Part I

Introduction and Overview

1
Introduction

1.1 Central Questions

The subject of this work is how to prove correctness of concurrent programs. To argue the relevance of this theme a number of questions should be answered:

- How important is verification for the development of correct software?
- Why does one need to give correctness proofs for concurrent programs? Must these be formal? Or even machine checked?
- Which style of proof method is more appropriate, a compositional or non-compositional one?

These questions will be answered in the present chapter.

Note that here, as elsewhere in this volume, concurrency is used as a generic term covering the execution mechanisms for programs which communicate through distributed message passing as well as through shared variables. Such programs will also be called parallel programs. If we want to focus on programs communicating through distributed message passing we use the term *distributed* programs . Such programs are implemented on distributed locations and communicate by some form of message exchange. When no such physical separation is intended, communication is usually by means of shared variables, and we speak of *shared-variable concurrency*.

1.2 Structure of this Chapter

In Section 1.3 we define some basic concepts in the theory of concurrency which we shall need later. We argue in Section 1.4 that concurrent programs should be proved correct because of their unimaginable and bewildering complexity, thereby answering the first two questions posed above. We do so by

starting out with a small refinement problem in the context of a simple mutual exclusion algorithm due to Peterson [Pet83], then discuss several attempted solutions in the area of concurrent garbage collection, consider a distributed mutual exclusion algorithm due to Szymanski [Szy88], and finish Section 1.4 by concluding that operational reasoning gives no guarantee that a program satisfies its specifications. Instead one should always use state-based reasoning for concurrent programs – that is, proofs based on invariance properties. Especially when these proofs are large and complicated, one should use (semi-)automated proof checkers to eliminate errors in hand-checked verification proofs. This answers the first two questions raised above.

Section 1.5 discusses the approach followed in this book, which is a semantically-based dual-language state-based property-oriented approach, gives a brief introduction to Floyd's inductive assertion method and Hoare logic, then discusses their extensions to concurrency – first noncompositional proof methods, and then compositional ones based on the assume-guarantee paradigm and finishes by briefly describing a noncompositional proof method called the communication-closed-layers paradigm suitable for verifying network algorithms and protocols.

Section 1.6 discusses compositionality, one of the leading themes in this work. This central concept is approached from the viewpoint of the verify-while-develop paradigm, thus enabling a detailed analysis of its definition. Then its relationship to machine-supported verification and modularity is explained, followed by a discussion of the complexity of compositional reasoning. We finish the section by answering the third question posed above.

This chapter ends with Section 1.7, which puts the development of noncompositional to compositional state-based proof methods for concurrency into a historical perspective.

1.3 Basic Concepts of Concurrency

Why should one prove concurrent programs correct? As a preparation for answering this question in the next section, we first introduce some basic concepts for reasoning about such programs. Next we sketch a way to understand them by defining their meaning. Then we discuss communication and synchronisation, which are the two main forms of interaction between processes executing in parallel.

1.3.1 Differences between Sequential Programming and Concurrency

But first: *What are the differences between sequential programs on the one*

hand and concurrent programs on the other?

First and foremost, a sequential program is intended to run on a single processor, whereas a concurrent program can be executed in principle on as many processors as there are parallel components in the program. The second difference concerns our way of characterising such programs.

For the characterisation of sequential programs it is sufficient to observe their pairs of initial and corresponding final states. For a given program, the set of such pairs is called its *observable behaviour.* (Note that even for sequential programs, there are different notions of such behaviour, varying according to whether or not, e.g., nontermination and/or runtime errors are also observed.) Given such a pair of initial and corresponding final states, it is not necessary to record *how* that final state has been computed from the given initial state. This is so, because two different sequential programs having the same observational behaviour are regarded as equivalent, since, whenever they are plugged as modules inside a third program, this causes no difference in observational behaviour of the latter. That is, from this point of view, sequential programs can be regarded as *atomic* units.

In case of concurrent programs, the same characterisation would not suffice, because the possibility of synchronisation and communication between such programs makes intermediate states as important as final ones. Hence the observational behaviour of these programs should also include some observable form of intermediate states, e.g., the values of those variables which are shared between processes or the messages communicated between them.

In general, a *concurrent program* consists of a collection of processes and shared objects, such as shared channels and/or shared variables. Each process can be considered as a sequential program which can run concurrently with other processes within the same program. The shared objects allow these processes to cooperate in accomplishing some task. They can be implemented in shared memory or might simply be a computer-communication network.

Let us illustrate these concepts.

- *Shared memory* (cf. Figure 1.1): External processes P_1 and P_2 both have access to a pool of shared memory cells. What should be prevented is that P_1 is able to access information in this shared memory while the information is being changed by P_2, and vice versa, because this would cause that information to become temporarily undefined. We want to do this independently of the specific nature of P_1 and P_2, i.e., we need a solution which is canonical w.r.t. programs accessing shared memory. To obtain such a solution, processes P_1 and P_2 have to be synchronised w.r.t. reading and writing memory cells; this is further illustrated in Section 1.4.

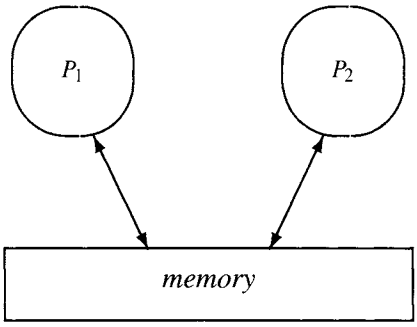

Fig. 1.1. Shared memory concept.

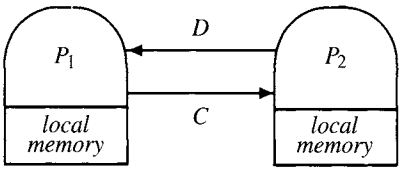

Fig. 1.2. Computer-communication network concept.

- *Computer-communication network* (cf. Figure 1.2): Every process has its own local memory. However, processes share channels. Hence, in order to implement message passing one needs to guarantee that when, e.g., P_1 sends a message along a channel C to P_2, and this message reaches P_2, P_2 eventually reacts to that message. And this again requires some form of synchronisation between, at the very least, P_2 and its environment.

When formulating proof rules and discussing compositionality we need to characterise the observable behaviour of a program component or process more precisely by its so-called *observables*. In a purely sequential context, one sometimes needs to know the precise set of variables occurring, or involved (this term is explained below), in that component, i.e., those variables whose values are read and/or changed during execution of that component. These variables are called the observables of that component. In case of shared-variable concurrency, the observables of a component consist of two sets, the set of variables occurring, or involved, in that component, and the set of variables through which that component communicates with its environment. In a case of message passing, the observables of a component also consist of two sets, the set of variables occurring, or involved, in that component, and the set of channels through which it communicates with its environment. In this book no mixtures of these two communication mechanisms are considered.

In a purely semantic setting we shall use the term "involved in" to express the dependency of a process upon (the values of) variables and/or (the values communicated along) channels, e.g., a variable or channel involved in a process, whereas in a syntactic setting (i.e., in Chapters 5, 10, 11 and 12) we use "occurring in" for this purpose, because then one can point to the particular syntactic occurrences of the variable or channel concerned. These terms will also be defined in a semantic setting for boolean-valued functions, and in a syntactic setting for assertions from first-order predicate logic.

1.3.2 Semantics of Concurrency

Next we give a model for understanding such programs [SA86]. A program *state* σ associates a value with each variable. Execution of a sequential program results in a sequence of *atomic transitions,* each of which transforms the state indivisibly. Execution of a concurrent program results in an interleaving of these sequences of atomic transitions for each component process and can be essentially viewed as a *history* or *computation sequence*

$$\sigma_0 \stackrel{\alpha_1}{\rightarrow} \sigma_1 \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_i}{\rightarrow} \sigma_i \stackrel{\alpha_{i+1}}{\rightarrow} \cdots$$

where the σ_i's denote the states, the α_i's denote atomic transitions, and the sequence $\alpha_1\alpha_2\ldots$ is an interleaving of these sequences of atomic transitions resulting from execution of the processes.

The behaviour of a concurrent program is defined by its set of histories, each history corresponding to one possible interleaving of those sequences of atomic actions that result from execution of its processes. For all but trivial programs, this set is apt to be quite large – so large that it might be impossible to enumerate, much less inspect, each of its elements in order to ascertain aspects of the behaviour of the program. This will be illustrated by the examples in Section 1.4. Therefore, we take in Section 1.5 an approach for developing and analysing such programs which is based on the use of abstraction; it is called *assertional reasoning*. Instead of enumerating sets of sequences of states, we characterise the computation sequences in these sets by describing their properties of interest. Instead of enumerating program states, we use *predicates* (boolean functions) which are expressible by *assertions* – formulae of predicate logic – to characterise sets of states. As a result, use of assertional reasoning allows a program to be understood as a *relation between assertions, rather than as an object that is executed.* As will be clear from Section 1.4 onwards, this change of viewpoint is crucial to our understanding of concurrent programs, for it allows us to master their complexity by assertional, or even compositional, reasoning.

1.3.3 Communication and Synchronisation

Thirdly we discuss the important subjects of communication and synchronisation between processes, which have already been introduced above.

In order to interact, processes must communicate and synchronise. *Communication* allows one process to influence execution of another one and can be accomplished using *shared variables* or *message passing*. When shared variables are used, a process writes to a variable that is read by another process; when message passing is used, a process sends a message to another process.

To communicate, one process sets the state of a shared object and the other reads it. This works only if the shared object is read after it has been written – reading the object before it is written can return a meaningful, but erroneous, value. Thus, meaningful communication between processes cannot occur without preventing the latter; this is called synchronisation.

Three notions of synchronisation appear in this book. The first, *mutual exclusion,* groups actions into *critical sections* that are never interleaved during execution. Within these critical sections no program variables are changed which influence the control flow of the synchronisation mechanism implemented (i.e., mutual exclusion). The second form, *conditional synchronisation*, delays a process until the state satisfies some specified condition. This synchronisation mechanism is, e.g., used in order to prevent the value of a shared variable being read in one process before that variable has been written in another process, as discussed above. The third form arises when communication between processes itself is synchronised. Then the acts of sending and receiving messages along given channels are synchronised, and, hence, one speaks of *synchronous communication.* The remaining forms of communication between processes are then called *asynchronous.* For example, communication by shared variables is asynchronous. All these forms of synchronisation restrict interleavings of processes. Mutual exclusion restricts interleavings by preventing interleaving from occurring at certain internal control points in a process; conditional synchronisation and synchronous communication restrict interleavings by causing a process to be delayed at given control points.

A simple example illustrates these types of synchronisation. Communication between a sender process and a receiver process is often implemented using a shared buffer. The sender writes into the buffer; the reader reads from the buffer. Mutual exclusion is used to ensure that a partially written message is not read – access to the buffer by the sender and receiver is made *mutually exclusive*. Conditional synchronisation is used to ensure that the message is not overwritten or read twice – the reader is prevented from reading the buffer until a new message has been written. In a case of synchronous communica-

tion, special protocols, which are implemented using such buffers at a lower level of abstraction, convey the impression at a higher level of abstraction that communication between processes along the same channel is simultaneous.

Having introduced these concepts, in the next section we answer the first two questions raised in Section 1.1.[1]

1.4 Why Concurrent Programs Should be Proved Correct

1.4.1 Introduction

In this section we answer the first two questions raised in Section 1.1. In particular, we argue that it is in general impossible to convince oneself of the correctness of concurrent programs without recourse to formal methods. This we do by giving a number of problems which are characteristic for the area, and for which we propose a number of increasingly complicated incorrect approximations until we finally arrive at an alleged correct solution. The separate steps in the derivations of these solutions are certainly in themselves trivial, and yet several errors, sometimes subtle ones, will be shown to creep in. The design of the final solutions proceeds through a disquieting series of trials and errors. It should be clear that an informal justification of programs constructed in such a manner is not sufficient. The difficulties illustrated explain why we are interested in proving the correctness of such programs.[2]

The first example concerns Peterson's mutual exclusion algorithm [Pet83] in a presentation due to Amir Pnueli. The second example deals with the difficult task of concurrent garbage collection; specifically, we discuss a deep logical bug in one of the first concurrent garbage collectors ever written, by E.W. Dijkstra, L. Lamport and others, and how to correct it [DLM+78]. Manna & Pnueli's paper [MP91c] drew our attention to the third example concerning a sophisticated distributed mutual exclusion algorithm due to Szymanski [Szy88].

As pointed out by Leslie Lamport, the history of concurrent algorithms seems to abound with published incorrect algorithms. He writes to de Roever [Lam99]:

> To my knowledge, the first published concurrent algorithm was Dijkstra's '65 mutual exclusion paper [Dij65b]. The second one was by H. Hyman in a letter to the *CACM* proposing a simple mutex algorithm for two processors [Hym66]. The third algorithm was by Knuth, also in a letter to the *CACM*, in which he improved Dijkstra's algorithm and pointed out that Hyman's was completely wrong [Knu66].

[1] Parts of Section 1.3 originate from [SA86].
[2] This paragraph has been inspired by a similar paragraph in [AO91].

And also, are you aware of the error in Ben-Ari's garbage collection paper [BA82, BA84]? He visited me at SRI soon after that paper was published, and I was discussing proofs with him. ... Going over the proof of one of the algorithms in that paper, and trying to turn it into a real invariance proof, we discovered that the algorithm was wrong.[3]

But sequential algorithms and their correctness proofs also share this fate, as pointed out by Donald Knuth, in an interview with Lád'a Lhotka about structured programming [Knu99, pp. 613–614]:

I was talking with Tony Hoare, who was editor of a series of books ... He told me I ought to *publish* my program for TEX.

As I was writing TEX I was using for the second time in my life a set of ideas called "structured programming", which were revolutionizing the way computer programming was done in the middle 70s (sic). ... Well, this was frightening – a very scary thing, for a professor of computer science to show someone a large program. At best, a professor might publish very small routines as examples of how to write programs. And we could polish those until ... well, every example in the literature about such programs had bugs in it. Tony Hoare was a great pioneer for proving the correctness of programs. But if you looked at the details ... I discovered from reading some of the articles, you know, I could find three bugs in a program that was proved correct. [*laughter*] These were *small* programs. Now, he says, take my *large* program and reveal it to the world, with all its compromises. ... But then I also realized how much need there was for examples of fairly large programs that could be considered as reasonable models of good practice, not just small programs.

1.4.2 First Example: Peterson's Mutual Exclusion Algorithm

Consider the following solution to the mutual exclusion problem:

Example 1.1 Let

$P_1 \equiv \ell_0$: **loop forever do**
ℓ_1 : **noncritical section**;
ℓ_2 : $(y_1, s) := (1, 1)$;
ℓ_3 : **wait** $(y_2 = 0) \vee (s \neq 1)$;
ℓ_4 : **critical section**;
ℓ_5 : $y_1 := 0$
od

and

[3] The history of concurrent garbage collection and errors in its algorithms is extensively documented in Section 8.10 of [JL96]; e.g., the errors in Ben-Ari's algorithm were published independently by J. van de Snepscheut, C. Pixley, and D. Doligez & X. Leroy.

$P_2 \equiv m_0$: **loop forever do**
m_1 : **noncritical section**;
m_2 : $(y_2, s) := (1, 2)$;
m_3 : **wait** $(y_1 = 0) \vee (s \neq 2)$;
m_4 : **critical section**;
m_5 : $y_2 := 0$
od

and let

$$Pet1 \equiv s := 1; y_1 := 0; y_2 := 0; [P_1 || P_2].$$ □

The statements **noncritical section** and **critical section** in mutual exclusion algorithms do not change the values of the variables which are used to implement the mutual exclusion mechanism, i.e., in this particular case, do not change the values of y_1, y_2 and s. Here **wait** b is a statement expressing conditional synchronisation. It acts like a traffic light that can only be passed when b evaluates to *true*. Now consider an occurrence of **wait** $(y_j = 0) \vee (s \neq i)$, for $j \neq i$, within a process P_i; then making $(y_j = 0) \vee (s \neq i)$ true is typically done inside another process P_j which operates in parallel with P_i, hence $j \neq i$. This explains the name conditional synchronisation.

Integer variables y_1 and y_2 are used by each process to signal the other process of active interest in entering the critical section. Thus, on leaving the noncritical section, process P_i sets its own variable y_i, $i = 1, 2$, to 1 indicating interest in entering the critical section. In a similar way, on exiting the critical section, P_i resets y_i to 0. Variable s is used to resolve a tie situation between the two processes, which may arise when both processes are actively interested in entering their critical sections at the same time.

Variable s serves as a logbook in which each process that sets its y variable to 1 signs as it does so. The test at ℓ_3 says that P_1 may enter its critical section if either $y_2 = 0$, implying that P_2 is not interested in entering a critical section, or if $s \neq 1$, implying that P_2 performed its assignment to y_2 *after* P_1 assigned 1 to y_1, and, consequently, can only pass m_3 after P_1 has executed its critical section. Since P_1 and P_2 are symmetric, the same kind of reasoning applies when the rôles of P_1 and P_2 are interchanged. Consequently, *Pet1* implements mutual exclusion of the critical sections of P_1 and P_2 (using conditional synchronisation).

Formal methods can help to prove this intuition formally, i.e., proving that no computation of program *Pet1* contains a state in which P_1 is executing at ℓ_4 while P_2 is executing at m_4 (cf. Exercise 3.4). (This can even be verified automatically.)