \to y := y - x \textbf{ od}.$$

Failures

Remember that an alternative command fails rather than terminates if none of the guards evaluate to true. Thus, in general, **if** $B \to S$ **fi** and **if** B **then** S **fi** differ because failures signal exceptional states of computation. For example,

$$\textbf{if } 0 \leq i < n \to x := a[i] \textbf{ fi}$$

raises a failure before the array a can be accessed outside the interval $\{0, \ldots, n-1\}$. Such guarded assignments are useful to model access to bounded arrays.

Nondeterminism

Guarded commands allow us to express nondeterminism through the use of non-exclusive guards. As an example, consider the following program computing the largest powers of 2 and 3 that divide a given integer x in which the division function / is used:

$twop := 0$; $threep := 0$;
do 2 divides $x \to x := x/2$; $twop := twop + 1$
$\square$ 3 divides $x \to x := x/3$; $threep := threep + 1$
od.

If 6 divides x, both guards can be chosen. In fact, it does not matter which one will be chosen—the final values of the variables *twop* and *threep* will always be the same.

4.2 Proof Theory

As in the previous chapter we are now interested in two notions of program correctness—partial correctness and total correctness. Their definitions are the same as before. However, when studying total correctness we should be aware that a computation can now fail to terminate for two reasons: divergence or abortion. This will be reflected in two differences between the proof systems for partial and total correctness.

To see the difference between partial correctness and total correctness resulting from abortion, consider (once more) the program

$$S \equiv \textbf{if } 0 \leq i < n \rightarrow x := a[i] \textbf{ fi}.$$

Then $\{\textbf{true}\}\ S\ \{x = a[i]\}$ holds in the sense of partial correctness but not in the sense of total correctness because S fails when activated in a state not satisfying $0 \leq i < n$.

We first present a proof system PN for partial correctness of nondeterministic programs. PN includes Axioms 1 and 2 and Rules 3 and 6 introduced for PD, the system for partial correctness of deterministic programs. But Rules 4 and 5 of PD are now replaced by:

RULE 8: ALTERNATIVE COMMAND

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \textbf{if } \Box_{i=1}^{n}\ B_i \rightarrow S_i \textbf{ fi}\ \{q\}}$$

RULE 9: REPETITIVE COMMAND

$$\frac{\{p \wedge B_i\}\ S_i\ \{p\},\ i = 1, \ldots, n}{\{p\}\ \textbf{do } \Box_{i=1}^{n}\ B_i \rightarrow S_i \textbf{ od}\ \{p \wedge \wedge_{i=1}^{n} \neg B_i\}}$$

A system TN for total correctness results from PN by strengthening Rule 8 to the following rule:

RULE 10: ALTERNATIVE COMMAND II

$$\frac{\begin{array}{l} p \rightarrow \vee_{i=1}^{n}\ B_i, \\ \{p \wedge B_i\}\ S_i\ \{q\},\ i = 1, \ldots, n \end{array}}{\{p\}\ \textbf{if } \Box_{i=1}^{n}\ B_i\ \rightarrow S_i \textbf{ fi}\ \{q\}}$$

and by replacing Rule 9 by:

RULE 11: REPETITIVE COMMAND II

$$\frac{\begin{array}{l} \{p \wedge B_i\}\ S_i\ \{p\},\ i = 1, \ldots, n \\ \{p \wedge B_i \wedge t = z\}\ S_i\ \{t < z\}, i = 1, \ldots, n \\ p \rightarrow t \geq 0,\ i = 1, \ldots, n \end{array}}{\{p\}\ \textbf{do } \Box_{i=1}^{n}\ B_i \rightarrow S_i \textbf{ od}\ \{p \wedge \wedge_{i=1}^{n} \neg B_i\}}$$

where t is an integer expression and z is an integer variable not appearing in p, t, B_i or $S_i, i = 1, \ldots, n$.

As in Chapter 3 we shall present correctness proofs in the form of proofs outlines. We leave their definitions to the reader.

Example 4.1 The following is a proof outline for total correctness of the symmetric gcd program mentioned in the previous section.

$$
\begin{array}{l}
\{x = a \wedge y = b \wedge a > 0 \wedge b > 0\} \\
\{\mathbf{inv} : p\}\{\mathbf{bd} : t\} \\
\mathbf{do}\ x > y \rightarrow \{p \wedge x > y\} \\
\qquad\qquad x := x - y \\
\Box\ \ x < y \rightarrow \{p \wedge x < y\} \\
\qquad\qquad x := y - x \\
\mathbf{od} \\
\{p \wedge \neg(x > y) \wedge \neg(x < y)\} \\
\{x = y \wedge y = gcd(a, b)\}
\end{array}
$$

As an invariant we use here

$$p \equiv gcd(x, y) = gcd(a, b) \wedge x > 0 \wedge y > 0$$

where the binary function symbol gcd is to be interpreted as "greatest common divisor of" and where the fresh variables a and b represent the initial values of x and y. As a bound function we use

$$t \equiv x + y.$$

Note that due to their preconditions, each assignment decreases the bound function. □

As before proof outlines $\{p\}\ S^*\ \{q\}$ for partial correctness enjoy the following property: whenever the control of S in a given computation started in a state satisfying p reaches a point annotated by an assertion, this assertion is true.

4.3 Development of provably correct programs

We now discuss an approach of Dijkstra [1976] allowing us to develop programs together with their correctness proofs. To this purpose, we shall make use of the proof system TN to guide us in the construction of a program. All correctness formulas are supposed to hold in the sense of total correctness.

The main issue in Dijkstra's approach is the development of loops. Suppose we want to find a program R of the form

$$T;\ \mathbf{do}\ B \rightarrow S\ \mathbf{od}$$

that satisfies, for a given preconditon r and postcondition q, the correctness formula

$$\{r\}\ R\ \{q\}. \tag{1}$$

To avoid trivial solutions for R (cf. the comment in Example 3.2), we usually postulate that some variables in r and q, say $x_1, \ldots, x_n$, may not be modified by R, i.e. we require

$$x_1, \ldots, x_n \notin change(R).$$

To prove (1), it is sufficient to find a loop invariant p and a bound function t satisfying the following five conditions:

1°. p is initially established, i.e., $\{r\}\ T\ \{p\}$ holds;

2°. p is a loop invariant, i.e., $\{p \wedge B\}\ S\ \{p\}$ holds;

3°. upon loop termination q is true, i.e., $p \wedge \neg B \rightarrow q$;

4°. p implies $t \geq 0$, i.e., $p \rightarrow t \geq 0$;

5°. t is decreased with each iteration, i.e., $\{p \wedge B \wedge t = z\}\ S\ \{t < z\}$ holds where z is a fresh variable.

Of course, analogous conditions can be provided when the loop in R can have more than one guard. Conditions 1°-5° can be conveniently presented by the following proof outline for total correctness:

$\{r\}$
T;
$\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}$
do $B \rightarrow \{p \wedge B\}$
S
$\{p\}$
od
$\{p \wedge \neg B\}$
$\{q\}$.

(Here we assume that condition 5° can be proved by establishing the appropriate conditions listed in the premises of the rule for proof outlines for total correctness of repetitive commands.)

Now, when only r and q are known, the first step in finding R consists of finding a loop invariant. One useful strategy consists of generalizing the postcondition q by replacing a constant by a variable. The following toy example illustrates the point.

A simple summation problem

The problem is to find a program S which stores in an integer variable x the sum of the elements of a given section $a[0 : n-1]$ of an integer array a. Here n is a constant with $n \geq 0$. By definition, the sum is 0 if $n = 0$. Of course, we require that $a, n \notin change(S)$. Define now

$$r \equiv n \geq 0$$

and

$$q \equiv x = \Sigma_{i=0}^{n-1}\ a[i].$$

The assertion q states that x stores the sum of the elements of the section $a[0:n-1]$.

We replace the constant n by a fresh variable k. Putting appropriate bounds on k we obtain

$$p \equiv 0 \leq k \leq n \wedge x = \Sigma_{i=0}^{k-1}\ a[i]$$

as a proposal for the invariant of the program to be developed.

We now attempt to satisfy the conditions 1-5 by choosing B, S and t appropriately.

ad 1°. p is easily established by the command

$$T \equiv k := 0;\ x := 0.$$

ad 3°. We clearly have $p \wedge k = n \rightarrow q$, so we can take $k \neq n$ as the guard of the loop.

ad 4°. We have $p \rightarrow n - k \geq 0$, which suggests to choose

$$t \equiv n - k$$

as the bound function.

ad 5°. To reduce the bound function with each iteration, we choose the program $k := k + 1$ as part of the loop body.

ad 2°. Thus far we have the following incomplete proof outline

$\{r\}$
$k := 0;\ x := 0;$
$\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}$
do $k \neq n \rightarrow$ $\{p \wedge k \neq n\}$
$S_1;$
$\{p[k+1/k]\}$
$k := k + 1$
od
$\{p \wedge k = n\}$
$\{q\}$

where S_1 is still to be found.

To this end, we compare now the precondition and postcondition of S_1. The precondition $p \wedge k \neq n$ implies

$$0 \leq k + 1 \leq n \wedge x = \Sigma_{i=0}^{k-1}\ a[i]$$

and the postcondition $p[k+1/k]$ is equivalent to

$$0 \leq k+1 \leq n \wedge x = (\Sigma_{i=0}^{k-1}\ a[i]) + a[k].$$

We see that adding $a[k]$ to x will "transform" one assertion into another. Thus, we can choose

$$S_1 \equiv x := x + a[k]$$

to ensure that p is a loop invariant.

Summarizing, we have developed the following program together with its correctness proof.

$k := 0;\ x := 0;$
do $k \neq n \rightarrow x := x + a[k];$
$\qquad\qquad k := k+1$
od

The next example will illustrate another strategy in the development of correct programs.

The welfare crook problem

We now study the following problem due to W. Feijen, where a nondeterministic program seems more appropriate as a solution. We follow here the exposition of Gries [1982]. Given are three magnetic tapes, each containing a list of names in alphabetical order. The first contains the names of people working at IBM Yorktown Heights, the second the names of students at Columbia University, and the third the names of people on welfare in New York City. Practically speaking, all three lists are endless, so no upper bounds are given. It is known that at least one person is on all three lists. The problem is to write a program to locate the alphabetically first such person.

Slightly more abstract, we consider three *ordered arrays* a, b, c of type **integer** $\rightarrow$ **integer**, i.e. such that $i < j$ implies $a[i] < a[j]$, and similarly for b and c. We suppose that there exist values $iv \geq 0, jv \geq 0$, and $kv \geq 0$ such that

$$a[iv] = b[jv] = c[kv]$$

holds, and moreover we suppose that the triple (iv, jv, kv) is the smallest one in the lexicographic ordering among those ones satisfying this condition. The values iv, jv and kv can be used in the assertions but *not* in the program. We are supposed to develop a program which computes them.

Thus our precondition r is a list of the assumed facts — that a, b, c are ordered together with the formal definition of iv, jv and kv. We omit the formal definition. The postcondition is

$$q \equiv i = iv \wedge j = jv \wedge k = kv.$$

Additionally we require $a, b, c, iv, jv, kv \notin change(S)$, where S is the program to be found. Assuming that the search starts from the beginning of the lists, we are brought to the following invariant by placing appropriate bounds on i, j and k:

$$p \equiv 0 \leq i \leq iv \wedge 0 \leq j \leq jv \wedge 0 \leq k \leq kv \wedge r.$$

A natural choice for the bound function is:

$$t \equiv (iv - i) + (jv - j) + (kv - k).$$

The invariant is easily established by

$$i := 0;\ j := 0;\ k := 0.$$

The simplest ways to decrease the bound functions are the assignments $i := i + 1, j := j + 1$ and $k := k + 1$. In general, it will be necessary to increment all three variables, so we arrive at the following incomplete proof outline:

$\{r\}$
$i := 0;\ j := 0;\ k := 0;$
$\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}$
do $B_1 \rightarrow \{p \wedge B_1\}\ i := i + 1$
$\square$ $B_2 \rightarrow \{p \wedge B_2\}\ j := k + 1$
$\square$ $B_3 \rightarrow \{p \wedge B_3\}\ k := k + 1$
od
$\{p \wedge \neg B_1 \wedge \neg B_2 \wedge \neg B_3\}$
$\{q\}$

where B_1, B_2 and B_3 are still to be found. Of course the simplest choice for B_1, B_2 and B_3 are, respectively, $i \neq iv, j \neq jv$ and $k \neq kv$ but the values iv, jv and kv cannot be used in the program. On the other hand, $p \wedge i \neq iv$ is equivalent to $p \wedge i < iv$ which means by the definition of iv, jv and kv that $a[i]$ is not the crook. Now, assuming p, the last statement is guaranteed if $a[i] < b[j]$. Indeed, a, b and c are ordered, so $p \wedge a[i] < b[j]$ implies $a[i] < b[jv] = a[iv]$ which implies $i < iv$.

We can thus choose $a[i] < b[j]$ for the guard B_1. In a similar fashion we can choose the other two guards which yield the following program and a proof outline

$\{r\}$
$i := 0;\ j := 0;\ k = 0;$
$\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}$
do $a[i] < b[j] \rightarrow \{p \wedge a[i] < b[j]\}$
$\{p \wedge i < iv\}$
$i := i + 1$

$$
\begin{array}{ll}
\square & b[j] < c[k] \rightarrow \{p \wedge b[j] < c[k]\} \\
 & \qquad\qquad\quad \{p \wedge j < jv\} \\
 & \qquad\qquad\quad j := j + 1 \\
\square & c[k] < a[i] \rightarrow \{p \wedge c[k] < a[i]\} \\
 & \qquad\qquad\quad \{p \wedge k < kv\} \\
 & \qquad\qquad\quad k := k + 1 \\
\multicolumn{2}{l}{\text{od}} \\
\multicolumn{2}{l}{\{p \wedge \neg(a[i] < b[j]) \wedge \neg(b[j] < c[k]) \wedge \neg(c[k] < a[i])\}} \\
\multicolumn{2}{l}{\{q\}}
\end{array}
$$

In developing this program, the crucial step consisted of the choice of the guards B_1, B_2 and B_3. Accidentally, the choice made turned out to be sufficient to ensure that upon loop termination the postcondition q holds. Observe that the final program admits nondeterminism.

The programs we developed here were very simple. However, they exemplify the approach. Its essence thus consists of relying on a number of useful heuristics together with the idea of using proof as a guideline in the design process. This approach has been successfully applied to derive some larger and highly nontrivial programs. The idea of developing the program together with its proof turns out to be a powerful method which simplifies both tasks.

5 Disjoint Parallel Programs

In this chapter we begin to study *concurrent programs*. Whereas in a sequential program only one statement is executed at each moment of time, in a concurrent program several components can be active at the same time. Clearly, one reason for the interest in such programs is the desire for higher execution speed: each component of a concurrent program can be executed on an individual processor. But there are also other reasons: concurrency allows us to express explicitly when a program achieves its specification independently of the execution order of its subprograms or independently of how many processors are assigned to it. Moreover, concurrency is a most natural concept when modelling a system consisting of several independent components.

Usually, the components of a concurrent program have to exchange some information in order to achieve their common goal. This exchange is known as *communication*. Depending on the mode of communication, we distinguish between two types of concurrent programs, viz. *parallel programs* and *distributed programs*. The former may communicate only by means of shared variables whereas the latter communicate instead by explicit message passing. However, to simplify matters, we first study concurrent program without any communication between their components at all, viz. *disjoint parallel programs*, originally defined in Hoare [1972]. Parallel programs and distributed programs are just two different extensions of disjoint

parallel programs.

5.1 Syntax

Two **while**-programs S_1 and S_2 are called *disjoint* if none of them can change the variables accessed by the other one, i.e. if

$$change(S_1) \cap var(S_2) = var(S_1) \cap change(S_2) = \emptyset.$$

For example, the programs

$$x := z \text{ and } y := z$$

are disjoint because $change(x := z) = \{x\}$, $var(y := z) = \{y, z\}$ and $var(x := z) = \{x, z\}, change(y := z) = \{y\}$.

Disjoint parallel programs are generated by the same clauses as those defining **while**-programs in Chapter 3 together with the following clause for *disjoint parallel composition*:

$$S ::= [S_1 \| \ldots \| S_n]$$

where for $n \geq 1$ the components $S_1, \ldots, S_n$ are pairwise disjoint **while**-programs. Thus we do not allow nested parallelism, but we allow parallelism to occur within sequential composition, conditional statements and **while**-loops.

Intuitively, a disjoint parallel program of the form $S \equiv [S_1 \| \ldots \| S_n]$ terminates if and only if all of its components $S_1, \ldots, S_n$ terminate; the final state is then the union of the final states of $S_1, \ldots, S_n$.

5.2 Proof Theory

The following proof rule for disjoint parallel programs was proposed in Hoare [1972]. It links parallel composition of programs with logical conjunction of the corresponding pre- and postconditions and it sets the basic pattern for the more complicated proof rules needed to deal with shared variables and synchronization in Chapters 6 and 7.

RULE 12: DISJOINT PARALLELISM

$$\frac{\{p_i\}\ S_i\ \{q_i\}, i = 1, \ldots, n}{\{\wedge_{i=1}^n\ p_i\}\ [S_1 \| \ldots \| S_n]\ \{\wedge_{i=1}^n\ q_i\}}$$

where $S_1, \ldots, S_n$ are pairwise disjoint **while**-programs and $free(p_i, q_i) \cap change(S_j) = \emptyset$ for $i \neq j$.

The premises of this rule are to be proven with the proof systems PD or TD for deterministic programs. Depending on whether we choose PD or TD, the conclusion of the rule holds in the sense of partial or total correctness, respectively. Requiring disjointness of the pre- and postconditions and the component programs is necessary. Without it we could for example derive from the true formulas

$$\{y = 1\}\ x := 0\ \{y = 1\} \text{ and } \{\mathbf{true}\}\ y := 0\ \{\mathbf{true}\}$$

the conclusion

$$\{y = 1\}\ [x := 0 \| y := 0]\ \{y = 1\},$$

which is of course wrong.

Rule 12 alone is not sufficient for proving the correctness of disjoint parallel programs. The problem is that in correctness proofs we sometimes have to use properties of the program execution that cannot be expressed in terms of the existing program variables. The solution to this problem is to extend the program by *auxiliary variables*. These variables should neither influence the control flow nor the data flow of the program, but record only some additional information about the program execution. Once we have proven the desired correctness formula about the extended program, we may delete the auxiliary variables again and thus obtain a correctness formula about the original program. The following definition identifies sets of auxiliary variables in an extended program.

Definition 5.1 Let A be a set of simple variables in a program S. We call A a *set of auxiliary variables* of S if the variables in A occur in S only in assignments of the form $z := t$ with $z \in A$. □

Since auxiliary variables do not appear in Boolean expressions, they cannot influence the control flow in S, and since they are not used in assignments to variables outside of A, auxiliary variables cannot influence the data flow in S. As an example, consider the program

$$S \equiv z := x;\ [x := x + 1 \| y := y + 1].$$

Then

$$\emptyset, \{y\}, \{z\}, \{x, z\}, \{y, z\}, \{x, y, z\}$$

are all sets of auxiliary variables of S.

The following proof rule was first introduced in Owicki and Gries [1976].

RULE 13: AUXILIARY VARIABLES

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S_0\ \{q\}}$$

where for some set of auxiliary variables A of S with $free(q) \cap A = \emptyset$, the program S_0 results from S by deleting all assignments to the variables in A.

Like Rule 12, this rule is appropriate for proofs of both partial and total correctness. Let us denote by PP the proof system for partial correctness of disjoint parallel programs consisting of the group of axioms and rules 1–6, 12 and 13, and by TP the proof system for total correctness consisting of the group of axioms and rules 1–5, 7, 12 and 13.

Example 5.2 Let us apply the above proof rules to establish the following simple correctness formula:

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

By using a fresh variable z which records the initial values of x or y, respectively, we first derive the correctness formulas

$$\{x = z\}\ x := x + 1\ \{x = z + 1\}$$

and

$$\{y = z\}\ y := y + 1\ \{y = z + 1\}.$$

Now Rule 12 yields

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

Since the postcondition implies $x = y$, the consequence rule yields

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

Note that the consequence rule does not allow us to replace the precondition by $x = y$ because the implication

$$x = y \rightarrow x = z \wedge y = z$$

is false. Instead we consider the following correctness formula

$$\{x = y\}\ z := x\ \{x = z \wedge y = z\}$$

which can be easily established.

By the composition rule, we obtain

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

Since $\{z\}$ is a set of auxiliary variables of the above program, Rule 13 finally yields the desired result:

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

Observe that the semicolon ";" after $z := x$ is deleted, as well. □

Proof outlines for partial and total correctness of parallel programs are generated in a straightforward manner by the rules given for **while**-programs together with the following rule:

$$\frac{\{p_i\}\ S_i^*\ \{q_i\}, i = 1, \ldots, n}{\{\wedge_{i=1}^n\ p_i\}\ [\{p_1\}\ S_1^*\ \{q_1\} \| \ldots \| \{p_n\}\ S_n^*\ \{q_n\}]\ \{\wedge_{i=1}^n\ q_i\}}$$

For instance, the following proof outline summarizes the steps in Example 5.2:

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

The fact that z is used as an auxiliary variable is not visible from this proof outline; it has to be stated separately.

5.3 Verification: Find Positive Element

We study here a problem treated in Owicki and Gries [1976], Consider an integer array a and a constant $N \geq 1$. The task is to write a program S that finds the smallest index $k \in \{1, \ldots, N\}$ with

$$a[k] > 0$$

if such an element of a exists, otherwise the dummy value $k = N + 1$ should be returned.

Formally, the program S should satisfy the input-output specification

$$\{\mathbf{true}\}\ S\ \{k \leq N+1 \wedge \forall l(0 < l < k \rightarrow a[l] \leq 0) \wedge (k \leq N \rightarrow a[k] > 0)\} \quad (1)$$

in the sense of total correctness. Clearly, we require $a \notin \mathit{change}(S)$.

To speed up the computation, S is split into two components which are executed in parallel: the first component S_1 searches for an odd index k and the second component S_2 for an even one. The component S_1 uses a variable i for the (odd) index currently being checked and a variable *oddtop* to mark the end of the search:

$S_1 \equiv i := 1;\ \mathit{oddtop} := N + 1;$
while $i < \mathit{oddtop}$ **do**
if $a[i] > 0$ **then** $\mathit{oddtop} := i$

$$\textbf{else}\ \ i := i + 2\ \textbf{fi}$$

od.

The component S_2 uses variables j and *eventop* for analogous purposes:

$S_2 \equiv j := 2;\ \mathit{eventop} := N + 1;$
while $j < \mathit{eventop}$ **do**
if $a[j] > 0$ **then** $\mathit{eventop} := j$
else $j := j + 2$ **fi**
od.

The parallel program S is then given by

$$S \equiv [S_1 \| S_2];\ k := \mathit{min}(\mathit{oddtop}, \mathit{eventop}).$$

This is a version of the program *Findpos* studied in Owicki and Gries [1976] where the loop conditions have been simplified to achieve disjoint parallelism. For the original program *Findpos* see Chapter 6.

To prove that S satisfies its input-output specification (1), we first deal with its components. The first component S_1 searching for an odd index stores its result in the variable *oddtop*. Thus it should satisfy

$$\{\textbf{true}\}\ S_1\ \{q_1\} \tag{2}$$

in the sense of total correctness where q_1 is the following adaptation of the postcondition of (1):

$$\begin{aligned} q_1 \equiv\ & \mathit{oddtop} \leq N + 1 \\ \wedge\ & \forall l(\mathit{odd}(l) \wedge 0 < l < \mathit{oddtop} \rightarrow a[l] \leq 0) \\ \wedge\ & (\mathit{oddtop} \leq N \rightarrow a[\mathit{oddtop}] > 0). \end{aligned}$$

Symmetrically, the second component S_2 should satisfy

$$\{\textbf{true}\}\ S_2\ \{q_2\} \tag{3}$$

where

$$\begin{aligned} q_2 \equiv\ & \mathit{eventop} \leq N + 1 \\ \wedge\ & \forall l(\mathit{even}(l) \wedge 0 < l < \mathit{eventop} \rightarrow a[l] \leq 0) \\ \wedge\ & (\mathit{eventop} \leq N \rightarrow a[\mathit{eventop}] > 0). \end{aligned}$$

The notation $\mathit{odd}(l)$ and $\mathit{even}(l)$ expresses that l is odd or even, respectively.

We prove (2) and (3) using the system TD for total correctness of deterministic programs (Chapter 3). We start with (2). As usual, the main task is to find an appropriate invariant p_1 and a bound function t_1 for the loop in S_1.

As a loop invariant p_1 we choose a slight generalization of the postcondition q_1 which takes into account the loop variable i of S_1:

$$\begin{aligned} p_1 \quad \equiv \quad & \mathit{oddtop} \leq N+1 \wedge \mathit{odd}(i) \wedge i \leq \mathit{oddtop}+1 \\ \wedge \quad & \forall l(\mathit{odd}(l) \wedge 0 < l < i \rightarrow a[l] \leq 0) \\ \wedge \quad & (\mathit{oddtop} \leq N \rightarrow a[\mathit{oddtop}] > 0). \end{aligned}$$

As a bound function t_1, we simply choose

$$t_1 \equiv \mathit{oddtop} + 1 - i.$$

Note that the invariant p_1 ensures that $t_1 \geq 0$ holds.

We verify our choices by exhibiting a proof outline for the total correctness of S_1:

$\{\mathbf{true}\}$
$i := 1;$
$\{i = 1\}$
$\mathit{oddtop} := N + 1;$
$\{i = 1 \wedge \mathit{oddtop} = N + 1\}$
$\{\mathbf{inv} : p_1\}\{\mathbf{bd} : t_1\}$
while $i < \mathit{oddtop}$ **do**
  $\{p_1 \wedge i < \mathit{oddtop}\}$
  if $a[i] > 0$ **then** $\{p_1 \wedge i < \mathit{oddtop} \wedge a[i] > 0\}$
    $\{\quad i \leq N+1 \wedge \mathit{odd}(i) \wedge i \leq i+1$
    $\wedge\ \forall l(\mathit{odd}(l) \wedge 0 < l < i \rightarrow a[l] \leq 0)$
    $\wedge\ (i \leq N \rightarrow a[i] > 0)\}$
    $\mathit{oddtop} := i$
    $\{p_1\}$
  else $\{p_1 \wedge i < \mathit{oddtop} \wedge a[i] \leq 0\}$
    $\{\quad \mathit{oddtop} \leq N+1 \wedge \mathit{odd}(i+2)$
    $\wedge\ i+2 \leq \mathit{oddtop}+1$
    $\wedge\ \forall l(\mathit{odd}(l) \wedge 0 < l \leq i \rightarrow a[l] \leq 0)$
    $\wedge\ (\mathit{oddtop} \leq N \rightarrow a[\mathit{oddtop}] > 0)\}$
    $i := i + 2$
    $\{p_1\}$
  fi
  $\{p_1\}$
od
$\{p_1 \wedge \mathit{oddtop} \leq i\}$
$\{q_1\}.$

It is easy to see that in this outline all pairs of subsequent assertions form valid implications as required by the consequence rule. Also, note that both assignments within the loop decrease the bound function t_1 on the account of their respective preconditions.

For the second component S_2 we choose of course a symmetric invariant p_2 and bound function t_2:

$$\begin{aligned} p_2 \quad &\equiv \quad eventop \leq N+1 \wedge even(j) \wedge j \leq eventop+1 \\ &\wedge \quad \forall l(even(l) \wedge 0 < l < j \rightarrow a[l] \leq 0) \\ &\wedge \quad (eventop \leq N \rightarrow a[eventop] > 0), \\ t_2 \quad &\equiv \quad eventop + 1 - j. \end{aligned}$$

The verification of (3) with p_2 and t_2 is symmetric to (2) and is omitted.

We can now apply the rule of disjoint parallelism to (2) and (3) because the corresponding disjointness conditions are satisfied. We obtain

$$\{\mathbf{true}\}\ [S_1 \| S_2]\ \{q_1 \wedge q_2\}. \tag{4}$$

To complete the correctness proof, we look at the following proof outline

$$\begin{aligned} &\{q_1 \wedge q_2\} \\ &\{\quad min(oddtop, eventop) \leq N+1 \\ &\quad \wedge \forall l(0 < l < min(oddtop, eventop) \rightarrow a[l] \leq 0) \\ &\quad \wedge (min(oddtop, eventop) \leq N \rightarrow a[min(oddtop, eventop)] > 0)\} \\ &k := min(oddtop, eventop) \\ &\{k \leq N+1 \wedge \forall l(0 < l < k \rightarrow a[l] \leq 0) \\ &\qquad\qquad \wedge (k \leq N \rightarrow a[k] > 0)\}. \end{aligned} \tag{5}$$

Combining (4) and (5) by the composition rule yields the desired formula (1) about S.

6 Parallel Programs with Shared Variables

Disjoint parallelism is a rather restricted form of concurrency. In applications, concurrently operating components often share resources, e.g. a common data base, a line printer or a data bus. Sharing is necessary when resources are too costly to have one copy for each component as in the case of a large data base. Sharing is also useful to establish communication between different components as in the case of a data bus. This form of concurrency can be modelled by means of parallel programs with *shared variables*, i.e. variables that can be changed and read by several components.

As we shall see, proving the correctness of such programs is much more demanding than in the case of disjoint parallelism. The problem is while executing them different components can *interfere* with each other by changing the shared variables. To restrict the points of interference, we use so-called *atomic regions* whose execution cannot be interrupted by other components.

6.1 Syntax

Shared variables are introduced by dropping the disjointness requirement for parallel composition. Atomic regions may appear inside a parallel composition. Syntactically, these are statements enclosed in angle brackets $\langle$ and $\rangle$.

Thus we first define *component programs* as programs generated by the same clauses as those defining **while**-programs in Chapter 3 together with the following clause for atomic regions:

$$S ::= \langle S_0 \rangle$$

where S_0 is loop free and does not contain further brackets $\langle$ and $\rangle$. Now, *parallel programs* are generated by the same clauses as those defining **while**-programs together with the following clause for parallel composition:

$$S ::= [S_1 \| \ldots \| S_n]$$

where $S_1, \ldots, S_n$ are component programs ($n \geq 1$). Again, we do not allow nested parallelism, but we allow parallelism within sequential composition, conditional statements and **while**-loops.

Intuitively, an execution of $[S_1 \| \ldots \| S_n]$ is obtained by interleaving the atomic, i.e. non-interruptible steps in the executions of the components $S_1, \ldots, S_n$. By definition,

- Boolean expressions,
- assignments and skip, and
- atomic regions

are all evaluated or executed as atomic steps. An atomic region $\langle S_0 \rangle$ is executed by executing the program S_0. Since S_0 is required to be loop free, atomic steps are certain to terminate. An interleaved execution of $[S_1 \| \ldots \| S_n]$ terminates if and only if the individual execution of each component terminates.

For convenience, we shall identify

$$\langle A \rangle \equiv A$$

if A is an assignment or skip. By a *normal* subprogram of a program S we mean a subprogram of S not occurring within any atomic region of S. For example, the assignment $x := 0$, the atomic region $\langle x := x+2;\ z := 1 \rangle$ and the program $x := 0;\ \langle x := x+2;\ z := 1 \rangle$ are the only normal subprograms of $x := 0;\ \langle x := x+2;\ z := 1 \rangle$.

6.2 Proof Theory

It is very easy to give a proof rule for atomic regions because atomicity has no influence on the input-output behaviour of individual component programs:

RULE 14: ATOMIC REGION

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \langle S\rangle\ \{q\}}$$

where S is loop free.

This rule is appropriate for both partial and total correctness.

Proof outlines for partial and total correctness of component programs are generated by the rules given for **while**-programs plus the following one:

$$\frac{\{p\}\ S^*\ \{q\}}{\{p\}\ \langle S^*\rangle\ \{q\}}$$

where as usual S^* stands for an annotated version of S.

When defining proof outlines for total correctness of component programs, we have to modify the rule for loops by taking into account the atomic regions. To this end we include atomic regions in the definition of a path, that is we additionally stipulate the following clause in Definition 3.7:

- $path(\langle S\rangle) = \{\langle S\rangle\}$.

Moreover, we now allow T in rule (v') given in Definition 3.8 to vary over *normal* assignments *and* atomic regions.

For component programs S the definition of a *standard* proof outline $\{p\}\ S^*\ \{q\}$ is as follows: within S^* every *normal* subprogram T is preceded by exactly one assertion, called $pre(T)$, and there are no further assertions within S^*. In particular, there are no assertions within atomic regions.

Atomicity matters only in the context of a parallel composition with shared variables. In fact, the correctness formulas of a parallel program $[S_1 \| \ldots \| S_n]$ cannot now be determined any more from the correctness formulas of its components $S_1, \ldots, S_n$, but only from a detailed analysis of the atomic steps in the executions of $S_1, \ldots, S_n$.

Example 6.1 As an illustration of these difficulties let us look at the following three programs:

$$\begin{aligned} S_1 &\equiv x := x + 2, \\ S_2 &\equiv \langle x := x + 1;\ x := x + 1\rangle, \\ S_3 &\equiv x := x + 1;\ x := x + 1. \end{aligned}$$

Considered in isolation, their input-output behaviours are identical, i.e. for all assertions p and q and all $i, j \in \{1, 2, 3\}$ the correctness formula

$$\{p\}\ S_i\ \{q\}$$

is true in the sense of partial or total correctness iff

$$\{p\}\ S_j\ \{q\}$$

is true in the same sense.

However, with our explanation of the interleaved execution of parallel programs in mind, it is clear that

$$\{\mathbf{true}\}\ [x := 0 \| S_1]\ \{x = 0 \vee x = 2\}$$

and

$$\{\mathbf{true}\}\ [x := 0 \| S_2]\ \{x = 0 \vee x = 2\}$$

are true in the sense of both partial and total correctness whereas

$$\{\mathbf{true}\}\ [x := 0 \| S_3]\ \{x = 0 \vee x = 2\}$$

is false in both senses because the final value of x might be 1. This value is generated if the assignment $x := 0$ "interferes" with the execution of S_2, i.e. if it is executed in between the two assignments of S_2. □

To reason about the atomic steps taken in the components of a parallel program, we use standard proof outlines for the components instead of correctness formulas. A standard proof outline provides just the right level of detail because every possible atomic step of the component is preceded by exactly one assertion. Based on the assertions and bound functions in standard proof outlines, we can now introduce the important notion of *interference freedom* due to Owicki and Gries [1976].

Definition 6.2

(1) Let S be a component program. Consider a standard proof outline $\{p\}\ S^*\ \{q\}$ for total correctness and a statement R with the precondition $pre(R)$. We say that R *does not interfere with* $\{p\}\ S^*\ \{q\}$ if the following two conditions hold:

(i) for all assertions r in $\{p\}\ S^*\ \{q\}$ the correctness formula

$$\{r \wedge pre(R)\}\ R\ \{r\}$$

holds in the sense of total correctness,

(ii) for all bound functions t in $\{p\}\ S^*\ \{q\}$ the correctness formula

$$\{t = z \wedge pre(R)\}\ R\ \{t \leq z\}$$

holds in the sense of total correctness where z is some fresh variable not occuring in R, t and $pre(R)$.

(2) Let $[S_1 \| \ldots \| S_n]$ be a parallel program. Standard proof outlines $\{p_i\}\ S_i^*\ \{q_i\}\ , i = 1, \ldots, n$, for total correctness are called *interference free* if no normal assignment or atomic region of a component S_i interferes with the proof outline $\{p_j\}\ S_j^*\ \{q_j\}$ of another component $S_j, i \neq j$.

□

Thus interference freedom means that the execution of atomic steps of one component program neither falsifies the assertions (condition (i)) nor increases the bound function (condition (ii)) in the proof outline of any other component program.

Interference freedom of proof outlines for partial correctness is defined similarly, but with condition (ii) deleted.

With these preparations we can state the following conjunction rule for general parallel composition.

RULE 15: PARALLELISM WITH SHARED VARIABLES

$$\frac{\text{The standard proof outlines } \{p_i\}\ S_i^*\ \{q_i\},\ i = 1, \ldots, n, \text{ are interference free}}{\{\wedge_{i=1}^n\ p_i\}\ [S_1 \| \ldots \| S_n]\ \{\wedge_{i=1}^n\ q_i\}}$$

The correctness formula in the conclusion is true in the sense of partial or total correctness depending on whether proof outlines for partial or total correctness are used in the premises. Let us call PSV the proof system for *p*artial correctness of parallel programs with *s*hared *v*ariables consisting of the group of axioms and rules 1–6 and 13–15, and TSV the corresponding proof system for total correctness consisting of the group of axioms and rules 1–5, 7, 13–15. Proof outlines for parallel programs are defined in a straightforward manner (cf. Chapter 5).

The test of interference freedom makes correctness proofs for parallel programs more difficult than for sequential programs. For example, in the case of two component programs of length l_1 and l_2 proving interference freedom requires proving $l_1 \times l_2$ additional correctness formulas. In practice, however, most of these fomulas are trivially satisfied because they check an assignment or atomic region R against an assertion or bound function which does not contain the variables changed by R.

Example 6.3 We prove the correctness formula

$$\{\mathbf{true}\}\ [x := 0 \| x := x + 2]\ \{even(x)\} \tag{1}$$

in the system PSV. Since the program is loop free, this will be also a proof in TSV. To this purpose it is sufficient to consider the following correctness formulas which obviously hold:

$$\{\mathbf{true}\}\ x := 0\ \{even(x)\}$$

and

$$\{\mathbf{true}\}\ x := x + 2\ \{\mathbf{true}\}.$$

To prove interference freedom we need to prove 4 correctness formulas. Three out of them trivially hold and the fourth, $\{even(x)\}\ x := x+2\ \{even(x)\}$ clearly holds, as well. By Rule 15 we now get (1) as desired. □

Example 6.4 We now prove the correctness formula

$$\{\mathbf{true}\}\ [x := 0 \| x := x + 2]\ \{x = 0 \vee x = 2\} \tag{2}$$

in the system PSV. The proof makes use of an auxiliary Boolean variable "*done*" indicating whether the assignment $x := x + 2$ has been executed. This leads us to consider the correctness formula

$$\begin{array}{l} \{\mathbf{true}\} \\ done := \mathbf{false}; \\ [x := 0 \| \langle x := x + 2;\ done := \mathbf{true}\rangle] \\ \{x = 0 \vee x = 2\}. \end{array} \tag{3}$$

Since $\{done\}$ is indeed a set of auxiliary variables of the extended program, the rule of auxiliary variables (Rule 13) allows us to deduce (2) whenever (3) has been proved.

To prove (3), we consider the following standard proof outlines for the components of the parallel composition:

$$\{\mathbf{true}\}\ x := 0\ \{(x = 0 \vee x = 2) \wedge (\neg done \rightarrow x = 0)\} \tag{4}$$

and

$$\{\neg done\}\ \langle x := x + 2;\ done := \mathbf{true}\rangle\ \{\mathbf{true}\}. \tag{5}$$

Note that Rule 14 is used in the proof of (5).

It is straightforward to check that (4) and (5) are interference free. To this purpose 4 correctness formulas need to be verified. For example, the proof that the atomic region in (5) does not interfere with the postcondition

of (4) is as follows:

$$\begin{array}{l}\{(x = 0 \vee x = 2) \wedge (\neg done \rightarrow x = 0) \wedge \neg done\} \\ \{x = 0\} \\ \langle x := x + 2;\ done := \mathbf{true}\rangle \\ \{x = 2 \wedge done\} \\ \{(x = 0 \vee x = 2) \wedge (\neg done \rightarrow x = 0)\}.\end{array}$$

The remaining three cases are in fact trivial. Rule 15 applied to (4) and (5), and the consequence rule now yield

$$\begin{array}{l}\{\neg done\} \\ [x := 0 \| \langle x := x + 2;\ done := \mathbf{true}\rangle] \\ \{x = 0 \vee x = 2\}.\end{array} \tag{6}$$

On the other hand, the correctness formula

$$\{\mathbf{true}\}\ done := \mathbf{false}\ \{\neg done\} \tag{7}$$

obviously holds. Thus, applying the composition rule to (6) and (7) yields (3) as desired. □

The last correctness proof is more complicated than expected. Suprisingly, it cannot be simplified because it can be shown that any proof of (2) needs an auxiliary variable. This poses the question: how do we find appropriate auxiliary variables? Is there perhaps a systematic way of introducing them? The answer if positive. Following the lines of Lamport [1977], one can show that it is sufficient to introduce a separate *program counter* for each component of a parallel program. A program counter is an auxiliary variable which has a different value in front of every substatement in a component. It thus mirrors exactly the control flow in the component. In most applications, however, it suffices to have only partial information about the control flow. This can be represented by a few suitable auxiliary variables such as the variable "*done*" above.

6.3 Verification: Find Positive Element Quicker

In Section 5.3, we studied the problem of finding a positive element in an array a. More precisely, the problem was to find a program S with $a \notin change(S)$ which satisfies the total correctness formula

$$\{\mathbf{true}\}\ S\ \{k \leq N+1 \wedge \forall l(0 < l < k \rightarrow a[l] \leq 0) \wedge (k \leq N \rightarrow a[k] > 0)\}. \tag{8}$$

Here we consider a more sophisticated program S. As before it consists of two components S_1 and S_2 activated in parallel, such that S_1 searches for an odd index k of a positive element and S_2 searches for an even one.

However, now S_1 should stop searching once S_2 has found a positive element and vice versa for S_2. Thus some communication has to take place between S_1 and S_2. This is achieved by making *oddtop* and *eventop* shared variables of S_1 and S_2 by refining the loop conditions of S_1 and S_2 into

$$i < min\{oddtop, eventop\} \text{ and } j < min\{oddtop, eventop\},$$

respectively. Additionally, the initialization of *oddtop* and *eventop* have to be moved outside the parallel composition. Thus the program S is now of the form

$$\begin{aligned} S \quad \equiv \quad & oddtop := N+1;\ eventop := N+1; \\ & [S_1 \| S_2]; \\ & k := min(oddtop, eventop) \end{aligned}$$

where

$$\begin{aligned} S_1 \equiv\ & i := 1; \\ & \textbf{while } i < min(oddtop, eventop) \textbf{ do} \\ & \quad \textbf{if } a[i] > 0 \textbf{ then } oddtop := i \\ & \qquad\qquad \textbf{else } \ i := i+2 \textbf{ fi} \\ & \textbf{od} \end{aligned}$$

and

$$\begin{aligned} S_2 \equiv\ & j := 2; \\ & \textbf{while } j < min(oddtop, eventop) \textbf{ do} \\ & \quad \textbf{if } a[j] > 0 \textbf{ then } eventop := j \\ & \qquad\qquad \textbf{else } \ j := j+2 \textbf{ fi} \\ & \textbf{od.} \end{aligned}$$

The program *Findpos* studied in Owicki and Gries [1976] is like S, but with the initializations of the variables i, j outside of the parallel composition.

To prove (8) in the system TSV, we first construct appropriate proof outlines for S_1 and S_2. Let p_1, p_2 and t_1, t_2 be the invariants and bound functions introduced in Section 5.3. Then we consider the following standard proof outlines for total correctness. For S_1

$$\begin{aligned} & \{oddtop = N+1\}; \\ & \{\textbf{inv} : p_1\}\{\textbf{bd} : t_1\} \\ & \textbf{while } i < min(oddtop, eventop) \textbf{ do} \\ & \quad \{p_1 \wedge i < oddtop\} \\ & \quad \textbf{if } a[i] > 0 \textbf{ then } \{p_1 \wedge i < oddtop \wedge a[i] > 0\} \\ & \qquad\qquad\qquad oddtop := i \\ & \qquad\qquad \textbf{else } \ \{p_1 \wedge i < oddtop \wedge a[i] \leq 0\} \\ & \qquad\qquad\qquad i := i+2 \\ & \quad \textbf{fi} \end{aligned}$$

od
$\{p_1 \wedge i \geq min(oddtop, eventop)\}$

and there is a symmetric proof outline for S_2. Note that, except for the new postconditions which are the consequence of the new loop conditions, all other assertions are taken from the corresponding proof outlines in Section 5.3.

To apply Rule 15 for the parallel composition of S_1 and S_2, we have to show interference freedom of the two proof outlines. This amounts to checking 42 correctness formulas! Fortunately, 40 of them are trivially satisfied because the variable changed by the assignment does not appear in the assertion or bound function under consideration. The only non-trivial cases deal with the interference-freedom of the postcondition of S_1 with the assignment to the variable *eventop* in S_2 and, symmetrically, of the postcondition of S_2 with the assignment to the variable *oddtop* in S_1.

We deal with the postcondition of S_1, viz.

$$p_1 \wedge i \geq min(oddtop, eventop),$$

and the assignment $eventop := j$. Since $pre(eventop := j)$ implies $j < eventop$, we have the following proof of interference freedom:

$\{p_1 \wedge i \geq min(oddtop, eventop) \wedge pre(eventop := N+1)\}$
$\{p_1 \wedge i \geq min(oddtop, eventop) \wedge j < eventop\}$
$\{p_1 \wedge i \geq min(oddtop, j)\}$
$eventop := j$
$\{p_1 \wedge i \geq min(oddtop, eventop)\}$.

An analogous argument takes care of the postcondition of S_2. This finishes the overall proof of interference freedom of the two proof outlines.

Now Rule 15 is applicable and yields

$\{oddtop = N+1 \wedge eventop = N+1\}$
$[S_1 \| S_2]$
$\{p_1 \wedge p_2 \wedge i \geq min(oddtop, eventop) \wedge j \geq min(oddtop, eventop)\}$.

By the assignment axiom and the consequence rule,

$\{\mathbf{true}\}$
$oddtop := N+1;\ eventop := N+1;$
$[S_1 \| S_2]$
$\{\quad min(oddtop, eventop) \leq N+1$
$\quad \wedge \forall l(0 < l < min(oddtop, eventop) \rightarrow a[l] = 0)$
$\quad \wedge\ (min(oddtop, eventop) \leq N \rightarrow a[min(oddtop, eventop)] > 0)\}$.

Hence the final assignment $k := min(oddtop, eventop)$ in S establishes the desired postcondition of (8).

7 Parallel Programs with Synchronization

For many applications we need parallel programs whose components can synchronize with each other, i.e. they wait or get *blocked* until the execution of the other components changes the shared variables into a more favourable state. We therefore now extend the program syntax by a synchronization construct, the **await**-statement introduced in Owicki and Gries [1976]. This construct enables a very flexible way of programming, but at the same time opens the door for subtle programming errors where the program execution ends in a *deadlock*. This is a situation where all non-terminated components of a parallel program have become blocked. Hence total correctness of parallel programs with synchronization will now also require a proof of deadlock freedom.

7.1 Syntax

Now **await**-statements may appear inside a parallel composition. Thus a *component program* is now a program generated by the same clauses as those defining **while**-programs in Chapter 3 together with the following clause:

$$S ::= \textbf{await } B \textbf{ then } S_0 \textbf{ end}$$

where S_0 is a loop free **while**-program. *Parallel programs* are then generated by the same clauses as those defining **while**-programs together with the following clause for parallel composition:

$$S ::= [S_1 \| \ldots \| S_n]$$

where $S_1, \ldots, S_n$ are component programs ($n \geq 1$). Thus as before, we do not allow nested parallelism, but we do allow parallelism within sequential composition, conditional statements and **while**-loops.

To explain the meaning of an **await**-statement, first note that they can occur only within a parallel composition. Consider now an interleaved execution of a parallel program where one component is about to execute a statement **await** B **then** S_0 **end**. If B evaluates to true, then S_0 is executed as an atomic region whose activation cannot be interrupted by the other components. If B evaluates to false, the component gets *blocked* and the other components take over the execution. If during their execution B becomes true, the blocked component can resume its execution. Otherwise, it remains blocked forever.

Thus **await**-statements model *conditional atomic regions*. If $B \equiv \textbf{true}$, we obtain the same effect as with an unconditionally atomic region of the previous chapter. Hence we identify

$$\textbf{await true then } S_0 \textbf{ end} \equiv \langle S_0 \rangle.$$

For the extended syntax of this chapter, a subprogram of a program S is called *normal* if it does not occur within an **await**-statement of S.

7.2 Proof Theory

Partial Correctness

First we deal with partial correctness. For component programs, we use the proof rules of the system PD for **while**-programs plus the following simple rule given in Owicki and Gries [1976]:

RULE 16: CONDITIONAL ATOMIC REGION

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \textbf{await}\ B\ \textbf{then}\ S\ \textbf{end}\ \{q\}}$$

where S is loop free.

Note that with $B \equiv \textbf{true}$ we get Rule 14 for atomic regions as a special case.

Proof outlines for partial correctness of component programs are generated by the rules for **while**-programs together with the following one:

$$\frac{\{p \wedge B\}\ S^*\ \{q\}}{\{p\}\ \textbf{await}\ B\ \textbf{then}\ S^*\ \textbf{end}\ \{q\}}$$

where S^* stands for an annotated version of the loop free statement S. The definition of *standard* proof outlines is stated as in the previous chapter, but it refers now to the extended notion of a normal subprogram given in Section 7.1. Thus there are no assertions within **await**-statements.

Interference freedom refers now to **await**-statements instead of atomic regions. Thus standard proof outlines $\{p_i\}\ S_i^*\ \{q_i\}$, $i = 1, \ldots, n$, for partial correctness are called *interference free* if no normal assignment or **await**-statement of a component program S_i interferes (in the sense of the previous chapter) with the proof outline of another component program S_j, $i \neq j$.

For parallel composition we use Rule 15 of the previous chapter. However, since **await**-statements may now appear in the component programs, this rule refers now to the above notions of a standard proof outline and interference freedom. Hence the proof system for partial correctness of *p*arallel programs with *sy*nchronization, abbreviated PSY, consists of the group of axioms and rules 1–6, 13, 15 and 16.

Total Correctness

For total correctness things are more complicated. The reason is that in the presence of **await**-statements program termination not only requires

divergence freedom (absence of infinite computations), but also deadlock freedom (absence of infinite blocking). Deadlock freedom is a *global* property that can be proved only by examining all components of a parallel program together. Thus none of the components of a terminating program need to terminate when considered in isolation; each of them may get blocked. Of course, each component must be divergence free.

In order to deal with such subtleties, we introduce the notion of *weak total correctness* which combines partial correctness with divergence freedom. In other words, a correctness formula $\{p\}\ S\ \{q\}$ holds in the sense of weak total correctness if every execution of S starting in a state satisfying p is finite and either terminates in a state satisfying q or gets blocked.

To prove total correctness of a parallel program, we first prove weak total correctness of its components, then establish interference freedom and finally use an extra test for deadlock freedom that refers to all components together.

Proving weak total correctness of component programs is simple. We use all the proof rules of the system TD for **while**-programs and Rule 16 when dealing with **await**-statements. Note that Rule 16 permits only weak total correctness because the execution of **await** B **then** S **end**, when started in a state satisfying $p \wedge \neg B$, does not terminate. Instead it gets blocked (see Example 7.2). This blocking can only be resolved with the help of other components executed in parallel.

(Standard) proof outlines for weak total correctness of component programs are generated by the rules given for total correctness of **while**-programs together with the rule above which deals with **await**-statements. However, due to the presence of **await**-statements we also have to ensure that they decrease or leave unchanged the corresponding bound functions. This is resolved in an analogous way as for atomic regions in Chapter 6.

Standard proof outlines $\{p_i\}\ S_i^*\ \{q_i\}$, $i = 1, \ldots, n$, for weak total correctness are called *interference free* if no normal assignment or **await**-statement of a component program S_i interferes with the proof outline of another component program S_i, $i \neq j$.

We prove deadlock freedom of a parallel program by examining interference free standard proof outlines for weak total correctness of its component programs. We follow the strategy of Owicki and Gries [1976] and first enumerate all potential deadlock situations and then use certain combinations of the assertions from the proof outlines to show that these deadlock situations can never actually occur.

Definition 7.1 Consider a parallel program $S \equiv [S_1 \| \ldots \| S_n]$.

(1) A tuple $< R_1, \ldots, R_n >$ of statements is called a *potential deadlock of* S if the following holds:

(i) each R_i, $i = 1, \ldots, n$, is either an **await**-statement in the component S_i or the symbol E which stands for the empty statement

and represents termination of S_i,

(ii) at least one R_i, $i = 1, \ldots, n$, is an await-statement in S_i.

(2) Given interference free standard proof outlines $\{p_i\}\ S_i^*\ \{q_i\}$ for weak total correctness, $i = 1, \ldots, n$, we associate with every potential deadlock of S a corresponding tuple $< r_1, \ldots, r_n >$ of assertions by putting for $i = 1, \ldots, n$:

(i) $r_i \equiv pre(R_i) \wedge \neg B$ if $R_i \equiv$ **await** B **then** S **end**,

(ii) $r_i \equiv q_i$ if $R_i \equiv E$.

□

If we can show $\neg \wedge_{i=1}^n\ r_i$ for every such tuple $< r_1, \ldots, r_n >$ of assertions, none of the potential deadlocks can actually arise. This is how deadlock freedom is established in the second premise of the following proof rule for parallel composition.

RULE 17: PARALLELISM WITH DEADLOCK FREEDOM

(1) The standard proof outlines $\{p_i\}\ S_i^*\ \{q_i\}$ for weak total correctness are interference free, $i = 1, \ldots, n$.

(2) For every potential deadlock $< R_1, \ldots, R_n >$ of $[S_1 \| \ldots \| S_n]$ the corresponding tuple of assertions $< r_1, \ldots, r_n >$ satisfies $\neg \wedge_{i=1}^n\ r_i$.

$$\frac{}{\{\wedge_{i=1}^n\ p_i\}\ [S_1 \| \ldots \| S_n]\ \{\wedge_{i=1}^n\ q_i\}}$$

By TSY we denote the proof system consisting of the group of axioms and rules 1–5, 7, 13, 16 and 17. It stands for *t*otal correctness of parallel programs with *s*ynchronization. Proof outlines for parallel programs are defined in a straightforward manner (cf. Chapter 5).

The following example illustrates the use of Rule 17 and demonstrates that for the components of parallel programs we cannot prove in isolation more than weak total correctness.

Example 7.2 We wish to prove the correctness formula

$$\{x = 0\}\ [\textbf{await}\ x = 1\ \textbf{then}\ \textit{skip}\ \textbf{end} \| x := 1]\ \{x = 1\} \qquad (1)$$

in the system TSY. For the component programs we use the following interference free standard proof outlines for weak total correctness:

$$\{x = 0 \vee x = 1\}\ \textbf{await}\ x = 1\ \textbf{then}\ \textit{skip}\ \textbf{end}\ \{x = 1\} \qquad (2)$$

and

$$\{x = 0\}\ x := 1\ \{x = 1\}.$$

Formula (2) is proved using Rule 16; it is true only in the sense of weak total correctness because the execution of the **await**-statement gets blocked when started in a state satisfying $x = 0$.

Deadlock freedom is proved as follows. The only potential deadlock is

$$< \textbf{await } x = 1 \textbf{ then } \textit{skip} \textbf{ end}, E > . \tag{3}$$

The corresponding pair of assertions is

$$< (x = 0 \vee x = 1) \wedge x \neq 1, x = 1 >,$$

the conjunction of which is clearly false. Hence (3) cannot arise as an actual deadlock. Now Rule 17 is applicable and yields (1) as desired. □

7.3 Verification: The Producer Consumer Problem

A reoccurring task in the area of parallel programming is the coordination of producers and consumers. A producer generates a stream of $M \geq 1$ values for a consumer. We assume that the producer and consumer work in parallel and proceed at a variable but rougly equal pace.

The problem is to coordinate their work so that all values produced arrive at the consumer and that they arrive in the order of production. Moreover, the producer should not have to wait with the production of a new value if the consumer is momentarily slow with its consumption. Conversely, the consumer should not have to wait if the producer is momentarily slow with its production.

The general idea of solving this producer/consumer problem is to interpose a buffer between producer and consumer. Then the producer adds values to the buffer and the consumer removes values from the buffer. This way small variations in the pace of producers are not noticeable for the consumer and vice versa. However, since in reality the storage capacity of a buffer is limited, say to $N \geq 1$ values, we have to synchronize producer and consumer in such a way that the producer never attempts to add a value into the full buffer and that the consumer never attempts to remove a value from the empty buffer.

Following Owicki and Gries [1976] we express the producer/consumer problem as a parallel program with shared variables and **await**-statements. The producer and consumer are modelled as two components $PROD$ and $CONS$ of a parallel program. Production of a value is modelled as reading an integer value from a finite section

$$a[0 : M - 1]$$

of an array a of type **integer** $\rightarrow$ **integer** and consumption of a value as writing an integer value into a corresponding section

$$b[0 : M - 1]$$

of an array b of type **integer** $\rightarrow$ **integer**. The buffer is modelled as a section

$$buffer[0 : N - 1]$$

of a shared array $buffer$ of type **integer** $\rightarrow$ **integer**. M and N are integer constants $M, N \geq 1$. For a correct access of the buffer the components $PROD$ and $CONS$ share an integer variable in counting the number of values added to the buffer and an integer variable out counting the number of values removed from the buffer. Thus at each moment the buffer contains $in - out$ values; it is full if $in - out = N$ and it is empty if $in - out = 0$. Adding and removing values to and from the buffer is performed in a cyclic order

$$buffer[0], \ldots, buffer[N-1], buffer[0], \ldots, buffer[N-1], buffer[0], \ldots$$

Thus the expressions in *mod* N and out *mod* N determine the subscript of the buffer element where the next value is to be added or removed. This explains why we start numbering the buffer elements from 0 onwards.

With these preparations we can express the producer/consumer problem by the following parallel program:

$$S \equiv in := 0;\ out := 0;\ i := 0;\ j := 0;\ [PROD \| CONS]$$

where

$$\begin{aligned} PROD \equiv\ & \textbf{while } i < M \textbf{ do} \\ & \quad x := a[i]; \\ & \quad ADD(x); \\ & \quad i := i + 1 \\ & \textbf{od} \end{aligned}$$

and

$$\begin{aligned} CONS \equiv\ & \textbf{while } j < M \textbf{ do} \\ & \quad REM(y); \\ & \quad b[j] := y; \\ & \quad j := j + 1 \\ & \textbf{od}. \end{aligned}$$

Here i, j, x, y are integer variables and $ADD(x)$ and $REM(y)$ abbreviate the following synchronized statements for adding and removing values from the shared buffer:

$$ADD(x) \equiv \textbf{wait } in - out < N;$$
$$buffer[in \bmod N] := x;$$
$$in := in + 1$$

and

$$REM(y) \equiv \textbf{wait } in - out > 0;$$
$$y := buffer[out \bmod N];$$
$$out := out + 1$$

Here for a Boolean expression B the statement **wait** B abbreviates **await** B **then** *skip* **end**.

We claim that the following correctness formula holds in the sense of total correctness:

$$\{\textbf{true}\}\ S\ \{\forall k(0 \leq k < M \rightarrow a[k] = b[k])\}, \tag{4}$$

i.e. the program S is deadlock free and terminates with all values from $a[0 : M-1]$ copied in that order into $b[0 :, M-1]$. The verification of (4) follows closely the presentation in Owicki and Gries [1976].

First consider the component program $PROD$. As a loop invariant we take

$$p_1 \equiv \quad \forall k(out \leq k < in \rightarrow a[k] = buffer[k \bmod N]) \tag{5}$$
$$\wedge\ 0 \leq in - out \leq N \tag{6}$$
$$\wedge\ 0 \leq i \leq M \tag{7}$$
$$\wedge\ i = in \tag{8}$$

and as a bound function

$$t_1 \equiv M - i.$$

Further on, we introduce the following abbreviation for the conjunction of some the lines in p_1:

$$I \equiv (5) \wedge (6)$$

and

$$I_1 \equiv (5) \wedge (6) \wedge (7).$$

As a standard proof outline we consider

$\{\textbf{inv} : p_1\}\{\textbf{bd} : t_1\}$
while $i < M$ **do**
  $\{p_1 \wedge i < M\}$
  $x := a[i];$
  $\{p_1 \wedge i < M \wedge x = a[i]\}$
  wait $in - out < N;$

$\{p_1 \wedge i < M \wedge x = a[i] \wedge in - out < N\}$
$buffer[in \bmod N] := x;$
$\{p_1 \wedge i < M \wedge a[i] = buffer[in \bmod N] \wedge in - out < N\}$ (9)
$in := in + 1;$
$\{I_1 \wedge i + 1 = in \wedge i < M\}$ (10)
$i := i + 1$
od
$\{p_1 \wedge i = M\}.$

It is straightforward to see that this is indeed a proof outline for weak total correctness of $PROD$. In particular, note that (9) implies

$$\forall k(out \leq k < in + 1 \rightarrow a[k] = buffer[k \bmod N])$$

which justifies the conjunct (5) of the postcondition (10) of the assignment $in := in + 1$. Note also that the bound function t_1 clearly satisfies the conditions required by the definition of proof outline.

Now consider the component program $CONS$. As a loop invariant we take

$$\begin{aligned} p_2 \equiv \quad & I && (11)\\ \wedge \ & \forall k(0 \leq k < j \rightarrow a[k] = b[k]) && (12)\\ \wedge \ & 0 \leq j \leq M && (13)\\ \wedge \ & j = out, && (14) \end{aligned}$$

i.e. the I-part of p_1 reappears here, and as a bound function we take

$$t_2 \equiv M - j.$$

Let us abbreviate

$$I_2 \equiv (11) \wedge (12) \wedge (13)$$

and consider the following standard proof outline:

$\{\mathbf{inv} : p_2\}\{\mathbf{bd} : t_2\}$
while $j < M$ **do**
$\{p_2 \wedge j < M\}$
wait $in - out > 0$;
$\{p_2 \wedge j < M \wedge in - out > 0\}$
$y := buffer[out \bmod N];$
$\{p_2 \wedge j < M \wedge in - out > 0 \wedge y = a[j]\}$ (15)
$out := out + 1;$
$\{I_2 \wedge j + 1 = out \wedge j < M \wedge y = a[j]\}$
$b[j] := y;$
$\{I_2 \wedge j + 1 = out \wedge j < M \wedge a[j] = b[j]\}$

$$
\begin{array}{l}
\quad j := j + 1 \\
\textbf{od} \\
\{p_2 \wedge j = M\}
\end{array}
$$

It is easy to see that this is a correct proof outline for weak total correctness. In particular, note that the conjunct $y = a[j]$ in the assertion (15) is obtained as follows:

$$
\begin{array}{rcl}
y & = & buffer[out \bmod N] \\
& = & \{(5) \wedge in - out > 0\} \\
& & a[out] \\
& = & \{(14)\} \\
& & a[j].
\end{array}
$$

Also the bound function t_2 satisfies the conditions required by the definition of proof outline.

Let us now turn to the test of interference freedom of the two proof outlines. Naive calculations suggest that 80 correctness formulas have to be checked! However, most of these checks can be dealt with by a single argument, viz. that I-part of p_1 and p_2 is kept invariant in both proof outlines. In other words, all assignments T in the proof outlines for $PROD$ and $CONS$ satisfy

$$\{I \wedge pre(T)\}\ T\ \{I\}.$$

It thus remains to check the assertions outside the I-part against possible interference. Consider first the proof outline for $PROD$. Examine all conjuncts occurring in the assertions used in this proof outline. Among them, apart of I, only the conjunct $in - out < N$ contains a variable which is changed in the component $CONS$. But this change is done only by the assignment $out := out + 1$. Obviously, we have here interference freedom:

$$\{in - out < N\}\ out := out + 1\ \{in - out < N\}.$$

Now consider the proof outline for $CONS$. Examine all conjuncts occurring in the assertions used in this proof outline. Among them, apart of I, only the conjunct $in - out > 0$, contains a variable which is changed in the component $PROD$. But this change is done only be the assignment $in := in + 1$. Obviously, we have here again interference freedom:

$$\{in - out > 0\}\ in := in + 1\ \{in - out > 0\}.$$

Next, we show deadlock freedom. The potential deadlocks are

$< \textbf{wait } in - out < N, \textbf{wait } in - out > 0 >,$
$< \textbf{wait } in - out < N, E >,$
$< E, \textbf{wait } in - out > 0 >$

and logical consequences of the corresponding pairs of assertions from the above proof outlines are

$< in - out \geq N, in - out \leq 0 >,$
$< in < M \wedge in - out \geq N, out = M >,$
$< in = M, out < M \wedge in - out \leq 0 >.$

Since $N \geq 1$, the conjunction of the corresponding two assertions is in all three cases false. This proves deadlock freedom.

We can now apply Rule 17 for the parallel composition of $PRODS$ and $CONS$ and obtain:

$$\{p_1 \wedge p_2\}\ [PROD \| CONS]\ \{p_1 \wedge p_2 \wedge in = M \wedge j = M\}.$$

Since

$$\{\textbf{true}\}\ in := 0;\ out := 0;\ i := 0;\ j := 0\ \{p_1 \wedge p_2\}$$

and

$$p_1 \wedge p_2 \wedge i = M \wedge j = M \rightarrow \forall k(0 \leq k < M \rightarrow a[k] = b[k]),$$

we obtain the desired correctness formula (4) about S by straightforward application of the composition rule and the consequence rule.

8 Distributed Programs

Distributed programs are concurrent programs with disjoint components which communicate by explicit message passing. Many real systems can be modeled by distributed programs. As an example consider an airline reservation system consisting of a large number of terminals in many different travel agencies and a central data base for keeping the current status of all flights. The data base and the terminals can be modeled as the components of a distributed program. In this case communication will involve a two way connection between each terminal and the database.

There are two ways of organizing message passing. We consider here *synchronous communication* where the sender of a message can deliver it only when the receiver is ready to accept it at the same moment. An example is communication by telephone. Synchronous communication is also called *handshake* communication or *rendezvous*. Another possibility is *asynchronous communication* where the sender can always deliver its message. This stipulates an implicit buffer where messages are kept until the receiver

collects them. Communication by mail is an example. Asynchronous communication can be modeled by synchronous communication if the buffer is introduced as an explicit component of the distributed program.

As a syntax for distributed programs we consider a simple subset of the language CSP (standing for Communicating Sequential Processes) introduced in Hoare [1978]. CSP extends Dijkstra's guarded command language (studied in Chapter 4) by the introduction of disjoint parallel composition and input-output commands for synchronous communication. We will explain this now in detail.

8.1 Syntax

A (*sequential*) *process with name* P or simply a *process* P is a component

$$P :: S$$

where P is a name and S, called a *body* of P, is a statement of the form

$$S \equiv S_0;\ \mathbf{do}\ \Box_{j=1}^{m}\ g_j \to S_j\ \mathbf{od}$$

such that $m \geq 0, S_0, \ldots, S_m$ are nondeterministic programs as defined in Chapter 4, and $g_1, \ldots, g_m$ are generalized guards. The statement

$$\mathbf{do}\ \Box_{j=1}^{m}\ g_j \to S_j\ \mathbf{od}$$

is called the *main loop* of S. A *generalized guard* has the form

$$g \equiv B; \alpha$$

where B is a Boolean expression and α an input-output command or shorter an *i/o command*.

A main loop is exited when the Boolean part of each generalized guard of the loop evaluates to false.

There are two types of i/o commands: an *input command*, written as $P_j?u$, and an *output command*, written as $P_j!t$. The first, when used within a process P_i, expresses its request to process P_j to send a value which will be assigned to the simple or subscripted variable u. An output command is the action which makes it possible. When used in a process P_i it expresses its request to process P_j to receive the value of the expression t. *Both* requests are delayed until they can be performed together. In particular, the output command cannot be executed independently. The joint execution of two i/o commands, called a *communication*, is possible when they match.

Definition 8.1 We say that two i/o commands *match* when one is an input command, say $P_j?u$, and the other an output command, say $P_i!t$, such that

$P_j?u$ is contained in the process P_i and $P_i!t$ is contained in the process P_j, and the types of u and t agree. □

Two generalized guards match if their i/o commands match. They can be passed jointly when they match and their Boolean parts evaluate to true. Then the communication between the i/o commands takes place.

The effect of a communication between two matching i/o commands $\alpha_1 \equiv P_j?u$ and $\alpha_2 \equiv P_i!t$ is the assignment $u := t$. Formally, we define

$$Eff(\alpha_1, \alpha_2) \equiv Eff(\alpha_2, \alpha_1) \equiv u := t.$$

For a process $P :: S$ let $change(S)$ denote the set of all simple or array variables that appears in S on the left-hand side of an assignment or in an input command and let $var(S)$ denote the set of all simple or array variables appearing in S. Processes $P_1 :: S_1$ and $P_2 :: S_2$ are called *disjoint* if the following condition holds:

$$change(S_1) \cap var(S_2) = var(S_1) \cap change(S_2) = \emptyset.$$

Now, distributed programs are generated by the same clauses as those defining nondeterministic programs in Chapter 4 together with the following clause for parallel composition:

$$S ::= [P_1 :: S_1 \| \ldots \| P_n :: S_n]$$

where $P_1 :: S_1, \ldots, P_n :: S_n$ are disjoint processes with distinct names $P_1, \ldots, P_n$. We say that two processes $P_i :: S_i$ and $P_j :: S_j$ are connected by a *communication channel* if they contain a pair of matching generalized guards. When the bodies of P_i-s are clear from the context, we omit them and simply write $[P_1 \| \ldots \| P_n]$.

A distributed program terminates when all of its processes terminate. This means that distributed programs may fail to terminate because of divergence of a process or an abortion arising in one of the processes. However, they may also fail to terminate because of a deadlock. A *deadlock* arises here when not all processes have terminated, none of them has aborted and yet none of them can proceed. This will happen when all nonterminated processes will be in front of their main loops but no pair of their generalized guard matches.

We say that a distributed program S is *deadlock free relative to an assertion* p if no deadlock can arise in executions of S starting in a state satisfying p.

Example 8.2 Here and in the next example we assume a new basic type **character**. Thus we may use constants and variables ranging over it. The constants of type **character** which we shall use are the ASCII characters.

Here we wish to write a program

$$S \equiv [BUFFER||CONSOLE]$$

where the process *BUFFER* sends to the process *CONSOLE* a sequence of k ($k \geq 1$) characters. To this end, we use two array variables a, b of type **integer** $\rightarrow$ **character** and put

$$BUFFER :: i := 1;\ \textbf{do}\ i \neq k+1; CONSOLE!A[i] \rightarrow i := i+1\ \textbf{od}$$

and

$$CONSOLE :: j := 1;\ \textbf{do}\ j \neq k+1; BUFFER?B[j] \rightarrow j := j+1\ \textbf{od}.$$

Note that the above program is deterministic in the sense that only one computation is possible. It terminates after both *BUFFER* and *CONSOLE* execute their loops k times. □

Example 8.3 In the following program

$$S \equiv [BUFFER||FILTER||CONSOLE]$$

the process *BUFFER* sends to the process *CONSOLE* through the process *FILTER* a sequence of k ($k \geq 1$) characters ending with '$*$'. *FILTER* deletes all blanks in the sequence. It is assumed that '$*$' appears in the sequence only at its end. We have

BUFFER :: $i := 1;$
do $i \neq k+1; FILTER!A[i] \rightarrow i := i+1$ **od**,

FILTER :: $send := 1;\ rec := 1;\ b := \text{' '};$
do $b \neq \text{'}*\text{'}; BUFFER?b \rightarrow$
if $b = \text{' '} \rightarrow skip$
$\square b \neq \text{' '} \rightarrow B[rec] := b;$
$rec := rec + 1$
fi
$\square send \neq rec; CONSOLE!B[send] \rightarrow send := send + 1$
od,

CONSOLE :: $n := 1;\ c := \text{' '};$
do $c \neq \text{'}*\text{'}; FILTER?c \rightarrow C[n] := c; n := n+1$ **od**.

The process *FILTER* continues to receive characters from the process *BUFFER* until '$*$' is sent. It can also send to the process *CONSOLE* the nonblank characters received so far. The presence of two generalized guards in *FILTER* reflects its nondeterministic behavior and allows more than one computation of the program S.

BUFFER terminates once it has sent all k characters to *FILTER*. *FILTER* terminates when it has received the character '$*$' *and* has sent to *CONSOLE*

all characters it has received. Finally, *CONSOLE* terminates once it has received from *FILTER* the character '$*$'.

To better understand the nature of deadlock situations consider what would happen if the Boolean guard of *BUFFER* were changed to $i < k$. Then *BUFFER* would not send the last character of the sequence, that is '$*$'. Thus *FILTER* would not falsify its first Boolean guard and so would never exit the loop. *CONSOLE* would never exit its loop either and a deadlock would result once *CONSOLE* has received all nonblank characters from *FILTER*. □

8.2 Transformation into Nondeterministic Programs

Consider a parallel composition

$$S \equiv [P_1 :: S_1 \| \ldots \| P_n :: S_n]$$

of n disjoint processes where

$$S_i \equiv S_{i,0};\ \textbf{do}\ \Box_{j=1}^{m_i}\ B_{i,j}; \alpha_{i,j} \rightarrow S_{i,j}\ \textbf{od}$$

for $i = 1, \ldots, n$. As abbreviations we introduce

$$\Gamma = \{(i,j,k,\ell) \mid \alpha_{i,j} \text{ and } \alpha_{k,\ell} \text{ match and } i < k\}$$

and

$$\mathit{TERM} \equiv \wedge_{i=1}^{n}\ \wedge_{j=1}^{m_i}\ \neg B_{i,j}.$$

Observe that *TERM* holds upon termination of S.

We transform S into the following nondeterministic program $T(S)$:

$$\begin{aligned} T(S) \equiv\ & S_{1,0};\ \ldots;\ S_{n,0}; \\ & \textbf{do}\ \Box_{(i,j,k,\ell)\in\Gamma}\ B_{i,j} \wedge B_{k,\ell} \rightarrow \mathit{Eff}(\alpha_{i,j}, \alpha_{k,\ell});\ S_{i,j};\ S_{k,\ell}\ \textbf{od}; \\ & \textbf{if}\ \mathit{TERM} \rightarrow \mathit{skip}\ \textbf{fi} \end{aligned}$$

where the use of elements of Γ to "summate" all guards should be clear. Note that upon exit of the main loop of $T(S)$ the assertion

$$\mathit{BLOCK} \equiv \wedge_{(i,j,k,\ell)\in\Gamma}\ \neg(B_{i,j} \wedge B_{k,\ell})$$

holds. This formula holds also whenever deadlock is reached in S. The behaviour of the distributed program S is equivalent to the behaviour of the nondeterministic program $T(S)$ in the sense of partial correctness. We shall make use of this obervation in the next section.

8.3 Proof Theory

The proof theory of distributed programs is surprisingly simple. We follow here the approach of Apt [1986]. Adopt the notation of the previous section. Consider first partial correctness. We augment the proof system PN for partial correctness of nondeterministic programs by the following rule:

RULE 18: DISTRIBUTED PROGRAMS

$$\frac{\begin{array}{l}\{p\}\ S_{0,1};\ \ldots;\ S_{0,n}\ \{I\},\\ \{I \wedge B_{i,j} \wedge B_{k,\ell}\}\ Eff(\alpha_{i,j}, \alpha_{k,\ell});\ S_{i,j};\ S_{k,\ell}\ \{I\}\\ \quad \text{for all } (i,j,k,\ell) \in \Gamma\end{array}}{\{p\}\ S\ \{I \wedge TERM\}}$$

and call the resulting proof system PDP, standing for *p*artial correctness of *d*istributed *p*rograms.

When the premises of the above rule are satisfied then we say that I is a *global invariant relative to p.* Also, we shall refer to a statement of the form $Eff(\alpha_{i,j}, \alpha_{k,\ell}); S_{i,j}; S_{k,\ell}$ as a *transition.* An execution of a transition corresponds to a joint execution of a pair of branches of the main loops with matching generalized guards.

Informally the above rule can be phrased as follows. If I is established upon execution of all the $S_{0,i}$ sections and is preserved by each transition then I holds upon termination. This formulation explains why we call I a global invariant. The word "global" relates to the fact that we reason here about all processes simultaneously and consequently adopt a "global" view.

This rule can be justified by relating S to its nondeterministic version $T(S)$.

Similarly as in the previous chapter we now consider weak total correctness. It now combines partial correctness with absence of failures and divergence freedom. We augment the proof system TN for total correctness of nondeterministic programs by the following strengthening of the previous rule

RULE 19: DISTRIBUTED PROGRAMS II

$$\frac{\begin{array}{l}\{p\}\ S_{0,1};\ \ldots;\ S_{0,n}\ \{I\},\\ \{I \wedge B_{i,j} \wedge B_{k,\ell}\}\ Eff(\alpha_{i,j}, \alpha_{k,\ell});\ S_{i,j};\ S_{k,\ell}\ \{I\}\\ \quad \text{for all } (i,j,k,\ell) \in \Gamma,\\ \{I \wedge B_{i,j} \wedge B_{k,\ell} \wedge t = z\}\ Eff(\alpha_{i,j}, \alpha_{k,\ell});\ S_{i,j};\ S_{k,\ell}\ \{t < z\}\\ \quad \text{for all } (i,j,k,\ell) \in \Gamma,\\ I \rightarrow t \geq 0\end{array}}{\{p\}\ S\ \{I \wedge TERM\}}$$

where t is an integer expression and z is an integer variable which does not appear in t or P.

Again, this rule can be justified by relating S to $T(S)$. We call the resulting proof system WDP standing for *w*eak total correctness of *d*istributed *p*rograms.

Finally, consider total correctness. We have to take care of deadlock freedom. We now augment the proof system TN for total correctness of nondeterministic programs by a strengthened version of the last rule. It has the following form:

RULE 20: DISTRIBUTED PROGRAMS III

$$\frac{\begin{array}{l}\{p\}\ S_{0,1};\ \ldots;\ S_{0,n}\ \{I\},\\ \{I \wedge B_{i,j} \wedge B_{k,\ell}\}\ \mathit{Eff}(\alpha_{i,j},\alpha_{k,\ell});\ S_{i,j};\ S_{k,\ell}\ \{I\}\\ \quad \text{for all } (i,j,k,\ell) \in \Gamma,\\ \{I \wedge B_{i,j} \wedge B_{k,\ell} \wedge t = z\}\ \mathit{Eff}(\alpha_{i,j},\alpha_{k,\ell});\ S_{i,j};\ S_{k,\ell}\ \{t < z\}\\ \quad \text{for all } (i,j,k,\ell) \in \Gamma,\\ I \rightarrow t \geq 0,\\ I \wedge \mathit{BLOCK} \rightarrow \mathit{TERM}\end{array}}{\{p\}\ S\ \{I \wedge \mathit{TERM}\}}$$

The new premise allows us to deduce additionally that S is deadlock free relative to p, and consequently to infer the conclusion in the sense of total correctness. We call the resulting proof system TDP standing for *t*otal correctness of *d*istributed *p*rograms.

Also, we shall use the following additional rules which allow us to present the proofs in a more convenient way.

RULE D3:

$$\frac{I_1 \text{ and } I_2 \text{ are global invariant relative to } p}{I_1 \wedge I_2 \text{ is a global invariant relative to } p}$$

RULE D4:

$$\frac{\begin{array}{l}I \text{ is a global invariant relative to } p,\\ \{p\}\ S\ \{q\}\end{array}}{\{p\}\ S\ \{I \wedge q\}}$$

This rule can be used in proofs of partial, weak total or total correctness.

RULE 21:

$$\frac{\begin{array}{l}I \text{ is a global invariant relative to } p,\\ I \wedge \mathit{BLOCK} \rightarrow \mathit{TERM}\end{array}}{S \text{ is deadlock free relative to } p}$$

Note that Rule D3 has several conclusions so it is actually a convenient shorthand for a number of closely related rules. Rules D3 and D4 are actually derived rules (hence their numbering), whereas Rule 21 allows us to reason about deadlock freedom separately.

To illustrate the use of the proof systems we now prove correctness of the program from Example 8.2.

Example 8.4 We prove

$$\{k \geq 1\}\ S\ \{A[1:k] = B[1:k]\}$$

in the sense of total correctness. To this purpose we choose

$$I \equiv A[1:i-1] = B[1:j-1] \wedge i = j \wedge 1 \leq i \leq k+1$$

and

$$t \equiv k+1-i.$$

There is only one transition to consider. Clearly

$$\{I \wedge i \neq k+1 \wedge j \neq k+1\}\ B[j] := A[i];\ i := i+1;\ j := j+1\ \{I\}$$

holds. Other premises of Rule 20 are equally simple to establish. By Rule 20 and the consequence rule the desired conclusion follows. □

8.4 Verification: The Producer Consumer Problem

The program S given in Example 8.3 is a typical instance of the producer consumer problem originally studied in Section 7.3. The process *FILTER* acts as an intermediary process between the process *BUFFER* playing here a role of a producer and the process *CONSOLE* playing here a role of a consumer. We now prove correctness of this program.

We first formalize the property we wish to prove. Given an array variable A of type **integer** $\rightarrow$ **character**, we call a section $A[i:j]$ a *string*. For two strings $A[i:j]$ and $B[k:l]$, we write $A[i:j] = B[k:l]$ if they are equal (as sequences). Given a string $A[1:k]$ we define $delete(A[1:k])$ as the string $B[1:n]$ which results from $A[1:k]$ by deleting all blanks. Thus $delete(A[1:k]) = B[1:n]$ iff the following three conditions hold:

(i) $n = k - \#\{i : A[i] = \text{`}\ \text{'}\}$,

(ii) $\forall i(1 \leq i \leq n).B[i] \neq \text{`}\ \text{'}$ ($B[1:n]$ contains no blanks),

(iii) for some 1–1 order preserving function $f : \{1,\ldots,n\} \rightarrow \{1,\ldots,k\}$
$\forall i(1 \leq i \leq n).B[i] = A[f(i)]$
(i.e. $B[1:n]$ results from $A[1:k]$ by deleting some characters).

Here, $\#A$ stands for the cardinality of the set A. Indeed, note that by (i) and (iii) $B[1:n]$ results from $A[1:k]$ by deleting a number of characters

equal to the number of blanks in $A[1:k]$. Now by (ii) the deleted characters are exactly all blank characters of $A[1:k]$.

Note 8.5 For all $i = 1, \ldots, k-1$

(i) if $A[i+1] = \text{`}\ \text{'}$ then
$delete(A[1:i+1]) = delete(A[1:i])$,

(ii) if $A[i+1] \neq \text{`}\ \text{'}$ then
$delete(A[1:i+1]) = delete(A[1:i]) \wedge A[i+1]$
where "$\wedge$" appends the character at the end of the string.

□

Correctness of the program S now means that for

$$p \equiv k \geq 1 \wedge A[k] = \text{`}*\text{'} \wedge \forall i(1 \leq i < k).A[i] \neq \text{`}*\text{'}$$

the correctness formula

$$\{p\}\ S\ \{C[1:n-1] = delete(A[1:k])\} \tag{1}$$

is true in the sense of total correctness.

Step 1 We first prove (1) in the sense of partial correctness. To this purpose we first look for an appropriate global invariant I of S (relative to the initial assertion p).

We put

$$\begin{aligned} I \equiv \quad & B[1:rec-1] = delete(A[1:i-1]) \\ \wedge\ & B[1:send-1] = C[1:n-1] \\ \wedge\ & send \leq rec. \end{aligned}$$

We now check that I indeed satisfies the premises of Rule 18.

1°. We clearly have

$$\begin{array}{l} \{p\} \\ i := 1;\ send := 1;\ rec := 1; \\ b := \text{`}\ \text{'};\ n := 1;\ c := \text{`}\ \text{'} \\ \{I\} \end{array}$$

as by convention for any array a the string $a[1:0]$ is empty.

2°. We have here two pairs of matching i/o commands:

$$(FILTER!A[i], BUFFER?b)$$

and

$$(CONSOLE!B[send], FILTER?c).$$

We consider them in turn.

(i) We prove the following correctness formula:

$\{I \wedge i \neq k+1 \wedge b \neq \text{'*'}\}$
$b := A[i];\ i := i+1;$
if $b = \text{' '} \rightarrow$ *skip*
$\Box b \neq \text{' '} \rightarrow B[rec] := b;$
$rec := rec + 1$
fi
$\{I\}.$

To this purpose first observe that by Note 8.5

$B[1 : rec-1] = delete(A[1 : i-2]) \wedge A[i-1] = b \wedge b = \text{' '}\}$
skip
$\{B[1 : rec-1] = delete(A[1 : i-1])\}$

and

$\{B[1 : rec-1] = delete(A[1 : i-2]) \wedge A[i-1] = b \wedge b \neq \text{' '}\}$
$B[rec] := b;\ rec := rec+1$
$\{B[1 : rec-1] = delete(A[1 : i-1])\}$

hold.

Now by the alternative command rule and the composition rule

$\{B[1 : rec-1] = delete(A[1 : i-1])\}$
$b := A[i];\ i := i+1;$
if $b = \text{' '} \rightarrow$ *skip*
$\Box b \neq \text{' '} \rightarrow B[rec] := b;$
$rec := rec + 1$
fi
$\{B[1 : rec-1] = delete(A[1 : i-1])\}$

holds.

To obtain the desired correctness formula it suffices now to conjoin all assertions in the above proof with the assertion

$$B[1 : send-1] = C[1 : n-1] \wedge send \leq rec$$

which remains invariant.

(ii) We prove the following correctness formula

$\{I \wedge send \neq rec \wedge c \neq \text{'*'}\}$
$c := B[send];\ send := send+1;$
$C[n] := c;\ n := n+1$
$\{I\}.$

First observe that

$\{B[1 : send-1] = C[1 : n-1] \wedge send < rec\}$
$c := B[send];\ send := send+1;$
$C[n] := c;\ n := n+1$
$\{B[1 : send-1] = C[1 : n-1] \wedge send \leq rec\}$

from where the above correctness formula easily follows by conjoining the assertions with the assertion

$$B[1 : rec - 1] = delete(A[1 : i - 1])$$

which remains invariant. Thus I is indeed a global invariant relative to p.

By Rule 18 we now obtain from 1° and 2° the correctness formula

$$\{p\}\ S\ \{I \wedge TERM\}$$

in the sense of partial correctness. Here

$$TERM \equiv i = k + 1 \wedge b = \text{`}*\text{'} \wedge send = rec \wedge c = \text{`}*\text{'}.$$

By the consequence rule (1) holds in the sense of partial correctness.

Step 2 We now prove (1) in the sense of weak total correctness. To this end we exhibit an appropriate bound function by putting

$$t \equiv 2 \cdot (k - i + 1) + rec - send$$

which guarantees a decrease when *both* i and *send* are incremented by 1.

However, to apply Rule 19 we need to use an invariant which guarantees that t remains non-negative. We put

$$I_1 \equiv i \leq k + 1.$$

It is straightforward to prove that I_1 and t satisfy the premises of Rule 20, where TN is used as the underlying proof system.

By Rule 19 and Rule D4 we now get

$$\{p\}\ S\ \{I \wedge I_1 \wedge TERM\}$$

in the sense of weak total correctness which implies (1) in the sense of weak total correctness.

Step 3 Finally, we prove deadlock freedom. By Rule 21, it suffices to find a global invariant I' (relative to p) for which

$$I' \wedge BLOCK \rightarrow TERM \tag{2}$$

holds. Here

$$BLOCK \equiv (i = k + 1 \vee b = \text{`}*\text{'}) \wedge (send = rec \vee c = \text{`}*\text{'}).$$

We use Rule D3 and exhibit I' "in stages." First we wish to find a global invariant I_2 such that

$$I_2 \rightarrow (i = k + 1 \leftrightarrow b = \text{`}*\text{'}). \tag{3}$$

Next, we wish to find global invariants I_3 and I_4 for which

$$I_3 \wedge i = k+1 \wedge b = \text{`}*\text{'} \wedge send = rec \rightarrow c = \text{`}*\text{'} \tag{4}$$

and

$$I_4 \wedge i = k+1 \wedge b = \text{`}*\text{'} \wedge c = \text{`}*\text{'} \rightarrow send = rec \tag{5}$$

holds.

Then by Rule D3 and (3), (4) and (5)

$$I' \equiv I_2 \wedge I_3 \wedge I_4$$

is a global invariant. Note that each of the equalities used in (3), (4) and (5) is a conjunct of *TERM*; (3), (4) and (5) express certain implications between these conjuncts which guarantee that I' indeed satisfies (2).

First, we put

$$I_2 \equiv p \wedge (i > 1 \vee b = \text{`}*\text{'} \rightarrow b = A[i-1]).$$

I_2 relates variables of the processes *BUFFER* and *FILTER*. Note that (3) holds.

Next, we put

$$I_3 \equiv I \wedge p \wedge (n > 1 \rightarrow c = C[n-1]).$$

The last conjunct of I_3 states a simple property of the variables of the process *CONSOLE*. We have the following sequence of implications

$I_3 \wedge i = k+1 \wedge b = \text{`}*\text{'} \wedge send = rec \rightarrow$
$I_3 \wedge C[1:n-1] = delete(A[1:k]) \rightarrow$
$I_3 \wedge C[n-1] = \text{`}*\text{'} \wedge n > 1 \rightarrow$
$c = \text{`}*\text{'}.$

Finally, we put

$$I_4 \equiv I \wedge p \wedge (c = \text{`}*\text{'} \rightarrow C[n-1] = \text{`}*\text{'}).$$

Here as well, the last conjunct describes a simple property of the variables of the process *CONSOLE*. We have the following sequence of implications

$I_4 \wedge i = k+1 \wedge b = \text{`}*\text{'} \wedge c = \text{`}*\text{'} \rightarrow$
$I_4 \wedge C[n-1] = \text{`}*\text{'} \rightarrow$
$I_4 \wedge B[send-1] = A[k] \rightarrow$
$f(send-1) = k \wedge send - 1 \leq rec - 1 \leq k \wedge f(rec-1) \leq k$
for some $1-1$ order preserving function
$f : \{1,\ldots,n-1\} \rightarrow \{1,\ldots,k\}$
(see clause (iii) of the definition of $delete(A[1:k])$) $\rightarrow$
$send = rec.$

Thus we showed (4) and (5). Moreover, it is straightforward to see that each of I_2, I_3 and I_4 is indeed a global invariant. We have thus proved (2).

This concludes the proof of the correctness formula (1) in the sense of total correctness.

8.5 Conclusions

A key to the proper understanding of the proof systems PDP, WDP and TDP studied in this chapter is observation made in Section 8.2 that every simple distributed program S is equivalent to a nondeterministic program $T(S)$. This equivalence allows us to prove correctness of S by proving correctness of $T(S)$ instead and the Rules 18, 19 and 20 allow us to do just this—their premises refer to the subprograms of $T(S)$ and not S.

The same approach could be used when dealing with parallel programs. However, there such a translation of a parallel program into a nondeterministic one would necessitate a use of auxiliary variables. This would add to the complexity of the proofs and would make the approach clumsy and artificial. Here, thanks to a special form of the programs, the translation turns out to be very simple. We can summarize this discussion by conceding that the proof method presented here exploits the particular form of the programs studied.

References

[1] K. R. Apt [1986], Correctness proofs of distributed termination algorithms, *ACM TOPLAS 8*, pp. 388–405, 1986.

[2] K.R. Apt, N. Francez and W. P. de Roever [1981], A proof system for communicating sequential processes, *ACM TOPLAS 2*, pp. 359–385, 1980.

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

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

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

[6] D. Gries [1982], A note on a standard strategy for developing loop invariants and loops, *Science of Computer Programming 2*, pp. 207–214, 1982.

[7] R. Floyd [1967], Assigning meaning to programs, in: *Proc. Symp. on Appl. Math. 19* (Math. Aspects of Comp. Sci.), (J.T. Schwartz, ed.), pp. 19–32, American Math. Society, New York, 1967.

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

[9] C. A. R. Hoare [1972], Towards a theory of parallel programming, in: Operating Systems Techniques (C.A.R. Hoare, R.H. Perrot, eds.), pp. 61–71, Academic Press, 1972.

[10] C. A. R. Hoare [1978], Communicating Sequential Processes, *Communications of the ACM 21*, pp. 666–677, 1978.

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

[12] G. Levin and D. Gries [1981], A proof technique for Communicating Sequential Processes, *Acta Informatica 15*, pp. 281–302, 1981.

[13] E. R. Olderog and K. R. Apt [1988], Fairness in parallel programs, the transformational approach, *ACM TOPLAS 10*, pp.420–455, 1988.

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

[15] A. M. Turing [1949], On checking a large routine, in: *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 67–69, 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*, pp. 139–143, 1984.