CONCURRENCY VERIFICATION
Introduction to Compositional and Noncompositional Methods

Willem-Paul de Roever
Frank de Boer
Ulrich Hannemann
Jozef Hooman
Yassine Lakhnech
Mannes Poel
Job Zwiers

Contents

Preface		*page* x
	Part I: Introduction and Overview	**1**
1	**Introduction**	**2**
1.1	Central Questions	2
1.2	Structure of this Chapter	2
1.3	Basic Concepts of Concurrency	3
1.3.1	Differences between Sequential Programming and Concurrency	3
1.3.2	Semantics of Concurrency	6
1.3.3	Communication and Synchronisation	7
1.4	Why Concurrent Programs Should be Proved Correct	8
1.4.1	Introduction	8
1.4.2	First Example: Peterson's Mutual Exclusion Algorithm	9
1.4.3	Second Example: Concurrent Garbage Collection	13
1.4.4	Third Example: Szymanski's Mutual Exclusion Algorithm	23
1.4.5	Conclusion: Answers to Two Central Questions	30
1.4.6	Hand-Checked or Machine-Supported Verification?	30
1.5	The Approach of this Book	33
1.5.1	A State-Based Dual-Language Approach	33
1.5.2	A Semantic Approach	34
1.5.3	Floyd's Inductive Assertion Method	35
1.5.4	Hoare Logic	37
1.5.5	Extensions to Concurrency	39
1.5.6	The Assume-Guarantee Paradigm	41
1.5.7	The Communication-Closed-Layers Paradigm	45
1.6	Compositionality	46
1.6.1	The Verify-While-Develop Paradigm	47
1.6.2	Compositionality: the Very Concept	48

1.6.3 Compositionality and Machine-Supported Verification 55
1.6.4 Compositionality, Completeness and Modularity 56
1.6.5 On the Complexity of Compositional Reasoning 58
1.6.6 Compositional or Noncompositional Methods? 60
1.7 From Noncomp. to Comp. Proof Methods – a historical perspective 62

Part II: The Inductive Assertion Method 71

2 Floyd's Inductive Assertion Method for Transition Diagrams 72
2.1 Objectives of Part II 72
2.2 Structure of this Chapter 75
2.3 Sequential Transition Diagrams and Systems 76
2.3.1 Sequential Transition Diagrams 76
2.3.2 Transition Systems 80
2.4 Specification and Correctness Statements 82
2.4.1 Specifications, Logical Variables and Program Variables 82
2.4.2 Correctness Statements 86
2.5 A Proof Method for Partial Correctness 88
2.5.1 Definition plus Example 88
2.5.2 Floyd's Inductive Assertion Method 90
2.6 Soundness 92
2.7 Semantic Completeness of the Inductive Assertion Method 93
2.8 Proving Convergence 98
2.8.1 Wellfounded Sets 98
2.8.2 A Proof Method for Convergence 99
2.8.3 Soundness and Semantic Completeness 101
2.9 Proving Absence of Runtime Errors 104
2.9.1 A Direct Method for Proving Absence of Runtime Errors 106
2.9.2 An Implicit Method for Proving Absence of Runtime Errors 108
2.10 Historical Notes 111

3 The Inductive Assertion Method for Shared-Variable Concurrency 119
3.1 Objective and Structure of this Chapter 119
3.2 A Characterisation of Concurrent Execution 121
3.2.1 Basic Requirements and Atomic Actions 122
3.2.2 The Need for Fairness Assumptions 128
3.3 Is this Characterisation of Concurrent Execution Justified? 132
3.4 The Generalisation of Floyd's Approach to Nondeterministic Interleavings 134
3.5 Concurrent Transition Systems with Shared Variables 137
3.5.1 The Proof Method of Owicki & Gries 137

3.5.2 Examples 150
3.5.3 Soundness of the Owicki & Gries Proof Method 165
3.5.4 Completeness of the Proof Method of Owicki & Gries: Introduction 167
3.5.5 Technical Note: An Adaptation Rule 176
3.5.6 Completeness of the Method of Owicki & Gries: a Compositional Approach 177
3.5.7 An Alternative Completeness Proof 185
3.6 Proving Convergence for Shared-Variable Concurrency 188
3.6.1 A Global Method 189
3.6.2 A Local Method 190
3.6.3 Soundness and Semantic Completeness of the Local Method for Proving Convergence 196
3.7 Proving Deadlock Freedom 203
3.8 Proving Absence of Runtime Errors 206
3.9 Historical Notes 208
3.9.1 The Merging Lemma, a Combinatorial Approach to Completeness 208
3.9.2 Lamport's View of the Method of Owicki & Gries 212
3.9.3 More Background 214

4 The Inductive Assertion Method for Synchronous Message Passing 221

4.1 Objective and Introduction 221
4.2 Structure of this Chapter 223
4.3 Syntax and Semantics of Synchronous Transition Diagrams 223
4.4 Proof Methods for Partial Correctness 227
4.4.1 The Proof Method of Apt, Francez & de Roever 227
4.4.2 Example 238
4.4.3 Technical Note: The Proof Method of Levin & Gries 243
4.5 Semantic Completeness 249
4.5.1 Compositional Semantics of Synchronous Transition Diagrams 249
4.5.2 Semantic Completeness of the AFR-Method 255
4.5.3 Technical Note: Semantic Completeness of the Method of Levin & Gries 259
4.6 Technical Note: Modifications Towards Compositionality 264
4.6.1 Removing the Cooperation Test 265
4.6.2 Removing the Interference Freedom Test 267
4.6.3 Towards a Compositional Method 268
4.7 A Modular Method for Proving Convergence 269
4.7.1 Soundness and Semantic Completeness 271
4.8 Verifying Deadlock Freedom 277

4.9	Proving Absence of Runtime Errors	279
4.9.1	A Direct Method for Proving Absence of Runtime Errors	280
4.9.2	An Implicit Method for Proving Absence of Runtime Errors	281
4.10	Historical Notes	282
5	**Expressibility and Relative Completeness**	**291**
5.1	Objective	291
5.2	Structure of this Chapter	292
5.3	Syntactic Notions	292
5.3.1	Expressions and Boolean Expressions	293
5.3.2	Assignments	294
5.3.3	Assertion Language	296
5.4	Partial Correctness of Syntactic Transition Diagrams	298
5.5	Relative Completeness of Floyd's Inductive Assertion Method	300
5.5.1	Expressibility of φ-Minimal Predicates	303
5.6	Relative Completeness of the Method of Owicki & Gries	309
5.7	Relative Completeness of the Method of Apt, Francez & de Roever	312
5.8	Historical Notes	316
Picture Gallery		319
	Part III: Compositional Methods based on Assertion Networks	**353**
6	**Introduction to Compositional Reasoning**	**354**
6.1	Motivation	354
6.2	Introduction to Part III and to this Chapter	356
6.3	Assume-Guarantee-based Reasoning	359
6.4	Assumption-Commitment-based Reasoning	361
6.5	Rely-Guarantee-based Reasoning	363
7	**Compositional Proof Methods: Synchronous Message Passing**	**367**
7.1	Objective and Introduction	367
7.2	Structure of the Chapter	368
7.3	Top-level Synchronous Message Passing	369
7.3.1	Semantics	369
7.3.2	Proof Method	372
7.3.3	Application	376
7.3.4	Soundness and Completeness	379
7.4	A Compositional Proof Method for Nested Parallelism	379
7.4.1	Compositional Semantics	379
7.4.2	A Compositional Proof Method	380
7.4.3	An Extended Example – Specification and Correctness of a Distributed Priority Queue	383

7.4.4 Soundness 390
7.4.5 Semantic Completeness 393
7.5 Assumption-Commitment-based Reasoning 397
7.5.1 Semantics 399
7.5.2 Validity 401
7.5.3 Proof Method and Application 402
7.5.4 Soundness 412
7.5.5 Completeness 417
7.6 Historical Notes 429

8 Compositional Proof Methods: Shared-Variable Concurrency 438
8.1 Introduction and Overview 438
8.2 Concurrent Transition Diagrams 439
8.3 Top-Level Shared-Variable Concurrency 440
8.4 The Rely-Guarantee Method 447
8.4.1 Semantics 447
8.4.2 A Complete R-G Proof Method for Aczel Semantics 451
8.4.3 Application 454
8.4.4 Soundness 459
8.4.5 Completeness 462
8.4.6 The R-G Method and Reactive Sequences 474
8.5 Historical Notes 479

Part IV: Hoare Logic 487

9 A Proof System for Sequential Programs Using Hoare Triples 488
9.1 Introduction and Overview of Hoare Logics 488
9.1.1 Overview of Part IV 488
9.1.2 Hoare Triples 491
9.1.3 Hoare Triples, Compositionality and the Verify-While-Develop Paradigm 494
9.2 Structure of this Chapter 497
9.3 Syntax and Informal Meaning of GCL^+ Programs 498
9.4 Semantics of GCL^+ 501
9.4.1 A Compositional Translation of GCL^+ to Transition Systems 501
9.4.2 An Alternative Approach 506
9.5 A Proof System for GCL^+ Programs 506
9.6 Soundness and Relative Completeness 511
9.7 Proof Outlines 517
9.8 Alternative Definitions of Proof Outlines 521
9.9 Examples of Verification during Program Development 522
9.10 Historical Notes 526

10	**A Hoare Logic for Shared-Variable Concurrency**	**531**
10.1	Introduction and Overview	531
10.2	Syntax and Informal Meaning of SVL Programs	532
10.3	Semantics of SVL^+	537
10.4	A Proof System for SVL Programs	540
10.4.1	A Hoare Logic for Closed Shared-Variable Programs	547
10.4.2	A Historical Note / EWD 554	555
10.4.3	Labelling	559
10.5	An Extended Example: Concurrent Garbage Collection	563
10.5.1	The Mutator and the Collector	564
10.5.2	The Abstract Collector Algorithm	567
10.5.3	A Concrete Implementation of the Garbage Collector	576
10.5.4	Conclusion of the Extended Example	583
10.6	Completeness of the Owicki & Gries Method	584
10.6.1	The Decomposition Lemmas	585
10.6.2	The Completeness Proof	591
11	**A Hoare Logic for Synchronous Message Passing**	**600**
11.1	Structure of this Chapter	600
11.2	Syntax and Informal Meaning of DML Programs	601
11.3	Semantics of DML	606
11.4	A Hoare Logic for Synchronous Message Passing	608
11.4.1	Proofs from Assumptions and the Cooperation Test	608
11.4.2	A Proof System for DML	617
11.5	Soundness and Relative Completeness of this Hoare Logic	630
11.6	Technical Note: Modifications Towards Compositionality	638
11.6.1	Proof Outlines	638
11.6.2	The Proof System of Levin & Gries	642
11.6.3	Removing the Interference Freedom Test	645
11.6.4	Removing the Cooperation Test	648
	Part V: Layered Design	**653**
12	**Transformational Design and Hoare Logic**	**654**
12.1	Introduction and Overview	654
12.2	Structure of this Chapter	660
12.3	Syntax and Informal Meaning of SVL^{++} Programs	660
12.4	The Semantics of SVL^{++} Programs	662
12.5	Partial Orders and Temporal Logic	663
12.5.1	Action Ordering	670
12.6	The Communication-Closed-Layers Laws	676
12.6.1	CCL Laws for Shared Variables	679

12.6.2 CCL Law for Communication-Based Programs 684
12.7 The Two-Phase Commit Protocol 688
12.7.1 Derivation of a Layered Implementation 690
12.8 Assertion-Based Program Transformations 694
12.9 Loop Distribution 696
12.10 Set-partitioning Revisited 700
12.11 Historical Notes 704
Bibliography 710
Glossary of Symbols 747
Index 761

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.

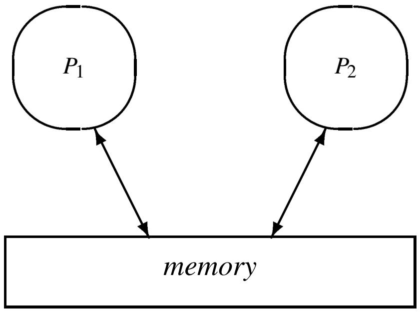

Fig. 1.1. Shared memory concept.

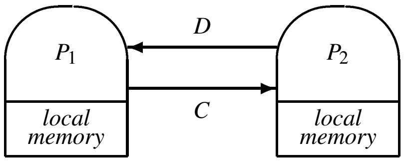

Fig. 1.2. Computer-communication network concept.

- *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*

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

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 communication, 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

$$
\begin{array}{lll}
P_1 \equiv \ell_0 : & \textbf{loop forever do} & \\
& \quad \ell_1 : & \textbf{noncritical section}; \\
& \quad \ell_2 : & (y_1, s) := (1, 1); \\
& \quad \ell_3 : & \textbf{wait}\ (y_2 = 0) \vee (s \neq 1); \\
& \quad \ell_4 : & \textbf{critical section}; \\
& \quad \ell_5 : & y_1 := 0 \\
& \textbf{od} &
\end{array}
$$

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.)

Program *Pet1* is too demanding in the sense that it forces the processor(s) executing this program to perform the assignments to y_i and to s within the same step. Programs which use these sorts of *multiple assignments* as atomic actions are said to be *of coarse granularity*. Most existing hardware systems cannot perform multiple assignments in one step. Consequently, we wish to refine this working program into a *finer-grained* program in which the assignments to y_i and to s will be performed in two separate assignments. There are two candidates to such a refinement. They are given by programs *Pet2* (solution 2) and *Pet3* (solution 3) respectively. Are both of these programs correct? Let us examine them each in turn.

Example 1.2 Let

$P_1 \equiv \ell_0 :$ **loop forever do**
- $\ell_1 :$ **noncritical section**;
- $\ell_2 :$ $s := 1$;
- $\ell_3 :$ $y_1 := 1$;
- $\ell_4 :$ **wait** $(y_2 = 0) \vee (s \neq 1)$;
- $\ell_5 :$ **critical section**;
- $\ell_6 :$ $y_1 := 0$

od

and

$P_2 \equiv m_0 :$ **loop forever do**
- $m_1 :$ **noncritical section**;
- $m_2 :$ $s := 2$;
- $m_3 :$ $y_2 := 1$;
- $m_4 :$ **wait** $(y_1 = 0) \vee (s \neq 2)$;
- $m_5 :$ **critical section**;
- $m_6 :$ $y_2 := 0$

od

and let

$$Pet2 \equiv s := 1; y_1 := 0; y_2 := 0; [P_1 \| P_2]. \qquad \square$$

Program *Pet2* is not correct. Consider the following computation, where states are presented as 5-tuples listing the current location of P_1, the current location of P_2, and the current values of s, y_1, and y_2.

$$\langle \ell_0, m_0, 1, 0, 0 \rangle, \quad \langle \ell_1, m_0, 1, 0, 0 \rangle, \quad \langle \ell_1, m_1, 1, 0, 0 \rangle, \quad \langle \ell_2, m_1, 1, 0, 0 \rangle,$$
$$\langle \ell_2, m_2, 1, 0, 0 \rangle, \quad \langle \ell_3, m_2, 1, 0, 0 \rangle, \quad \langle \ell_3, m_3, 2, 0, 0 \rangle, \quad \langle \ell_3, m_4, 2, 0, 1 \rangle,$$
$$\langle \ell_3, m_5, 2, 0, 1 \rangle, \quad \langle \ell_4, m_5, 2, 1, 1 \rangle, \quad \langle \ell_5, m_5, 2, 1, 1 \rangle, \quad \ldots$$

In this computation, P_2 could move from m_4 to m_5 because y_1 was 0 at the time, while P_1 could move from ℓ_4 to ℓ_5 because s was equal to 2 at the time.

Obviously, the last displayed state in this computation violates the requirement of mutual exclusion since both processes are executing in their critical sections at the same time. This shows that program *Pet2* is incorrect.

Next consider program *Pet3*.

Example 1.3 Let

$$
\begin{array}{lll}
P_1 \equiv \ell_0 : & \textbf{loop forever do} & \\
& \ell_1 : & \textbf{noncritical section}; \\
& \ell_2 : & y_1 := 1; \\
& \ell_3 : & s := 1; \\
& \ell_4 : & \textbf{wait } (y_2 = 0) \vee (s \neq 1); \\
& \ell_5 : & \textbf{critical section}; \\
& \ell_6 : & y_1 := 0 \\
\quad \textbf{od} & &
\end{array}
$$

and

$$
\begin{array}{lll}
P_2 \equiv m_0 : & \textbf{loop forever do} & \\
& m_1 : & \textbf{noncritical section}; \\
& m_2 : & y_2 := 1; \\
& m_3 : & s := 2; \\
& m_4 : & \textbf{wait } (y_1 = 0) \vee (s \neq 2); \\
& m_5 : & \textbf{critical section}; \\
& m_6 : & y_2 := 0 \\
\quad \textbf{od} & &
\end{array}
$$

and let

$$Pet3 \equiv s := 1; y_1 := 0; y_2 := 0; [P_1 \| P_2]. \qquad \square$$

For this program we are unable to produce a violating computation. The program can actually be proved correct. It is due to G.L. Peterson [Pet83]. A formal correctness proof of a variant of this algorithm is the subject of Exercise 3.4.[4]

In this discussion we have presented three programs that attempt to establish mutual exclusion. The programs are very similar and are certainly based on the same idea for coordinating the exclusion. Thus the intuition for all three is

[4] In his dissertation [Din99a], J. Dingel proves that the correctness of this algorithm does not depend on any atomicity or granularity assumptions; also, he refines the n-process version of the algorithm in a similar fashion as above.

basically sound. However, two of the three are correct while *Pet2* is incorrect. The differences between *Pet2* and *Pet3* are very small. This just shows that in studying even the simplest concurrent programs, intuition is very useful, but without formal methodologies we are lost.

1.4.3 Second Example: Concurrent Garbage Collection

In the previous example we have seen that when refining atomic actions by a sequence of other atomic actions the order in which the new actions are performed is crucial for the correctness of the refined version. In the following example we will observe the same phenomenon in the context of *concurrent garbage collection*. We have chosen this subject because the correctness of algorithms for concurrent garbage collection has been put in doubt ever since M. Woodger and N. Stenning found a deep logical bug in one of the first concurrent garbage collectors ever written, due to E.W. Dijkstra, L. Lamport and others. Since then several other algorithms for concurrent garbage collection have been published. Some of them have been formally proven correct, but others still suffer from bugs. A good indication of the complexity of developing correct concurrent garbage collection algorithms are the following words of Edsger W. Dijkstra:

> It had been surprisingly hard to find the published solution and justification [of this algorithm] ([DLM$^+$78]). We have fallen into nearly every logical trap possible. It was only too easy to design what looked – sometimes even for weeks and to many people – like a perfectly valid solution, until the effort to prove it correct revealed a (sometimes deep) bug. One firm conclusion, however, can be drawn: to believe that such solutions can be found without a very careful justification is optimism on the verge of foolishness.

Leslie Lamport's reaction to this bug goes even further [Lam99]:

> The lesson I learned from this is that behavioral proofs are unreliable and one should always use state-based [assertional] reasoning for concurrent algorithms – that is, using a proof based on invariance. All of my later work has been based on this belief.

The problem to be solved is the following. In systems using references or pointer variables, manipulation of these variables may cause certain parts of a dynamically changing data structure to become inaccessible from some designated root. For such systems the operating system has to perform the task of *garbage collection*, meaning identifying and recycling inaccessible nodes.

If we view the memory as an array of *n* homogeneous nodes, we can use the following abstract presentation of the garbage collection problem due to [DLM$^+$78]. We consider a directed graph of varying structure with a fixed number of nodes, where each node has a unique number $i \in \{1, \dots, \#nodes\}$

as identifier and at most two outgoing edges. In the graph there exists a fixed set of so-called *roots*. A node is called *reachable* if it is reachable via a directed path from one of these roots. Nonreachable nodes are called *garbage nodes*. The directed subgraph consisting of all reachable nodes and their outgoing edges represents the current *data structure*. Two extra roots are introduced: one root which points to the *list of free nodes*, i.e., nodes which can be used to modify the data structure, and a root called NIL, to which each node with less than two sons points with its free outgoing edges. The original program modifying the data structure is called the *mutator* in the context of garbage collection. The task of the so-called *collector* is to identify the set of garbage nodes, i.e., the set of nodes which are not reachable from any root, and to append them to the list of free nodes. In algorithms for *garbage-collection-on-the-fly*, the collector has to perform this task concurrently to the activity of a mutator. In this section we focus on such concurrent garbage-collection algorithms.

There are basically two main classes of algorithms. *Marking collectors* work in two phases; first they traverse the graph starting from the roots and mark all nodes they visit during the traversal (*marking phase*). In the second phase they append all non-marked nodes to the list of free nodes and remove all markings (*appending phase*).

In *reference counting* algorithms, in each node the number of references to this node is maintained. When the number of references to a node becomes zero the node is appended to the list of free nodes. A problem in reference counting algorithms is the proper handling of reference cycles between garbage nodes, which may cause the number of references to certain garbage nodes to never become zero.

The algorithms we look at belong to the first class. Thus the activities of the mutator and the collector can be described by repeated execution of:

- the **mutator**: redirecting an outgoing edge of a reachable node towards an already reachable one, and
- the **collector**: marking all reachable nodes, appending all non-marked nodes to the free list and removing all markings.

The correctness criteria which have to be satisfied by the combination of mutator and collector can now be formulated as:

- There are no changes made by the collector in the data structure, except for appending garbage nodes to the free list.
- Every garbage node will eventually be appended by the collector to the list of free nodes.

The first version of our garbage collector works as follows. At the beginning of the marking phase all nodes in the graph are white; several marking sweeps through the memory according to a certain visiting policy for the nodes will blacken all reachable nodes. If no further nodes are marked during a certain sweep, the collector assumes it has marked all reachable nodes and thus starts the appending phase. All remaining white nodes (i.e., garbage nodes, because they are not reachable) are appended to the free list, whereas black nodes are marked white again for the next marking cycle. In the presentation below $< a >$ denotes the atomic version of an arbitrary non-atomic statement a and $\{ \dots \}$ a comment.

Example 1.4 (First approximation) Let

```
C1 ≡  {marking phase}
      begin {all nodes are white}
         <blacken all roots>;
         nnb := #roots; {nnb = "new number of black nodes"}
         repeat {marking loop}
            onb := nnb; {onb = "previous number of black nodes"}
            for i from 1 to #nodes do
               if color of node i = black then
                  <blacken the successors of node i>
               fi
            od;
            < nnb := #black nodes in the graph>
         until onb = nnb
      end;
      {collecting and appending phase}
      begin
         for i from 1 to #nodes do
            if color of node i = white then
               <append node i to the free list>
            else <whiten node i >
            fi
         od {there are no more nodes that have been blackened
            during the marking phase}
      end.
```
□

In all examples of possible runs which are illustrated in this section the numbers identifying the nodes increase from top to bottom. This enumeration imposes a certain visiting policy by the algorithm for the nodes. By convention node number 1 denotes the root of the list of free nodes. Roots of the directed graph are represented by squares and are assumed to have the lowest numbers. We have omitted the special root NIL, as well as all references to it.

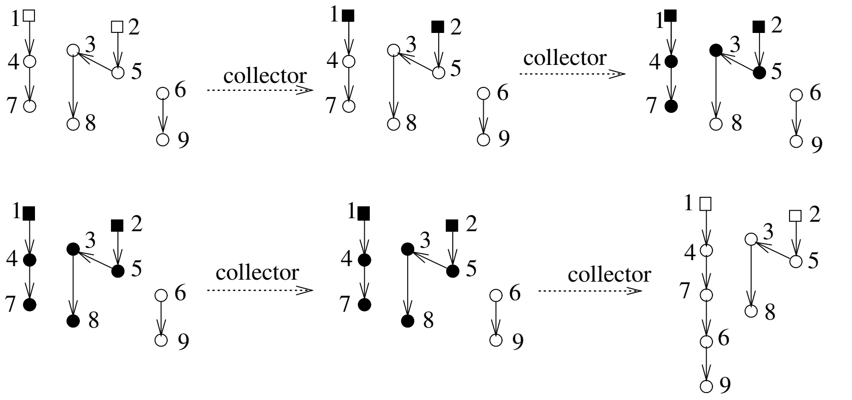

Fig. 1.3. Marking and collecting: The roots are represented by squares; node number 1 denotes the root of the list of free nodes.

Figure 1.3 demonstrates how the algorithm works. At first the collector marks all roots black; one sweep through the marking loop gives the situation shown in the third graph (notice that node number 8 has not been marked in the first sweep, although this node is reachable, because of the visiting policy). The second sweep blackens in this example all reachable nodes. Because the collector does not note a change in the number of black nodes after the third sweep, it starts collecting unmarked nodes, thereby appending node 6 and 9 to the list of free nodes, and removes all markings.

To understand the behaviour of the mutator, we indicate in Figure 1.4 edges to NIL nodes by broken lines. The operations "adding an edge" and "removing an edge" are special cases of "redirecting an edge" where the edge in question is targeted to NIL at the start and at the end of the operation, respectively.

In the example (Figure 1.3) above we only looked at the activities of the collector in isolation. Does this collector work properly if concurrently combined with an arbitrary mutator? Look at the perfectly valid mutator behaviour shown in Figure 1.5.

In Figure 1.5 node 5 has not been marked during the first marking sweep; the collector finds that three nodes are already marked; assume the collector has started the second sweep and is just going to inspect node 3; now the mutator may add edge $2 \longrightarrow 5$ and then as its next action remove edge $3 \longrightarrow 5$; then node 5 remains unmarked in the second sweep, too; the collector finishes the

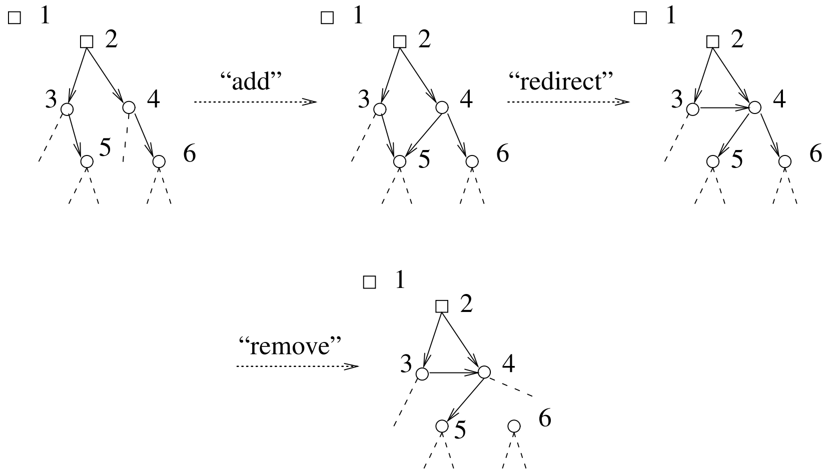

Fig. 1.4. Mutator behaviour: Edges to the NIL node are indicated by broken lines. By repeatedly redirecting an edge, the mutator causes node number 6 to become garbage.

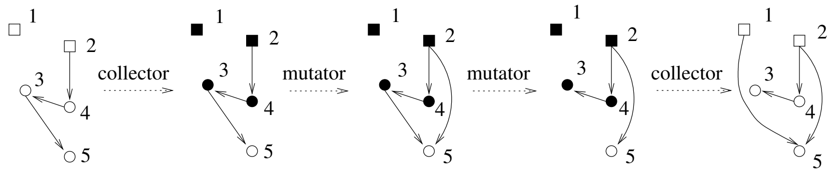

Fig. 1.5. Uncooperative mutator behaviour: By lack of cooperation between collector and mutator, a non-garbage node is appended to the free list.

marking loop and finds the number of black nodes unchanged and thus collects *incorrectly* the unmarked node 5 as garbage.

This incorrectness arises because the mutator made the unmarked node 5 inaccessible to the collector by redirecting edges to this node. It is easy to see that, whatever visiting policy is used by the collector to inspect nodes, there always exists a certain data structure and a certain mutator which makes reachable nodes unreachable for the collector by redirecting edges and thus causes malfunctioning of the garbage collector. This shows *that the mutator itself has to contribute to the garbage collection task.* In order to circumvent problems arising by redirecting edges from and to reachable nodes, we change the action of the mutator (for two arbitrary reachable nodes in the graph) to:

M_1 : $<$ blacken the new target and
redirect the outgoing edge to the new target $>$

Does the collector concurrently combined with the modified mutator work properly, i.e., is only garbage appended to the free list, and will all garbage eventually be collected?

We give an informal argument to "show" that the algorithm is indeed correct. If we can prove that, after the marking phase has terminated, all white nodes are garbage, we can conclude that only garbage is appended to the free list. Since it is clear that a nonreachable node, which is still marked black at the end of the marking loop, will be marked white at the end of the marking phase, and hence be collected in the next phase, we can conclude the correctness of the algorithm. So we will concentrate on the former. In order to terminate, the algorithm has to find the same number of marked nodes during two successive marking sweeps. This implies (because no black nodes are turned white again), that the collector did not mark any new node during the last sweep. Now we give an invariant property (a property which holds before and after execution of any action of the algorithm), which implies that all reachable nodes are marked after termination of the marking loop, and thus all unmarked nodes are garbage.

Take as invariant: *whenever the collector arrives at a certain point in the marking loop and has not marked any new node during this sweep, then there is no already inspected black node in the graph which was already black at the moment of inspection and which has an edge to an unmarked white node.*[5]

Readers are encouraged to convince themselves that this property is invariant under all possible steps of the mutator and the collector (Exercise 1.1). After termination of the last sweep, which did not mark any new node, we conclude from the invariant and the fact that the roots are marked, that *all* reachable nodes are marked. This has been shown formally for a similar garbage collector in [BA82].

So we have a first correct version of an algorithm for garbage collection with a very coarse granularity of the atomic actions. We have assumed that actions like "blacken all roots" and "blacken target and redirect edge" are performed *atomically* which requires a high synchronisation effort between collector and mutator and results in low efficiency of the algorithm. In order to find a more efficient solution, where the only form of synchronisation concerns access to shared variables, we refine the coarse atomic actions of M_1 and C_1 by sequences of atomic actions. A possible refinement is the following:

Example 1.5 (Second approximation) Let

[5] Node i is considered to have been inspected after "**if** color of node i = black **then** <blacken the successors of node i> **fi**" has been executed.

```
M2 ≡    <blacken the new target>;
        <redirect edge to the new target>
```

and

```
C2 ≡    {marking phase}
        begin {all nodes are white}
            for i from 1 to #roots do
                <blacken node i >;
            od;
            nnb := #roots; {nnb = "new number of black nodes"}
            repeat {marking loop}
                onb := nnb; {onb = "previous number of black nodes"}
                for i from 1 to #nodes do
                    if color of node i = black then
                        <blacken left successor of node i>;
                        <blacken right successor of node i>
                    fi
                od;
                nnb := 0;
                for i from 1 to #nodes do
                    if color of node i = black then
                        nnb := nnb + 1
                    fi
                od
            until onb = nnb
        end;
        {collecting and appending phase}
        begin
            for i from 1 to #nodes do
                if color of node i = white then
                    <append node i to the free list>
                else <whiten node i >
                fi
            od {there are no more nodes that have been blackened
                during the marking phase}
        end.                                                    □
```

We have derived a "straightforward" fine-grained solution from the correct coarse-grained solution. But no one will succeed in proving this new version correct. Even such famous researchers as E.W. Dijkstra and L. Lamport were

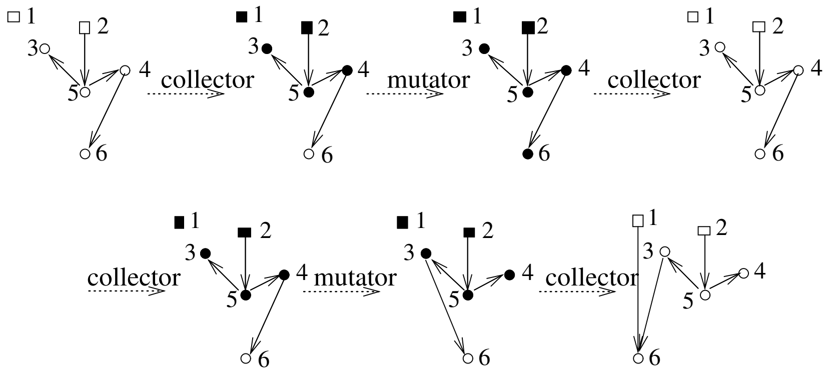

Fig. 1.6. Stenning & Woodger's bug: The mutator marks node number 6 but delays redirecting the edge from node number 3 to its new target, node number 6. Consequently, that node is marked white in the next collector actions, and upon redirecting that edge it remains white and is collected.

convinced of the correctness of the solution, till N. Stenning and M. Woodger found a deep logical bug in this first version of [DLM$^+$78].

Figure 1.6 shows the situation where a non-garbage node is erroneously appended to the free list. After the first marking sweep of the collector the mutator wants to establish a reference from node 3 to node 6 and thus blackens node 6; assume that the mutator *delays* establishing this new reference; then the collector will find during the following marking sweep that the number of black nodes did not change and removes all markings; the first sweep of the next marking phase now ends with nodes 1 to 5 marked black and node 6 unmarked (situation 5) and the number of black nodes equal to 5; during the second sweep just before the collector will colour node 6 as a successor of the black node 4, the mutator establishes first the reference from node 3 to 6 and then removes the reference from node 4 to 6; thus node 6 remains unmarked and is thus incorrectly collected as garbage, because the collector finds the number of black nodes unchanged!

In order to fix this bug we could try to swap the actions of the mutator. But what happens, if the colouring of the new target by the mutator is delayed? Do we get other unexpected behaviour? And if not, have we now really considered all possible situations that might occur, when the mutator and the collector act concurrently? A correct version of this algorithm is sketched in Exercise 1.2. It has been formally proved correct in [Gri77]. In Section 10.5 a general strategy is presented for generating correctness proofs for such complex algorithms, using concurrent garbage collection as an example. First an abstract version of a concurrent garbage collector is proved correct, and then a correctness proof is given for a concrete version. The general problem of how to prove correctness of a concurrent algorithm when its atomic actions are refined has been solved by P. Gribomont in [Gri96].

The algorithms that we have investigated above are only partly distributed: the collector and the mutator work in parallel, but the whole data structure resides in one shared memory. The task of garbage collection becomes even more complicated than it is already, if the data structure is *distributed* over a network of nodes, where each node represents a processor. Then we have to deal with several mutators and collectors (possibly spawning new mutator and collector processes), that all work in parallel on a distributed structure. As shown in [Rud90], almost all of the distributed garbage collector algorithms proposed before 1990 contain bugs. In the opinion of M. Rudalics, *it is even doubtful whether some of these incorrect algorithms can be corrected at all!* The relation between garbage collection algorithms and algorithms in other areas of distributed computing, as, e.g., termination detection, global (virtual) time, distributed snapshots, and construction of spanning trees, gives rise to the suspicion, that also in these areas many incorrect published algorithms exist.[6]

We hope that the presented (non-)solutions for the above on-the-fly garbage collection and the incorrect algorithms for distributed garbage collection convince the reader of the truth of the cited words of Dijkstra: *"To believe that* [correct] *solutions can be found without a very careful justification is optimism on the verge of foolishness."*

To convince the remaining believers in "programming using common sense" that their position is in general untenable, we present in Section 1.4.4 a rather complicated mutual exclusion algorithm due to Szymanski and attempt to justify the correctness of this algorithm using behavioural reasoning. It is our belief that this exercise gives the final deathblow to that position. It is just impossible to survey the informally given arguments for the correctness of that algorithm. Although in this case the final version of the algorithm is correct – a formal proof is given in Section 3.5.2 – we believe that it is not possible to be convinced of this using behavioural reasoning [MP90, MP91c].

The same conclusion is reached by Leslie Lamport on the basis of his experience with [DLM$^+$78], in the following quotation [Lam99]:

I received an early version of this paper [about concurrent garbage collection] from Dijkstra (probably as an EWD[7]) and I made a couple of suggestions. Dijkstra incor-

[6] See Section 8.10 of [JL96] for a thorough discussion of the history of concurrent garbage collection and the errors found in these algorithms.

[7] At an early stage Edsger W. Dijkstra started numbering all his professional writings, labelling them by "EWD", followed by their number.

porated them and, rather generously, added me to the list of authors. He submitted the paper to *CACM*. The next I heard about it was when I received a copy of a letter from Dijkstra to the editor withdrawing the paper. The letter said that someone had found an error in the algorithm, but gave no indication of what the error was. Since Dijkstra's proof was so convincing, I figured that it must be a trivial error that could easily be corrected.

I had fairly recently developed my version of what was to become known as the Owicki–Gries method. So, I decided to write a proof using this method, thinking that I would then find and correct the error. In about 15 minutes, the proof method led me to the error. To my surprise, it was a serious error.

I had a hunch that the algorithm could be fixed by changing the order in which two operations were performed. But I had no good reason to believe that would work. Indeed, I could see no simple informal argument to show that it worked. However, I decided to go ahead and try to write a formal correctness proof. It took me about two days of solid work, but I constructed the proof. When I was through, I was convinced that the algorithm was now correct, but I had no intuitive understanding of why it worked.

In the meantime, Dijkstra figured out that the algorithm could be fixed by interchanging two other operations, and wrote the same kind of behavioural proof as before. His fix produced an arguably more efficient algorithm, so that's the version we used. I sketched an assertional (Owicki–Gries style) proof of that algorithm. Given the evidence of the unreliability of his style of proof, I tried to get Dijkstra to agree to a rigorous assertional proof. He was unwilling to do that, though he did agree to make his proof somewhat more assertional and less operational.

Gries later published an Owicki–Gries style proof of the algorithm that was essentially the same as the one I had sketched. He simplified things a bit by combining two atomic operations into one. He mentioned that in a footnote, but the *CACM* failed to print the footnote. (However, they did print the footnote number in the text.)

The lesson I learned from this is that behavioral proofs are unreliable and one should always use state-based [assertional] *reasoning for concurrent algorithms – that is, using a proof based on invariance.* All of my later work, leading up to TLA, has been based on this belief.

We conclude this discussion with another quotation from Dijkstra's work [Dij75b]:

The first moral of this story is *never* to believe the correctness of a solution without a proof for it. The second moral of this story is *never* to forget the first one. The third moral is that for the design of multiprocessor installations we cannot rely on the traditional approach of the optimistic engineer, who, when the design looks reasonable, puts it together to see if it works.

With VLSI-technology existing it seems unavoidable that multiprocessor installations will be built, and, people being as they are, it seems equally unavoidable that many of them will be put together by the aforementioned optimistic engineer. I shudder at the thought of all the new bugs: they will only delight the Devil. Am I too pessimistic? Nobody knows the trouble I have seen ....

1.4.4 Third Example: Szymanski's Mutual Exclusion Algorithm

In their paper [MP91c], Zohar Manna and Amir Pnueli prove a rather complicated mutual exclusion algorithm correct, which is due to B.K. Szymanski [Szy88]. This algorithm represented at the beginning of the 1990s the state of the art in mutual exclusion algorithms for distributed processes. In this algorithm each shared variable, although read in all processes, is updated in only one of them; moreover it enjoys the property of so-called linear delay, expressing that the period of waiting associated with entering critical sections and their "waiting rooms" is linearly bounded in the number of processes.

Manna and Pnueli write in [MP91c]:

> Although the algorithm appears to be quite simple and innocuous, the only way we could convince ourselves of its correctness was to construct the formal proof outlined in this paper. Szymanski presented an informal proof, which is as convincing as informal proofs can be. In fact, our formal proof derives its main ideas from a formalisation of his informal arguments. However, if the question of correctness is crucial, such as having to decide whether to include this algorithm as a contention-resolving component in a hardware chip, we see no way but to carry out a formal verification.

The interested reader is advised to read their paper, also because it contains a quick and effective introduction to temporal logic "for the practicing verifier".

Below we present the algorithm in a style dubbed "derivation of a correct algorithm through incorrect intermediate approximations". This style can be made formal, as argued in [CKdR92], and is particularly suitable for explaining mutual exclusion algorithms. In fact, Dijkstra was the first one to pioneer this style in his presentation of the first-ever published mutual exclusion algorithm [Dij65b, Dij68a] which was based on the ideas of T.J. Dekker.

But first we present, as a deliberate shock, Szymanski's complete program. Just study it for an hour or so to see whether you can make head or tail of it! Its full text is given below.

Example 1.6 (Final version of Szymanski's algorithm) Let

$P_i \equiv l_0$: **loop forever do**
    l_1 : **noncritical section**;
    l_2 : $\mathit{flag}[i] := 1$;
    l_3 : **wait** $\forall j : 0 \leq j < n.(\mathit{flag}[j] < 3)$;
    l_4 : $\mathit{flag}[i] := 3$;
    l_5 : **if** $\exists j : 0 \leq j < n.(\mathit{flag}[j] = 1)$ **then**
        l_6 : $\mathit{flag}[i] := 2$;
        l_7 : **wait** $\exists j : 0 \leq j < n.(\mathit{flag}[j] = 4)$;
      fi;

l_8 : $flag[i] := 4$;
l_9 : **wait** $\forall j : 0 \leq j < i.(flag[j] < 2)$;
l_{10} : **critical section**;
l_{11} : **wait** $\forall j : i < j < n.(flag[j] < 2 \vee flag[j] > 3)$;
l_{12} : $flag[i] := 0$

od

and let

$$\textbf{mutex5} \equiv flag := 0; [P_0 \| P_1 \| \ldots \| P_{n-1}]. \qquad \square$$

Before we start explaining this algorithm, we quote Szymanski's original explanation in [Szy90]:

The idea behind the algorithm is simple. The prologue section simulates a waiting room with a door. All processes requesting entry to the critical section at roughly the same time gather first in the waiting room. Then, when there are no more processes requesting entry, processes inside the waiting room shut the door and move to the exit from the waiting room. From there, one by one, they enter their critical sections in the order of their numbering. Any process requesting access to its critical section at that time has to wait in the initial part of the prologue section (at the entry to the waiting room).

The door to the waiting room is initially opened. The door is closed when a process inside the waiting room does not see any new processes requesting entry. The door is opened again when the last process inside the waiting room leaves the exit section of the algorithm.

Our first approximation towards explaining the algorithm starts below.

Example 1.7 (First approximation) Let

$P_i \equiv l_0$: **loop forever do**
l_1 : **noncritical section**;
l_2 : $flag[i] := 1$;
l_3 : **wait** $\forall j : 0 \leq j < n \wedge j \neq i.(flag[j] = 0)$;
l_4 : **critical section**;
l_5 : $flag[i] := 0$

od

and let

$$\textbf{mutex1} \equiv flag := 0; [P_0 \| P_1 \| \ldots \| P_{n-1}]. \qquad \square$$

Observe that each process P_i has a variable $flag[i]$, which is read by all other processes, and changed only by P_i. The assignment $flag := 0$ is an abbreviation of $flag[0] := 0; \ldots ; flag[n-1] := 0$.

This solution satisfies the mutual exclusion property for the following reason. When a process P_i is in its **critical section** then its *flag*$[i]$ equals 1. Because of the **wait** statement at l_3 a process P_j with $j \neq i$ can only enter its **critical section** when all other processes have their *flag*s set to 0, but we know *flag*$[i]$ equals to 1 so P_j cannot enter its **critical section**.

However, **mutex1** suffers from danger of deadlock: if all processes P_i execute their l_2 instructions simultaneously then all the *flag*$[i]$s are set equal to 1. Then no process can pass l_3, because there is no process with a *flag* equal to 0, and now and in the future all processes remain suspended at l_3.

A possible solution for this problem might be that the **wait** statement at l_3 can also be passed when $\forall j : 0 \leq j < n.\mathit{flag}[j] = 0 \vee \mathit{flag}[j] = 1$. But then the **wait** statement could always be passed because *flag* has only value 0 or 1. This can be solved as follows: a process which passes the **wait** statement at l_3 sets its *flag* to 4 before it enters the **critical section**. Then all processes trying to pass l_3 afterwards are suspended, until that one process has executed its **critical section** and has set its *flag* to 0. The passing of l_3 and the setting of the *flag* to 4, however, is not atomic so several other processes could pass l_3 before that process sets its *flag* to 4. So the mutual exclusion property is violated because more than one process can enter its **critical section**.

The solution to this problem is that from the processes that have passed l_3, the process with the smallest process number will first enter its **critical section**, followed by the process with the smallest process number of the remaining processes, and so on. The following **wait** statement will do this job: **wait** $\forall j : 0 \leq j < i.(\mathit{flag}[j] = 0 \vee \mathit{flag}[j] = 1)$. This statement is placed in between the assignment of *flag*$[i]$ to 4 and the **critical section**. A process can only pass this statement when all processes with a smaller process number are non-critical. So we obtain the following program **mutex2**:

Example 1.8 (Second approximation) Let

$P_i \equiv l_0$: **loop forever do**
- l_1 : **noncritical section**;
- l_2 : $\mathit{flag}[i] := 1$;
- l_3 : **wait** $\forall j : 0 \leq j < n.(\mathit{flag}[j] = 0 \vee \mathit{flag}[j] = 1)$;
- l_4 : $\mathit{flag}[i] := 4$;
- l_5 : **wait** $\forall j : 0 \leq j < i.(\mathit{flag}[j] = 0 \vee \mathit{flag}[j] = 1)$;
- l_6 : **critical section**;
- l_7 : $\mathit{flag}[i] := 0$

od

and let

$$\mathbf{mutex2} \equiv \mathit{flag} := 0; [P_0 \| P_1 \| \dots \| P_{n-1}]. \qquad \square$$

Unfortunately **mutex2** does not satisfy the mutual exclusion property. Let $n = 2$, $l_k \to l_{k+1}$ indicate a move of P_0 and $l'_k \to l'_{k+1}$ indicate a move of P_1. Consider the following computation, where states are presented as 4-tuples listing the current location l_k of P_0, the current location l'_k of P_1, and the current values of *flag*[0] and *flag*[1]:

$$\begin{array}{lllllllll}
\langle l_0, l'_0, 0, 0\rangle & \to & \langle l_1, l'_0, 0, 0\rangle & \to & \langle l_1, l'_1, 0, 0\rangle & \to & \langle l_2, l'_1, 0, 0\rangle & \to \\
\langle l_2, l'_2, 0, 0\rangle & \to & \langle l_3, l'_2, 1, 0\rangle & \to & \langle l_3, l'_3, 1, 1\rangle & \to & \langle l_4, l'_3, 1, 1\rangle & \to \\
\langle l_4, l'_4, 1, 1\rangle & \to & \langle l_4, l'_5, 1, 4\rangle & \to & \langle l_4, l'_6, 1, 4\rangle & \to & \langle l_5, l'_6, 4, 4\rangle & \to \\
\langle l_6, l'_6, 4, 4\rangle & \to & \dots
\end{array}$$

Since l_6 and l'_6 label critical sections, the mutual exclusion property is not satisfied.

How can we avoid this sequence being generated? We must force process P_0 to set its *flag* to 4 *before* process P_1 inspects the **wait** statement at l'_5. Or, in general, all the processes that have passed l_3 must wait with the inspection of the **wait** statement at l_5 until processes, which passed l_3 and have a lower index than the inspecting process, have set their *flag* to 4. We know that the value of the *flag* equals 1 before it is set to 4. By inserting the following test in front of label l_5 below we can check whether there are processes that have not yet set their *flag* to 4 by inserting the test: **if** $\exists j : 0 \le j < n.(\mathit{flag}[j] = 1)$ **then**. But this test does not solve the problem when there is a process suspended at l_3. We can let such processes pass l_3 if we move the test in front of the assignment of *flag*[i] to 4. We assign the value 2 to the *flag* of the process for which this test is true. And also we change l_3 into **wait** $\forall j : 0 \le j < n.(\mathit{flag}[j] = 0 \vee \mathit{flag}[j] = 1 \vee \mathit{flag}[j] = 2)$. So processes can slip through the **wait** statement at l_3 if there is not yet a process that has assigned its *flag* to 4. But before this is possible the process that has assigned 2 to its *flag* must wait until there is a process that has passed the test between l_3 and l_4, and set its *flag* to 4. This results in **mutex3**, which is as follows:

Example 1.9 (Third approximation) Let

$P_i \equiv l_0$: **loop forever do**
  l_1 : **noncritical section**;
  l_2 : $\mathit{flag}[i] := 1$;
  l_3 : **wait** $\forall j : 0 \le j < n.(\mathit{flag}[j] = 0 \vee \mathit{flag}[j] = 1 \vee \mathit{flag}[j] = 2)$;
  l_4 : **if** $\exists j : 0 \le j < n.(\mathit{flag}[j] = 1)$ **then**
    l_5 : $\mathit{flag}[i] := 2$;

l_6 : **wait** $\exists j : 0 \leq j < n.(flag[j] = 4)$
fi;
l_7 : $flag[i] := 4$;
l_8 : **wait** $\forall j : 0 \leq j < i.(flag[j] = 0 \vee flag[j] = 1)$;
l_9 : **critical section**;
l_{10} : $flag[i] := 0$
od

and let

$$\textbf{mutex3} \equiv flag := 0; [P_0 \| P_1 \| \ldots \| P_{n-1}]. \qquad \square$$

This approximation has a terrible fault. All the processes passing l_3 will go to l_5 because for these processes their own *flag* equals to 1. And thus all these processes get suspended at l_6 because there is no process that can set its *flag* to 4. This is a deadlock situation because now no process can enter its **critical section**.

How can we solve this problem? This is easy, just put between l_3 and l_4 the following statement: $flag[i] := 3$. The sequence described above cannot be generated because the process that checks the test at l_4 in the third approximation as the last one will go to l_7 and thus also awake the processes that are suspended at l_6, since such a process will eventually set its *flag* to 4. It can be proved that the resulting program **mutex4** satisfies the mutual exclusion property and does not deadlock.

Example 1.10 (Fourth approximation) Let

$P_i \equiv l_0$: **loop forever do**
l_1 : **noncritical section**;
l_2 : $flag[i] := 1$;
l_3 : **wait** $\forall j : 0 \leq j < n.(flag[j] = 0 \vee flag[j] = 1 \vee flag[j] = 2)$;
l_4 : $flag[i] := 3$;
l_5 : **if** $\exists j : 0 \leq j < n.(flag[j] = 1)$ **then**
l_6 : $flag[i] := 2$;
l_7 : **wait** $\exists j : 0 \leq j < n.(flag[j] = 4)$
fi;
l_8 : $flag[i] := 4$;
l_9 : **wait** $\forall j : 0 \leq j < i.(flag[j] = 0 \vee flag[j] = 1)$;
l_{10} : **critical section**;
l_{11} : $flag[i] := 0$
od

and let

$$\mathbf{mutex4} \equiv \mathit{flag} := 0; [P_0 \| P_1 \| \dots \| P_{n-1}]. \qquad \square$$

Program **mutex4**, however, does not satisfy the linear delay property. This property expresses that the time that a process waits in program segment labelled by l_4 to l_9 is linear in the number of processes. Look for instance at the following sequence:

$$\begin{array}{lllllllll}
\langle l_0, l'_0, 0, 0\rangle & \rightarrow & \langle l_1, l'_0, 0, 0\rangle & \rightarrow & \langle l_1, l'_1, 0, 0\rangle & \rightarrow & \langle l_2, l'_1, 0, 0\rangle & \rightarrow \\
\langle l_2, l'_2, 0, 0\rangle & \rightarrow & \langle l_3, l'_2, 1, 0\rangle & \rightarrow & \langle l_3, l'_3, 1, 1\rangle & \rightarrow & \langle l_4, l'_3, 1, 1\rangle & \rightarrow \\
\langle l_4, l'_4, 1, 1\rangle & \rightarrow & \langle l_4, l'_5, 1, 3\rangle & \rightarrow & \langle l_4, l'_6, 1, 3\rangle & \rightarrow & \boxed{\langle l_4, l'_7, 1, 2\rangle} & \rightarrow \\
\langle l_5, l'_7, 3, 2\rangle & \rightarrow & \langle l_8, l'_7, 3, 2\rangle & \rightarrow & \langle l_9, l'_7, 4, 2\rangle & \rightarrow & \langle l_{10}, l'_7, 4, 2\rangle & \rightarrow \\
\langle l_{11}, l'_7, 4, 2\rangle & \rightarrow & \langle l_1, l'_7, 0, 2\rangle & \rightarrow & \langle l_2, l'_7, 0, 2\rangle & \rightarrow & \langle l_3, l'_7, 1, 2\rangle & \rightarrow \\
\boxed{\langle l_4, l'_7, 1, 2\rangle} & \rightarrow & \dots
\end{array}$$

Since state $\langle l_4, l'_7, 1, 2\rangle$ occurred earlier in this sequence, P_1 can be infinitely often bypassed by P_0, i.e., P_1 can possibly be starved, i.e., wait forever! Hence, the waiting time of P_1 is not linear.

What can we do about this situation? We must suspend process P_0 after l_{10} and before l_{11}, because then P_1 is forced to pass l_7, since the *flag* of P_0 is then still 4 and P_0 has already executed its **critical section**. How long should we suspend P_0? Until P_1 has passed l_7 and set its *flag* to 4, because then process P_0 can set its *flag* to 0 without suspending process P_1 at l_9 (P_0 has to be resumed, otherwise P_1 gets suspended at l_9 since the *flag* of P_0 is still 4). Process P_1 has to set its *flag* to 4 because then P_0 can also not reenter its **critical section** since it gets suspended at l_3.

In general this means that a process has to wait at l_{11} until all processes that are suspended at l_7 have passed l_7 and have set their *flag* to 4. This can be done by inserting the statement: **wait** $\forall j : i < j < n.(\mathit{flag}[j] < 2 \vee \mathit{flag}[j] > 3)$. We must have $i < j < n$ because the processes that are between l_4 and l_9 will enter their **critical section** in increasing order of their process numbers. So process P_i has to be suspended until all processes that are between l_4 and l_9 (these are processes with a process number higher than i) have set their *flag* to 1 or 4. Notice that we have to take processes P_i with $\mathit{flag}[i] = 1$ into account, because such processes may pass l_3. This culminates in the final program **mutex5**:

Example 1.11 (Fifth and final approximation) Let

$P_i \equiv l_0$: **loop forever do**
    l_1 : **noncritical section**;
    l_2 : $\mathit{flag}[i] := 1$;

l_3 : **wait** $\forall j : 0 \leq j < n.(flag[j] < 3)$;
l_4 : $flag[i] := 3$;
l_5 : **if** $\exists j : 0 \leq j < n.(flag[j] = 1)$ **then**
  l_6 : $flag[i] := 2$;
  l_7 : **wait** $\exists j : 0 \leq j < n.(flag[j] = 4)$
fi;
l_8 : $flag[i] := 4$;
l_9 : **wait** $\forall j : 0 \leq j < i.(flag[j] < 2)$;
l_{10}: **critical section**;
l_{11}: **wait** $\forall j : i < j < n.(flag[j] < 2 \vee flag[j] > 3)$;
l_{12}: $flag[i] := 0$
od

and let

$$\textbf{mutex5} \equiv flag := 0; [P_0 \| P_1 \| \ldots \| P_{n-1}]. \qquad \square$$

This program satisfies the property of mutual exclusion, freedom from deadlock, and the linear delay property provided the tests at l_3, l_5, l_7, l_9 and l_{11} are tested as **atomic** conditions, i.e., these tests must be implemented as one atomic action. Look for instance at the following sequence in which this is not the case. We assume that process P_0 does not test l_3 as an atomic statement. Suppose we have three processes P_0, P_1 and P_2. First P_1 executes l_2, next P_2 executes l_2, then P_1 passes l_3, after which P_2 passes l_3 and subsequently executes l_4 and finds the test at l_5 equal to true. Next it executes l_6 and then gets suspended at l_7. Then P_1 executes l_4, finds the test at l_5 equal to false, passes thus to l_8 and subsequently executes l_8. Then it passes l_9, and executes its **critical section** and next gets suspended at l_{11}. Then P_0 executes l_2 and checks (at l_3) if the *flag* of process P_2 is less than 3, which is true because P_2 is suspended at l_7, thus its *flag* equals 2. After this check P_2 passes l_7, because the *flag* of P_1 equals 4, and then P_2 executes l_8. Now P_1 passes l_{11}, because the *flag* of P_2 equals 4, and then P_1 executes l_{12}. Then P_2 passes l_9 (the *flag* of P_0 is still 1 and the *flag* of P_1 equals 0). So P_2 is now in its **critical section**. Then P_0 checks the *flag* of P_1 at l_3, finds its value to be 0, then checks its own *flag* (which is 1) and therefore passes to l_4. Next P_0 executes l_4, passes from l_5 to l_8, because the *flag* of P_2 equals 4 and that of P_1 equals 0. Then P_0 executes l_8 and passes l_9 because P_0 can always pass l_9. Therefore P_0 also executes its **critical section**. (We owe this scenario to [MP91c].) So the mutual exclusion property is violated. In [MP90] it is proved that when the *flag*-variables are checked in ascending order of their indices the algorithm is still correct.

We have obtained the final version of Szymanski's algorithm by observing undesirable computations in its previous approximations and by correcting these approximations through elimination of those computations. But how does one guarantee, without giving a formal correctness proof, that version 5 does not also display some undesirable computation? How long is one willing to inspect version 5 before becoming convinced that this version is correct? And, if one succeeds at all in giving a correctness proof based on behavioural reasoning, who guarantees that no subtle error has crept in, only waiting to be revealed by a highly complex scenario, which demonstrates that its alleged properties are not satisfied? This applies especially when considering a version of Szymanski's algorithm in which the above-mentioned tests at l_3, etc., are no longer atomic, and flag variables are checked in ascending order of their indices!

We conclude that there is no alternative but to give formal correctness proofs of such algorithms based on assertional reasoning. For Szymanski's algorithm such a proof is given in Section 3.5.2, for a version in which tests are atomic.

1.4.5 Conclusion: Answers to Two Central Questions

The Stenning & Woodger bug in one of the first algorithms for concurrent garbage collection, see Figure 1.6, was a historical landmark. For it convinced a number of people in 1976, including the designers of that algorithm – amongst whom were some of the most brilliant programmers and most eloquent advocates of their time of the need for program verification, that the *informal* style of proof, which was used in the presentation of the algorithm [DLM$^+$78], was no longer sufficient to guarantee correctness. David Gries claims that if one had used the *formal* method applied in [Gri77] such a bug would have been discovered. Leslie Lamport found a bug in this algorithm using state-based assertional reasoning, when trying to give a proof of its correctness based on invariance properties.

We conclude that verification is inevitable, indeed, for the development of correct complex software for concurrent algorithms, and that one needs to give formal proofs when software is really complex. This answers the first two questions raised in Section 1.1.

1.4.6 Hand-Checked or Machine-Supported Verification?

So, in 1976 Stenning & Woodger's bug convinced Edsger W. Dijkstra and others that informal proofs are not convincing. Next, in 1993, even the value of *formal* proofs was put in doubt when these were *hand checked*. John Rushby and others claim [ORSvH93] that the formal proofs of all safety-critical applications they investigated were wrong. So the question becomes: Is *hand-checked* formal verification enough, or do we need *machine*-supported verification?

In view of the fact that machine-supported verification is very time consuming, Leslie Lamport and his coauthors argue in [ADH+99] that a disciplined form of hand-checked formal verification is sufficient to expose errors in a 2000-line industrial cache protocol. This style of verification by hand has been developed by Lamport and his colleagues in a series of papers, see, e.g. [AL91, AL93, AL95].

A practical argument in favour of machine-supported verification applies when correctness proofs become long and tedious, as is the case for some of the real-time algorithms discussed in the companion volume. Then, on account of their sheer multitude one does not trust one's own previous proof steps anymore *when they are merely checked by hand*, and, instead of progressing confidently with one's proof, one tends to spend more and more time on checking ever increasing lists of rather simple proof steps over and over again. When these steps are verified using a computer, this phenomenon disappears – provided one trusts the proof tool being used to a sufficient degree. This trust is only justified in case of a really mature proof tool, which has been debugged over a long period by many persons.

Especially since INTEL lost an estimated 500 million US dollars due to an error in its Pentium chip, machine-supported verification has acquired higher industrial priority. This is reported, e.g., by J Strother Moore and his colleagues in his verification of small crucial parts of an AMD chip using the ACL2 tool [KM94, MLK98], by Ed Clarke and his colleagues in their verification of crucial hardware components using algorithmic verification techniques such as model checking [CE82, QS81, QS82, CGP99], and, recently, by Alain Deutsch using machine-supported static-analysis techniques which rediscovered the very errors leading to the multi-billion US dollars ARIANE 5 disaster [LMR+98].

The higher industrial priority of machine-supported verification is illustrated by the reaction to the Pentium bug. This bug concerns an error in the floating-point division algorithm FDIV implemented in the first Pentium processor. It was reported in the New-York Times on Nov. 11, 1994. Natarajan Shankar describes the flurry of activity within the machine-supported verification community following its discovery like this [Sha01]:

> Vaughan Pratt [Pra95] gives a cogent account of the discovery and diagnosis of the Pentium FDIV bug. The Pentium hardware used an SRT divider, so named for its independent inventors Sweeney, Robertson, and Tocher. This hardware algorithm is much

like grade school long division. In each iteration, the divisor is multiplied by the quotient digit and subtracted from the partial remainder to obtain the new partial remainder. A lookup table is used to predict the quotient digit for the next iteration based on an approximation of the partial remainder and divisor, so that it can be computed in parallel with the partial remainder. The Pentium FDIV bug has been traced to certain incorrect entries in a lookup table that were perhaps thought to be unreachable during execution.

Randy Bryant used model-checking techniques to show how BDDs (binary decision diagrams) and BMDs (binary moment diagrams) could be used to check an invariant for FDIV [BC95, Bry95]. BDDs are notoriously bad for representing boolean functions for multiplication, but BMDs do not have this problem. Ed Clarke and Xudong Zhao showed how BMDs could be used to carry out word-level model checking of an SRT algorithm [CZ95]. Clarke, German, and Zhao used Analytica, a theorem prover built on Mathematica, to verify an SRT algorithm [CGZ96], as did Rueß, Shankar, and Srivas, using the PVS theorem prover [RSS96, RSS99]. Paul Miner extended the latter verification to show compliance with the IEEE floating-point standard [ML96]. He showed how a correct SRT lookup table could be generated using PVS within a matter of hours. Moore, Lynch and Kaufmann verified a floating point division microcode routine used in the AMD K5 processor [MLK98].

To this account J Moore adds [Moo01]:

The K5 was AMD's first Pentium-class microprocessor. We did the verification in the summer of 1995 and it was of commercial importance to AMD. The Pentium FDIV bug became widely known in November, 1994. In January, 1995, Intel announced a $475 million write-off to cover costs associated with FDIV. There was much public wringing of hands and anxiety about it in spring, 1995.

Simultaneous with that anxiety, AMD was preparing to tape out the K5 for summer, 1995. Because of management decisions changing the amount of silicon area devoted to the FPU, the FPU team had thrown out their well-tested FDIV hardware in late 1994 (nothing to do with Intel's FDIV problems) and reimplemented FDIV on the K5 in microcode. Intel's $500 million loss was obviously on the minds of the AMD folks in the Spring. They did not want to repeat it.

That is what makes our proof so important in the history of this. We did not come in after the fact and look at what might have been. We came in *before* the K5 was taped-out and looked at what *was*. We delivered a Q.E.D. to them 9 weeks after we started the project.

After the successful conclusion of the FDIV proof, David Russinoff, an AMD employee, was tasked to do K5's FSQRT. His ACL2 proof attempt uncovered a bug which had survived testing and which was fixed before tape out [Rus99].

AMD is continuing this work, with four full-time AMD employees using ACL2 all the time to verify aspects of their designs. All of the elementary floating-point operations on the AMD K7 (the "Athlon") have been mechanically verified with ACL2. Several bugs were found and fixed before tape out. None have been reported in either the fabricated K5 or Athlon (related to the formally verified operations).

But even with machine-supported verification one serious problem remains:

How to write a correct specification? This task is as hard as the verification problem.[8] A program which is proved to be correct relative to a faulty formal specification remains incorrect w.r.t. its intended purpose, whether machine checked or nor. Our conclusion is that formal methods do not give us a full 100% guarantee of the correctness of a program. They only *increase the confidence* one has in its correctness. Consequently, formal methods do not eliminate the need for testing – testing is still needed [BH95b, BH95a]!

1.5 The Approach of this Book

Next we describe the approach taken in this book.

In Section 1.5.1 this approach we characterise as a state-based dual-language approach, which is semantically-oriented as argued in Section 1.5.2. This approach is based on Floyd's inductive assertion method, which is defined in Section 1.5.3, but which is not appropriate for program development. Consequently, a change of paradigm is needed leading to Hoare logic, as discussed in Section 1.5.4. Hoare logic for sequential program constructs is compositional; why compositionality enables program derivation is explained in Section 1.6. These formalisms are extended to concurrency in Section 1.5.5. However, the resulting formalisms are again not appropriate anymore for concurrent program development since they have become noncompositional. A second change of paradigm, this time to assumption-guarantee-style formalisms, which are compositional w.r.t. concurrency, is described in Section 1.5.6. The closing Section 1.5.7 describes a noncompositional method for establishing correctness of both shared-variable concurrent and distributed network algorithms: the communication-closed-layers paradigm.

1.5.1 A State-Based Dual-Language Approach

The approach taken in this book is called *state-based.* Let us first explain this

[8] September 30, 1999: Mars Climate Orbiter team finds likely cause of loss in specification error. *One team used imperial units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation. This information was critical to the manoeuvers required to place the spacecraft in the proper Mars orbit* [NAS99].
December 3, 1999: Mars Polar Lander lost. It crashed due to a race condition in the program that steered the Lander during the last 40 metres of its descent prior to landing. This race condition was discovered by reprogramming the Lander's original landing program into JAVA, and then using static analysis tools for discovering that error. This reprogramming was necessary because the Mars Lander's original software was written in C, which is notoriously unverifiable because of its facilities for pointer arithmetic.
These failures point to the likelihood that such errors will continue to occur, as long as NASA does not exhaustively apply automatic verification tools to its software *prior to launching its vehicles*.

notion.

Recall that states are total functions from a set of variables to a set of values of these variables. In a state-based approach one uses states both to characterise the meaning of programs by sets of (action-labelled) state sequences, and to give meanings to their specifications.

However, there are significant differences in style between languages for formulating properties and specifications of a computational task, and programming languages which describe precise, unambiguous and efficient solutions on a computer which are reasonably easy to execute. The level of abstraction of the former is usually much higher than that of the latter. This certainly holds for the specification languages used in this book, which are based on first-order predicate logic. Consequently, the key issue in our mathematical theory of state-based program development and analysis is establishing the relationships between these two widely different levels of description of a computational task. This approach is, therefore, called the *dual-language* approach.

In general, the natural questions one could ask about the relationship between a programming language $\mathcal{P}$ and a specification language $\mathcal{S}$ can be classified as follows:

- **The Program Development Problem**: Given a specification $s \in \mathcal{S}$, construct a program $p \in \mathcal{P}$ which satisfies the specification.
- **The Analysis Problem**: Given a program $p \in \mathcal{P}$, find its appropriate description $s \in \mathcal{S}$.
- **The Verification Problem**: Given a specification $s \in \mathcal{S}$ and a program $p \in \mathcal{P}$, check whether they are consistent, i.e., whether p satisfies s.

As can be seen from the list of these problems, a crucial concept to all of them is that of a program *p satisfying* a specification s. Therefore, the verification problem is the most fundamental question of all, and the concepts and methodologies developed for its resolution underly the methodologies and approaches to the other problems mentioned.

Summarising, this book offers a state-based dual-language approach to the verification of concurrent and distributed systems, using specification languages based on first-order predicate logic. When this approach is *compositional*, we shall see that it can be used for formal program development.

1.5.2 A Semantic Approach

Certainly, programming languages require a precise syntax to make them amenable to computer implementation, and the same holds for specification languages. However, the aim of specific languages is primarily ease of expression and understanding by us, humans! And the same holds for high-level programming languages, for there is no single mortal who masters all the details involved in execution of these languages on even as simple a computer as a laptop. That is, one also needs high-level abstractions for understanding how a program functions. Thus our approach is based on giving meanings to both programs and their specifications, on the basis of which a precise answer can be given to the question whether a given program p satisfies a given specification s.

As we have seen before, the meaning of a concurrent program is its set of computation sequences. Therefore, the verification problem for a (concurrent) program reduces to checking whether its computation sequences satisfy the specification concerned. That is, once we know when a computation sequence satisfies a specification, the verification problem can be reduced to proving inclusion between two sets: the computation sequences generated by a given program, and the computation sequences satisfying a given specification.

More precisely, since programs manipulate data from given data domains, we will reduce the above-mentioned set-inclusion problem for a given program p and specification s in Chapter 2 to a finite set of mathematical properties which must hold in those data domains in order for p to satisfy s. Thus, using the terminology *verification conditions* for these domain properties, the field of program correctness concerns itself primarily with the question how to reduce the claim "p satisfies s" to a finite set of verification conditions which must hold in order for p to satisfy s. Observe that the finiteness of this set is the principal reason which makes it possible to give program correctness proofs at all.

Whether these verification conditions themselves are true or not, over, e.g., the domain of natural numbers, reals, lists and other data types, we consider primarily a problem whose solution belongs to the realm of mathematics, whereas the reduction of the problem whether p satisfies s to a finite set of verification conditions, and the expression of these verification conditions in given specification languages, are considered to be part of computer science.

The most direct way to generate verification conditions for programs and their specifications is based on *Floyd's inductive assertion method.* Therefore, this book is based on that approach, which will be introduced in the next section.

1.5.3 Floyd's Inductive Assertion Method

In order to analyse the behaviour of programs we represent them by extremely simple objects, called *transition diagrams,* with an equally simple semantics.

These were introduced by R. Keller [Kel76]. Yet their variety turns out to be sufficiently rich to express shared-variable concurrency and message-passing.

Transition diagrams are finite directed graphs with a distinguished entry node s and exit node t (indicating where the computations start and end, respectively), whose edges are labelled by transitions of the form $c \to f$ where c denotes a boolean state function, called *guard,* and f a state transformation used for expressing multiple assignments. A computation of transition diagram T starts in its entry node s and some initial state, and is, whenever possible, prolonged, having reached node l, $l \neq t$, in state σ, by executing some successor transition $c \to f$ of T for which $c(\sigma)$ is true and which labels some edge $\langle l, l' \rangle$ of T, resulting in reaching label l' in state $f(\sigma)$. The semantics of a transition diagram consists of the set of its *maximal* computations.

Since the observable behaviour of a sequential program construct is given by its set of pairs of initial and corresponding final states, the simplest way to specify this behaviour for a transition diagram is by providing a pair of predicates (boolean functions) $\langle \varphi, \psi \rangle$, with φ called *precondition* and ψ *postcondition,* which those initial and final states, respectively, should satisfy. Since φ and ψ describe properties of states, this specification approach is called *property-oriented.*

In order to prove that a transition diagram T satisfies a pre-postcondition pair $\langle \varphi, \psi \rangle$, we construct an *inductive assertion network.* This is a mapping Q of the nodes in T to boolean state functions satisfying certain verification conditions. These verification conditions are generated for every directed edge e of T, and are of the form $Q_l \wedge c \to Q_{l'} \circ f$, where $c \to f$ labels edge e, which leads from node l to l', and Q associates boolean state functions Q_l and $Q_{l'}$ with l and l'. Additionally, the verification conditions $\varphi \to Q_s$ and $Q_t \to \psi$ are generated. When all these verification conditions hold, it is easy to see that T satisfies the pre-postcondition pair $\langle \varphi, \psi \rangle$.

This proof method is due to Robert W. Floyd [Flo67], and is called the *inductive assertion method.*

Properly speaking, it only proves *partial correctness,* which is defined by: T is partially correct w.r.t. φ and ψ when

> *every computation of T which starts in an initial state satisfying precondition φ, and which terminates in T's exit node, does so in a final state satisfying postcondition ψ.*

Observe that partial correctness does not imply termination. Floyd also formulated the main proof principle for establishing termination of programs, which combines the use of inductive assertion methods with so-called wellfounded orderings. These two methods, as well as their adaptations to proving *absence of runtime errors* and *blocking,* are discussed in Chapter 2.

Runtime errors arise during a computation when the static transformation f associated with a successor transition, whose guard evaluates to true, is executed in a state for which f is undefined; they occur, e.g., when executing $x := 1/0$, and $x := 1/y$ for $y = 0$.

Blocking arises when a computation, having reached a node l, $l \neq t$, cannot be prolonged by executing some successor transition, because none of the guards associated with transitions starting in l evaluates in σ to true.

Both Floyd's inductive assertion method and his method for proving termination are ideally suited for giving *soundness* and *completeness* proofs – which are given for all proof methods formulated in this work, and generalise well to shared-variable concurrency and distributed message passing.[9]

1.5.4 Hoare Logic

The inductive assertion method has no use for *program development* because of its lack of structure – an inductive assertion network for a transition diagram is not (yet) *formally* related to any of the subnetworks for its corresponding subdiagrams and, consequently, no notion of abstraction or refinement is available for the particular way these subnetworks are constructed.

To overcome this obstacle, we add sufficient structure to the inductive assertion method to use it for program development, although this leads to a change of paradigm. For in the process of adding this structure we obtain a so-called *Hoare logic,* and this logic, when confined to sequential program constructs, will turn out to be ideal for program derivation.

Adding this structure is done by:

- introducing a simple sequential programming language (with operators as sequential composition, looping constructs and guarded commands),
- defining the meaning of a program construct S by associating a transition diagram $T[\![S]\!]$ with it (whose semantics has been explained above),
- extending the algebraic structure of S to $T[\![S]\!]$ and to inductive assertion networks for $T[\![S]\!]$.

For example, for $S = S_1 ; S_2$ we have that $T[\![S]\!]$ can be written as $T[\![S_1]\!] ; T[\![S_2]\!]$, and that an inductive assertion network Q for $T[\![S]\!]$ can be written as $Q_1 ; Q_2$ with Q_i being an appropriate inductive assertion network for $T[\![S_i]\!]$, $i = 1, 2$. A

[9] Without such proofs, in the experience of the senior author, one can be pretty sure that the methods proposed are neither sound nor complete. And who wants to use a proof method whose conclusions are possibly wrong?

minor consequence of this set-up is that one has to work with *transition systems* instead of transition diagrams; transition systems are equivalence classes of transition diagrams w.r.t. renaming their nodes.

This structural enrichment leads to the development of a logic for reasoning about (inductive assertion networks for the transition diagrams corresponding to) programs in this language, in which every program operator has its own associated rule. And this logic turns out to be isomorphic to *Hoare logic* [Hoa69].

Before explaining the concept of Hoare logic, we formulate its single most important property (for sequential program constructs) which justifies its use for program development: Hoare logic for sequential program constructs is *compositional,* that is,

> *that a program meets its specification can be verified (in Hoare logic) on the basis of the specifications of its constituent components, without the need of additional information about the interior construction of these components.*

Why compositionality justifies (formal) program derivation, and how, is explained in Section 1.6.

Hoare logic is a formal axiom system for proving *Hoare triples.* These are of the form $\{pre\}S\{post\}$ with *pre* and *post* expressions from first-order predicate logic, called assertions, and S a (syntactically-expressed) program construct (in some programming language). Such a triple is *valid* whenever the transition diagram $T[\![S]\!]$ associated with S is partially correct w.r.t. $\mathcal{T}[\![pre]\!]$ and $\mathcal{T}[\![post]\!]$, which denote the interpretations of *pre* and *post*.

Since the observable behaviour of sequential program constructs is given by their sets of initial and corresponding final states, this behaviour can be fully specified in Hoare logic. That is, no additional information is required about the implementation of these constructs in order to formulate proof rules for them in Hoare logic. This explains why Hoare logic for sequential program constructs is compositional.

As a consequence of the set-up given above, we are able to establish a one-to-one mapping between correctness proofs in Hoare logic for programs S and the corresponding inductive assertion networks for $T[\![S]\!]$. This enables one to "lift" the soundness and completeness proofs for the inductive assertion method to similar proofs for Hoare logic. Here we use the result that the boolean functions involved in the former proofs can be expressed as first-order predicates over the standard model of the natural numbers by means of Gödel-encoding techniques [Coo78], as discussed in Chapter 5.

This set-up of Hoare logic is followed in Part IV and developed for the language of guarded commands in Chapter 9.

1.5.5 Extensions to Concurrency

Straightforward as the above development is for sequential programming constructs, the direct application of Hoare logic to program development is lost as soon as concurrency, i.e., communication between processes, enters the picture. For then the resulting Hoare logic is not compositional anymore. Again, a change of paradigm is called for, this time by switching to so-called *assumption-commitment*-type formalisms, to be discussed in the next section and in Section 1.6. But first we explain how to extend our proof to concurrency in noncompositional fashion.

The first question to answer is how to characterise concurrency, i.e., both shared-variable concurrency and message-passing, within the framework of transition diagrams.

For shared-variable concurrency this is immediate. For example, consider two transition diagrams P_1 and P_2, called *processes.* Their parallel composition $P_1||P_2$ is the following transition diagram:

- its nodes are pairs $\langle l_1, l_2 \rangle$ with l_i a node of P_i,
- its transitions are essentially either transitions of P_1 or transitions of P_2, i.e., of the form $(\langle l_1, l_2 \rangle, c \to f, \langle l_1', l_2' \rangle)$ with $(l_i, c \to f, l_i')$ a transition of P_i and $l_j = l_j'$, $i \neq j$, i.e., the individual transitions of P_1 and P_2 are *interleaved* inside $P_1||P_2$,
- its entry node is $\langle s_1, s_2 \rangle$, with s_i the entry node of P_i,
- its exit node is $\langle t_1, t_2 \rangle$, with t_i the exit node of P_i,

for $i = 1,2$ and $j = 1,2$.

A similar kind of definition can be given for synchronous message passing. (*Asynchronous message passing* requires the addition of buffers to the synchronous picture, where buffers communicate synchronously with processes, processes only communicate with buffers, and buffers are also described by transition diagrams.)

It is essential here that the parallel composition of transition diagrams results again in a transition diagram. This implies that the proof methods developed for sequential constructs apply in principle also to concurrency. However, there is a well-known catch: Given n parallel processes of r nodes, their parallel composition has r^n nodes – a phenomenon called *state-space explosion.* Consequently, giving a correctness proof for this parallel composition using the inductive assertion method requires an inductive assertion network consisting of r^n assertions, and checking even more verification conditions. Therefore this *global* method, as it is called, cannot be applied in practice except for very small values of r and n.

As a way out of this exponential increase in complexity of a correctness proof, historically the following strategy has been followed for proving properties of $P_1\|\ldots\|P_n$:

(i) *For all $i = 1,\ldots,n$, give a local correctness proof Q_i for transition diagram P_i which only applies when execution of P_i takes place in isolation, i.e., is merely viewed as a sequential noncommunicating transition diagram.*

(ii) *Prove the additional properties which must be satisfied by inductive assertion network Q_i in order for it to apply also when P_i is executed in the context of $P_1\|\ldots\|P_{i-1}\|P_{i+1}\|\ldots\|P_n$, for $i = 1,\ldots,n$.*

The problem here is that these local correctness proofs merely characterise the initial-final state behaviour of the component process P_i, whereas this characterisation does not suffice for concurrency because the interaction between concurrent processes depends on their intermediate states as well. For example, in case of shared-variable concurrency of a process, the intermediate states belong also to the observable behaviour and therefore actions upon those intermediate states by the environment also need to be specified. So, instead of "adding up" these local correctness proofs (which would have been sufficient in the absence of any interaction between parallel processes), we need a more refined dependency between the inductive assertion networks $Q_1,\ldots,Q_n$ and $P_1,\ldots,P_n$, which expresses the additional interaction by its environment upon the intermediate states of a process.

The new element required is *that the individual assertions of assertion network Q_i for process P_i should be* invariant *under execution of the transitions of all remaining processes P_j, $j \neq i$.*

The *interference freedom test,* formulated by Susan Owicki [Owi75, OG76a] for Hoare logic and Leslie Lamport [Lam77] for the inductive assertion method, guarantees that communication between P_i and P_j at intermediate points (nodes) during the execution of P_i and P_j *preserves* the assertions attached (by Q_i and Q_j) to those points.

For synchronous message passing these additional verification conditions are called *cooperation tests* [AFdR80, LG81].

It can be checked that the complexity of these methods is $O(n \times s \times r)$ in the number of verification conditions, with n the number of processes, assuming that P_i has r locations and s edges, which is quite an improvement over the global method formulated at the beginning of this section, where the complexity is $O(n \times s \times r^{n-1})$ in the number of verification conditions.

A precise formulation and the mathematical analysis of the resulting proof methods is worked out for the inductive assertion method in Chapters 3 and 4; their corresponding Hoare logics are developed in Chapters 10 and 11. These chapters also contain numerous applications of these methods, e.g., for obtaining correctness proofs of (suitably corrected versions of) the algorithms for concurrent garbage collection discussed in Section 1.4, algorithms for partitioning a set, and for mutual exclusion.

Observe that these proof methods *are not compositional.* This is because the formulation of these additional verification conditions for, say, process P_i requires information about the individual transitions of all processes P_j, $j \neq i$, to be available for $i = 1, \ldots, n$, and this constitutes *"additional knowledge about the implementation of* $P_1, \ldots, P_n$ *which is not given by their specifications."* And the latter is forbidden in case of compositional proof methods.

Consequently, these proof systems can no longer be used for formal program derivation. The fact that the behaviour of processes is now also characterised by their interaction at intermediate control points, results for the approach sketched above in a tremendous number of verification conditions. As a result it is no longer clear how to formulate an appropriate formal notion of *development* of concurrent programs from their specifications, which takes these additional verification conditions into account. This is remedied in the next section.

Please note that the latter remarks only apply to the process of *formal* program derivation from specifications. They do not apply to more heuristically-oriented methods for deriving concurrent programs. For example, consult the recent work of Wim Feijen and Netty van Gasteren [FvG99] for successful concurrent program derivation (often of new algorithms) using noncompositional techniques in heuristic fashion, as developed in the school of Edsger W. Dijkstra [Dij76, Dij82].

1.5.6 The Assume-Guarantee Paradigm

The key to formulating compositional proof methods for concurrent processes is the realisation that one has to specify not only their initial-final state behaviour, but also *their interaction at intermediate points.* This was done for shared-variable concurrency by Cliff Jones in [Jon81, Jon83] and for synchronous message passing by Jay Misra and Mani Chandy in [MC81].

The basic intuition

The basic intuition underlying the assume-guarantee paradigm can be explained by considering the very simple example of an adder component P shown in Figure 1.7 that adds two input numbers x and y and places the output in z. We owe this example to Natarajan Shankar [Sha98b].

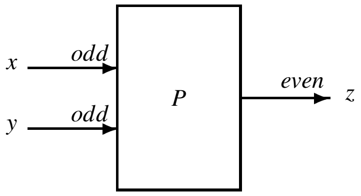

Fig. 1.7. Even number generator P.

Here x, y and z can be program variables, signals, or latches, depending on the chosen model of computation. The system containing P as its component might require its output z to be an even number, but obviously P cannot unconditionally guarantee this property of the output z. It might be reasonable to assume that the environment always provides odd number inputs at x and y, so that with this assumption it is easy to show that the output numbers at z are always even. Only local reasoning in terms of P is needed to establish that z is always even when given odd number inputs at x and y.

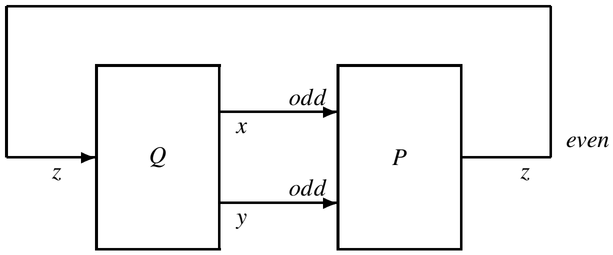

Fig. 1.8. Odd and even number generators.

If, as shown in Figure 1.8, P is now composed with another component Q that generates the inputs at x and y, then to preserve the property that only even numbers are output at z, Q must be shown to output only odd numbers at x and y. However, the demonstration that Q provides only odd numbers as outputs at x and y might require assumptions on the inputs taken by Q, where z itself might be such an input. If in showing that Q produces odd outputs at x and y, one has to assume that the z input is always even, then we have an obvious circularity and nothing can be concluded about the oddness or evenness of x, y and z. If this circularity can somehow be broken, then we have a form of well-founded mutual recursion between P and Q that admits a proof by simultaneous induction that x and y are always odd and z is always even. The circularity can be broken by noting that a z output for P is even as long as the *preceding* x and y inputs are odd, and the x and y outputs for Q are odd as long as the *preceding* z input is even. That is, using the terminology of the assume-guarantee paradigm, the guarantee that output z is even is satisfied by *P as long as the assumption that its previous inputs x and y are odd has always been satisfied before by P's environment*.

In the assume-guarantee approach for reasoning about concurrency, a property of a component process is expressed by a pair (A,G) consisting of a commitment G that the component will satisfy provided the environment of the component satisfies the assumption property A. The interpretation of (A,G) has to be carefully defined to be non-circular. Informally, a component P satisfies (A,G) if the environment of P violates A before the component P fails to satisfy G. That is, P satisfies (A,G) if at any stage of an on-going computation P's actions satisfy G as long as A is satisfied by all actions from P's environment. When two or more components, P_1 satisfying (A_1,G_1) and P_2 satisfying (A_2,G_2), are composed into a larger component $P_1||P_2$, the assumption A of $P_1||P_2$ together with property G_1 of component P_1 must be used to show that P_1's actions do not violate assumption A_2, and correspondingly, A and G_2 must be used to show that P_2 does not violate assumption A_1 [Sha98b].

And its formalisation

Jones proposed in his so-called *rely-guarantee* (R-G) formalism *to specify the interference allowed from the environment of a component during its execution without endangering fulfilment of the purpose of that component.* To this end Jones extended the format of Hoare triples with two additional predicates: a *rely* predicate, for specifying the expected behaviour of the environment, i.e., the interference allowed, and a *guarantee* predicate for expressing the task to be performed, by that component. Both these predicates are *action* predicates, i.e., they characterise *atomic transitions.*

More precisely, a rely-guarantee correctness formula has the form

$$\langle rely, guar\rangle : \{pre\}\ P\ \{post\},$$

where *pre* and *post* impose conditions upon the initial and final, respectively, states of a computation of process P, whereas *rely* and *guar* impose conditions upon environmental transitions and transitions of P itself, respectively. Validity of such an R-G formula for a process P is captured by the following intuitive characterisation of validity of an R-G correctness formula:

If

(i) P is invoked in an initial state which satisfies pre, and
(ii) at any moment during the computation of P all environment transitions satisfy rely,

then

(iii) all P-transitions up to that moment satisfy guar, and
(iv) if this computation terminates, its final state satisfies post.

A simple example of R-G-based reasoning is worked out in Section 6.5.

The proposal of Misra and Chandy is similar, be it that it applies to synchronous communication, and is called the *assumption-commitment* (A-C) formalism; a simple example is worked out in Section 6.4. A difference with the R-G formalism is that the *assumption* and *commitment* are predicates over the communication history of a component. Following Abadi and Lamport [AL93, AL95] we shall use the term *assume-guarantee* formalism (that is, paradigm) when discussing these formalisms, because they embody the same idea.

The crucial element common to both is that they specify the amount of interference which is allowed by a process *without endangering its purpose*. In the case of the A-C formalism this allowed amount is specified by an *assumption* predicate, whereas the own contribution of a process is specified by its *commitment* predicate (and postcondition). That is, in general one needs four predicates in an assume-guarantee specification: two for specifying initial-final state behaviour, and two for specifying the combination of allowed interference and desired behaviour.

As a consequence of this specification format, it turns out to be technically possible to give for each communicating process a specification of both (i) its initial and resulting final states and (ii) its full interaction with its environment. That is, this specification format allows a full characterisation of the observable behaviour of a process. This implies, in particular, that, in these formalisms, parallel composition rules can be formulated entirely in terms of those predicates without any need for additional details about the implementation of the processes concerned.

That is, the resulting proof methods are compositional! And this makes these formalisms suitable for the formal top-down derivation of concurrent processes from their specifications [Jon83, Oss83] (see Section 7.4.3 for an illustration of the systematic top-down development of a distributed priority queue).

An important application of assume-guarantee-based reasoning is given by its capacity to specify *open* systems, that is, systems which are intended to interact using certain prespecified shared variables or shared channels with the environment, without the precise implementation of that environment being known. Observe that this is consistent with the very purpose of compositional reasoning, since, because the environment of an open system is not known beforehand, *it can only be specified without giving any implementation details,* and this is exactly what compositional reasoning is about! Thus, one can summarise the above discussion by observing that the assume-guarantee paradigm is suitable for the formal top-down design (and refinement) of open concurrent systems.

It should be remarked that there are many other compositional state-based approaches to concurrency verification, see Section 1.6 for references. However, of all these approaches, this assume-guarantee paradigm is the one studied best.

Chapter 6 contains a gentle introduction to the assume-guarantee paradigm, Chapter 7 discusses the A-C formalism, and Chapter 8 analyses the R-G formalism.

Section 1.6 contains more material on compositional proof methods for concurrency.

We close Section 1.5 with Section 1.5.7 describing a promising noncompositional paradigm which has been successfully applied to proving the correctness of concurrent and distributed network algorithms and protocols, such as discussed in [Ray88]: *the communication-closed-layers paradigm,* originally proposed by Tzilla Elrad and Nissim Francez [EF82].

1.5.7 The Communication-Closed-Layers Paradigm

In Section 1.5.5 we observed that for noncompositional proof methods, which are based on interference freedom and cooperation tests, no clear notion of program derivation or program refinement exists on account of the additional complexity which is introduced by these tests within the verification process.

One way to reduce this complexity is the subject of Chapter 12. In that chapter we take the point of view that, in order to explain and clarify the final structure of a network of processes, *the structure of a correctness proof should reflect the structure of the original design process*, rather than the structure of the finally resulting network. To this end we introduce program transformation techniques that are used in combination with the Owicki & Gries style of program verification of Chapter 10. They are called *communication-closed-layer laws* (CCL).

These techniques distinguish between the "*physical structure*" and the "*logical structure*" of programs. The physical structure refers to a program S_D that has a syntactic structure that is reasonably close to the actual physical (process) architecture that will be used to execute a program. We assume here that this physical structure can be described as a collection of network nodes P_j, each of which executes a number of sequential phases $P_{0,j},\ldots,P_{n,j}$. Schematically S_D can be described as a program of the following form:

$$S_D \stackrel{\text{def}}{=} [(P_{0,0}\,;P_{1,0}\,;\cdots;P_{n,0}) \;\|\; \cdots \;\|\; (P_{0,m}\,;P_{1,m}\,;\cdots;P_{n,m})].$$

The logical structure on the other hand is a program S_L that has a different syntactic structure that allows for simple verification, and appears to be *sequentially phased*, that is, has the following form:

$$S_L \stackrel{\text{def}}{=} [P_{0,0} \;\|\; \cdots \;\|\; P_{0,m}]\,;\cdots;[P_{n,0} \;\|\; \cdots \;\|\; P_{n,m}].$$

The aim of Chapter 12 is to explain under which circumstances such sequentially-phased programs S_L are actually *equivalent* to their corresponding distributed version S_D, where "equivalence" is understood as "computing the same initial-final-state relation" (stronger forms of equivalence which are preserved under parallel composition are also discussed).

A number of examples, amongst others, correctness proofs for the two-phase commit protocol and set partitioning (one of the running examples of this volume), illustrate the effectiveness of these noncompositional CCL-based techniques.

1.6 Compositionality

This section presents a detailed discussion of the concept of compositionality (in Section 1.6.2) from the viewpoint of the verify-while-develop paradigm (discussed in Section 1.6.1), supplies additional motivation for its application in the context of machine-supported verification (in Section 1.6.3), gives a detailed account of its relationship with Lamport's view of specification as a contract between client and implementor (in Section 1.6.4) which is expressed by another concept – modularity, and analyses some beliefs about the complexity of compositional reasoning in Section 1.6.5. Finally, we answer in Section 1.6.6 the third question posed in Section 1.1: Should one use compositional or noncompositional methods?

1.6.1 The Verify-While-Develop Paradigm

Most of the literature on program correctness deals with *a posteriori* program verification, i.e., the situation that *after* the program has been obtained it is proved correct, see, for instance [Man74, And91, AO91, Dah92, Fra92, Sch97]. Although a posteriori verification has been indispensable for the development of a truly mathematical theory of program correctness, it is a frustrating approach in practice, because almost all programs contain errors. Discovering that error then leads to the need for correcting the program. And the program corrected in that way contains most of the time errors again. Nevertheless one hopes that this is no nonterminating cycle. Hence the attempt at a correctness proof leads in most cases to the discovery of errors, i.e., a counterexample violating the original specification, rather than a successful proof.

This situation becomes even more serious when considering concurrent programs and systems. For then the number of cases to be verified when checking the correctness of the interaction between their individual processes grows exponentially w.r.t. the number of those processes. Consequently, checking them all becomes well-nigh impossible, except, maybe, when this process of checking has been mechanised. But even then the limits of naive machine verification are quickly reached.

It would make sense, if, rather than disproving finished programs over and over again in the course of attempting to prove them correct, one could be convinced of the correctness of a design decision *at the very moment one took that decision* during the process of top-down development of a program from its specifications – a paradigm called *"verify-while-develop"*. This paradigm requires one to verify that a program-in-the-making meets its specification by making use of specifications for its constituent parts or modules *while the implementing program text of these parts is not yet available*. This has the advantage that, since every top-down development step now requires a correctness proof based on the specifications of these parts (rather than a description of their execution mechanism), any errors made later during development of these parts which violate those specifications *do not influence the correctness of this particular development step*. Consequently, no redundant verification work is done when following this paradigm, in contrast to the situation with a posteriori program verification.

Of course, the verify-while-develop strategy is not new. It is merely a reformulation of the notion of *hierarchically-structured program development,* advocated by Edsger W. Dijkstra, Niklaus Wirth and others since the end of the sixties [Dij68c, Wir71, Dij76, Dij82, FvG99].

As we shall explain in the next section, the verify-while-develop paradigm amounts to compositional verification.

1.6.2 Compositionality: the Very Concept

The technical property which is required of program correctness methods for supporting the verify-while-develop paradigm is called *"compositionality"*, whose definition is recalled below:

> *"That a program meets its specification should be verified on the basis of specifications of its constituent components only, without additional need for information about the interior construction of those components."*

To make this verification strategy possible, programs and their components are specified using predicates *over their observable behaviour, only*. Such specifications are called *assertional*. Consequently, assertional specifications of the constituent components of a program never depend on any additional knowledge about the underlying execution mechanism of those components. Consequently, *compositional verification should be based on an assertional specification language.*

What does a compositional proof method consist of?

It consists of:

(i) *Basic techniques* for proving that a program P, which is not decomposed any further, satisfies φ.
(ii) *Compositional proof techniques* to handle the case that P is composed of parts $P_1, \ldots, P_n$, i.e., $P = op^{lang}(P_1, \ldots, P_n)$, $n \geq 1$, with op^{lang} some operator of the programming language.

For the latter kind of program operators we develop *compositional proof rules*, i.e., logical inference rules of the form:

"From P_1 *satisfies* φ_1 and ... P_n *satisfies* φ_n infer P *satisfies* φ." [Zwi89]

In order to do so we need to be more specific about assertional specifications through predicates. They should not refer to the interior construction of components. Therefore, the available information for specifying a component consists only of:

- a description of the desired observable behaviour of the component P_i (as given by the assertional specification φ_i), and
- a description of the interface $obs(P_i)$ of that component P_i.

This leads to the introduction of restrictions upon the specifications φ_i of P_i, whose purpose it is to guarantee that φ_i is invariant under independent execution of P_j, $j \neq i$. These restrictions concern the sets of observables allowed to occur within φ_i, and impose that those observables, which are involved in independent executions of P_j, $j \neq i$, are disjoint from the observables of φ_i.

Example 1.12 In the A-C rule for parallel composition (see Section 7.5.3) one imposes the following restrictions on φ_i:

- $Chan(\varphi_i) \cap Chan(P_j) \subseteq Chan(P_i)$, $i \neq j$.
  Here $Chan(\varphi_i)$ lists those communication channels, communication along which influences the value of φ_i, and $Chan(P_j)$ the communications channels occurring in P_j. This restriction expresses that those channels along which communication influences the value of φ_i, and which also occur in P_j, $j \neq i$, must be connected to P_i. That is, along such channels communication between P_i and P_j is synchronised.
- $var(\varphi_i) \cap var(P_j) = \emptyset$.
  This restriction expresses that φ_i should not involve variables whose values might be changed by P_j. □

The characteristic feature of a compositional proof rule is its so-called *compositional proof step*. This step consists essentially of a proof that φ follows from some combination of $\varphi_1, \ldots, \varphi_n$. This proof amounts to checking the validity of a finite number of implications involving the predicates occurring in $\varphi_1, \ldots, \varphi_n$ and φ, and *therefore does not involve any representation of P_i at all*!

Example 1.13 In the A-C framework for reasoning compositionally about synchronously communicating processes, a specification φ consists, essentially, of a pair $\langle A, C \rangle$ consisting of an assumption A concerning the environment of a component and a commitment C of that component. The corresponding proof rule for parallel composition of two processes features the following compositional proof step:

$$\bigwedge_{i,j=1,2,\ i \neq j} (A \wedge C_i \rightarrow A_j),$$

expressing that the assumption A concerning the environment of $P_1 \| P_2$ plus the commitment C_i of the process P_i within this network together imply the assumption of the process P_j (this takes into account that the environment of P_j is process P_i plus the environment of $P_1 \| P_2$), for $i \neq j$. □

Consequently, a compositional proof rule introduces three kinds of premises for deducing that $op^{lang}(P_1,\dots,P_n)$ *satisfies* φ:

(i) A compositional proof (reduction) step, stating that φ follows from some combination of $\varphi_1,\dots,\varphi_n$.
(ii) Conditions upon the observables allowed within $\varphi_1,\dots,\varphi_n$ w.r.t. program components $P_1,\dots,P_n$.
(iii) Compositional verification that P_i *satisfies* φ_i, $i = 1,\dots,n$.

These considerations lead to the following definition of a *compositional proof rule for a program operator* $op^{lang}(P_1,\dots,P_n)$:

$$\frac{\begin{array}{c}\textit{for } i=1,\dots,n,\ P_i \textit{ satisfies } \varphi_i \textit{ and } obs(P_i)\subseteq O_i,\\ op^{spec}(\varphi_1,\dots,\varphi_n,\varphi;obs(P_1),\dots,obs(P_n))\end{array}}{op^{lang}(P_1,\dots,P_n)\ \textit{satisfies}\ \varphi,}$$

where $obs(P_i)$ refers to the interface of P_i, i.e., P_i's set of observables, $O_1,\dots,$ O_n denotes given sets of observables, $\varphi_1,\dots,\varphi_n$, and φ expresses assertional specifications, and $op^{spec}(\varphi_1,\dots,\varphi_n,\varphi;obs(P_1),\dots,obs(P_n))$ expresses points (i) and (ii) above[10].

Various proof methods are considered, each giving separate interpretations to φ, *"P satisfies* φ*"*, $obs(P_i)\subseteq O_i$, op^{lang}, and op^{spec}. For each of these methods it is essential that the *basic technique* to deal with programs which are not decomposed further (although adaptation rules may be applied) are formulated and, in particular, that appropriate (meta-)predicates $op^{spec}(\varphi_1,\dots,\varphi_n,\varphi;$ $obs(P_1),\dots,obs(P_n))$ for the compositional proof rules contained in the respective proof method are identified.

Although obviously for each n-ary operator op^{lang}, $n\geq 1$, one proof rule of the form above is needed, one needs additional proof rules in order to *adapt* one specification to another. These proof rules are not related to specific program constructs, but allow one to reason about specifications (and interfaces), only. They are of the following form:

$$\frac{\begin{array}{c}P \textit{ satisfies } \varphi\textit{, and } obs(P)\subseteq O,\\ op^{spec}(\varphi,\varphi';obs(P))\end{array}}{P\ \textit{satisfies}\ \varphi'}$$

The most frequently used rule of this type is the rule of consequence known from Hoare logic [Hoa69]. Since these rules do not depend on the structure of

[10] The clause $P_i\subseteq O_i$ is added here to cover restrictions on the components, e.g., the case in shared-variable concurrency where a process is required to have exclusive write-access to a variable. Its general purpose is to be able to impose further restrictions on $P_1,\dots,P_n$, although these are not considered here.

the actual program, but adapt its specification only, they are called *adaptation rules* [Old83b, Zwi89, ZHLdR95]. They do not refer to the internal structure of the program also and thus also qualify as compositional proof rules.

Therefore, a *compositional proof rule* is either a compositional proof rule for some program operator op^{lang} or an adaptation rule.

A proof rule is *sound* (or *valid*) whenever the truth of its premises implies the truth of its conclusions. Now a *compositional proof method* for establishing program correctness for some programming language consists of basic rules, dealing with constructs which are not decomposed any further, and compositional rules, dealing with

(i) constructs which are decomposed, and
(ii) the adaptation of specifications.

A proof method is *sound* if all its proof rules are sound.

Compositional reasoning owes its attractiveness to its application to parallel composition, since it replaces operational reasoning with a complexity *increasing exponentially* in the number of parallel components by reasoning compositionally, whose complexity increases *linearly* w.r.t. the number of those components. This is discussed further in Section 1.6.5.

Example 1.14 (The rule for sequential composition in Hoare logic is compositional) The rule for sequential composition in Hoare logic, cf. Rule 9.6, can also be formulated as:

$$\frac{\{pre_1\}P_1\{post_1\},\ \{pre_2\}P_2\{post_2\},\ post_1 \to pre_2,\ pre \to pre_1,\ post_2 \to post}{\{pre\}P_1\,;\,P_2\{post\}}$$

That this rule is compositional can be seen as follows:

- identify φ with a pair $\langle pre, post\rangle$ with *pre* and *post* predicates (that is, assertions from first-order predicate logic),
- define *P satisfies* φ by validity of $\{pre\}P\{post\}$,
- $obs(P)$ identifies the set of program variables occurring in P,
- $O_i \stackrel{\text{def}}{=} VAR$,
- $obs(P_i) \subseteq O_i$ is now satisfied,
- $op^{lang}(P_1, P_2) \stackrel{\text{def}}{=}$ “;”,
- $op^{spec}(\varphi_1, \varphi_2, \varphi; obs(P_1), obs(P_2)) \stackrel{\text{def}}{=} (pre \to pre_1) \wedge (post_2 \to post) \wedge (post_1 \to pre_2)$. □

Example 1.15 (The rule for parallel composition in the assumption-commitment formalism is compositional) The assumption-commitment rule for parallel composition, Rule 7.17, is formulated as:

$$\frac{\begin{array}{c}\langle A_1,C_1\rangle : \{\varphi_1\}\ P_1\ \{\psi_1\},\\ \langle A_2,C_2\rangle : \{\varphi_2\}\ P_2\ \{\psi_2\},\\ A\wedge C_1 \to A_2, A\wedge C_2 \to A_1\end{array}}{\langle A, C_1\wedge C_2\rangle : \{\varphi_1\wedge\varphi_2\}\ P_1 \| P_2\ \{\psi_1\wedge\psi_2\}}$$

provided

(i) $var(A_1,C_1,\psi_1)\cap var(P_2)=\emptyset$, $var(A_2,C_2,\psi_2)\cap var(P_1)=\emptyset$, and
(ii) $Chan(A_1,C_1,\psi_1)\cap Chan(P_2)\subseteq Chan(P_1)$,
$Chan(A_2,C_2,\psi_2)\cap Chan(P_1)\subseteq Chan(P_2)$.[11]

That this rule is compositional can be seen as follows:

- In general, identify φ with a quadruple $\langle A,C,pre,post\rangle$, where A and C are predicates from first-order predicate logic involving no program variables, and *pre* and *post* unrestricted predicates from first-order predicate logic; in this particular case, specification φ for $P_1\|P_2$ is chosen as $\langle A, C_1\wedge C_2, pre_1\wedge pre_2, post_1\wedge post_2\rangle$ with predicates subscripted by i referring to the corresponding predicates of the specification for P_i,
- validity of P *satisfies* φ is defined by validity of $\langle A,C\rangle : \{pre\}P\{post\}$,
- $obs(P_i)$: the variables $var(P_i)$ and channels $Chan(P_i)$ involved in P_i,
- $obs(P_i)\subseteq O_i$: take $O_i \stackrel{\text{def}}{=} \overline{var(P_j)}\cup Chan(P_i)$, $i\neq j$, where $\overline{var(P_j)} \stackrel{\text{def}}{=} VAR\setminus var(P_j)$,
- $op^{lang} \stackrel{\text{def}}{=}$ "$\|$",
- $op^{spec}(\varphi_1,\varphi_2,\varphi;obs(P_1),obs(P_2)) \stackrel{\text{def}}{=}$
$\bigwedge_{i\neq j}(A\wedge C_i\to A_j)\wedge$
$\bigwedge_{i\neq j}((var(A_i)\cup var(C_i)\cup var(post_i))\subseteq \overline{var(P_j)})\wedge$
$\bigwedge_{i\neq j}((Chan(A_i)\cup Chan(C_i)\cup Chan(post_i))\subseteq (Chan(P_i)\cup \overline{Chan(P_j)}))$,
with $Chan(post)$, for *post* a first-order predicate, as defined in Definition 7.10, and $i,j\in\{1,2\}$. □

Compositional completeness

The notion of *completeness* investigated here is that of *compositional completeness* [Zwi89]:
A proof method is compositionally complete if it is compositional and moreover complete in the following sense:

Whenever, for $P = op^{lang}(P_1,\ldots,P_n)$, P *satisfies* φ holds, there exist specifications $\varphi_1,\ldots,\varphi_n$ (and sets of observables $O_1,\ldots,O_n$) such that:

[11] Recall that $Chan(P_1,\ldots,P_n,\varphi,\ldots,\psi) \stackrel{\text{def}}{=} (\bigcup_{i=1}^{n} Chan(P_i))\cup Chan(\varphi)\cup\ldots\cup Chan(\psi)$.

(a) P_i *satisfies* φ_i, and $obs(P_i) \subseteq O_i$ holds for $i = 1, \ldots, n$, and

(b) $op^{lang}(P_1, \ldots, P_n)$ *satisfies* φ is provable within the proof system under the assumption of (a).

Observe the careful formulation of part (b) above. We do *not* state that $op^{lang}(P_1, \ldots, P_n)$ *satisfies* φ can be immediately deduced from P_i *satisfies* φ_i, $i = 1, \ldots, n$, using the compositional proof rule for op^{lang}. Rather, we state this inference can be carried out using a *number* of proof rules.

For example, in case of parallel composition one can prove that, in general, no $\varphi_1, \ldots, \varphi_n$ exist which allow $op^{lang}(P_1, \ldots, P_n)$ *satisfies* φ to be deduced from P_1 *satisfies* φ_1, $\ldots$, P_n *satisfies* φ_n, without taking additional properties into account. Rather, in case of synchronous message passing, $P_1 \| P_2$ *satisfies* φ is deduced by applying a combination of rules, such as the one for parallel composition, and the prefix invariance, conjunction and consequence rules.

Semantic completeness

Observe that *nowhere* in the account above we require assertional specifications to be syntactically expressible in some logical language. The above characterisation of compositionality can be formulated using assertional specifications which consist of *boolean functions* operating upon program states (used to capture the values of variables) and sequences of communications (for capturing the communication behaviour of a component or network).

This *semantic* approach abstracts from the issue of expressibility of the predicates and the derivability of the verification conditions, which in our framework are purely set-theoretic, i.e., directly formulated in terms of operations upon sets of states and communication histories.

The attractiveness of this abstraction is that completeness is now reduced to its very essence, i.e., the *existence of appropriate boolean functions* so that the proof methods and rules *are applicable*. This notion of completeness is called *semantic completeness*.

For example, semantic completeness of our formulation of the inductive assertion method for sequential transition diagrams P is demonstrated as follows. To show that every valid correctness formula $\{\varphi\}\ P\ \{\psi\}$, where φ and ψ denote sets of states (i.e., are set-theoretic predicates), is derivable, we have to associate with every location of P an appropriate set-theoretic predicate (denoting a set of states) such that the resulting network satisfies the required verification conditions.

In the sequential case, the resulting inductive network of semantic predicates is, in fact, a finite graphical representation of the computation tree of the given transition diagram (defined w.r.t. a given initial set of states), where finiteness is obtained by labelling each location with a set of states.

For concurrent systems, completeness of our proof methods involves making those verification conditions explicit *which allows one to combine* the (compositional) characterisations of the local proofs.

Semantically complete proof methods are worked out in Chapters 2–4, in which noncompositional approaches to concurrency verification are studied, and Chapters 6–8, focussing on compositional proof methods. Consequently, in Chapters 7 and 8 we develop *semantically compositionally complete* proof methods for concurrency.

Observe that by introducing the concept of semantic completeness, *expressibility* of predicates (within a given assertion language) and *derivability* of the verification conditions (which in our framework are purely set-theoretic, i.e., directly formulated in terms of operations on sets of states) now become separate issues. This leads to the introduction of notions like relative or arithmetical completeness, which are discussed in Section 2.7, and also in Chapter 5. This chapter introduces first-order predicate logic over the standard model of the natural numbers as assertion language.

This assertion language is used in Chapters 9–12, dealing with Hoare logic. Consequently, the verification conditions in those chapters refer to properties of that standard model. This has some repercussions on the notions of completeness which are used in those chapters, since by Gödel's result [Göd31] no complete axiomatisation exists for the standard model of the natural numbers. The impact of this on our work is the following. Instead of proving completeness we shall prove *relative completeness*. By this we mean that we assume that we have an oracle at our disposal to decide whether some assertion of our assertion language is valid or not. Relative completeness means that every valid specification can be formally deduced, where we may call upon this oracle to decide the validity of assertions. The typical case where the oracle is used is for application of Hoare's consequence rule.

Since we are relying on so-called strongly arithmetical interpretations for assertions in those chapters [Cla85], our notion of completeness is better called *arithmetical completeness* [Har79]. To avoid cumbersome terminology, we usually omit the adjectives "semantic", "relative" and "arithmetical" when we discuss completeness questions. That is, these qualifications are implicitly understood.

Compositionality and the verify-while-develop paradigm

Compositional proof techniques have the advantage that they allow a systematic top-down development of programs from their specification, which is correct by construction. And this process of program derivation is exactly the *verify-while-develop paradigm!* For, as a consequence of using compositional techniques, each compositional proof (reduction) step (as defined above) can be viewed as *the verification of a design step*, which only involves reasoning about (the assertional specification of) that particular step and does not involve any future design steps. This explains why in the definition of compositional proof techniques above we stipulate that no additional knowledge about the underlying execution mechanism of the constituent components (of the to-be-verified program) is allowed. For, without that clause, reasoning about a particular design step might involve reasoning about future design steps, and we want these two stages to be independent.

This explains why compositional techniques can be viewed as the proof-theoretical analogue of Dijkstra's hierarchically-structured program development.

In Section 9.1.3 the argument that compositional verification and the verify-while-develop paradigm refer to the same concept is worked out in detail for sequential Hoare logic.

1.6.3 Compositionality and Machine-Supported Verification

The promise of the joint verify-while-develop paradigm/compositional proof-generation strategy, which is discussed above, is based on its potential for widespread application, as witnessed by the growing list of academic and industrial claims in this respect [Lar87, LT87, CLM89, Jos90, KR90, LT91, HRdR92, LS92b, Bro93b, Jos93, KL93, CMP94, DH94, GL94, Jon94, Mos94, DJS95, Hoo95, Cau96, KV96, SY96, McM97, BLO98a, BLO98b, DJHP98, HQRT98, Sha98a, McM99, NT00].

This is mainly due to the effort made in recent years towards machine-support of this form of reasoning. This effort has been caused by the following factors.

As we have seen, compositional methods enable reasoning about such systems because they reduce properties of complex systems to properties of their components which can be checked independently of their environment. Now the same applies to reasoning about complex *finite state* systems. For example, verification approaches based on the fully automatic technique of *model checking* fail to scale up gracefully because the global state space that has to be explored can grow exponentially in the number of components [GL94]. Also here compositional verification offers a solution by restricting the model checking procedure to verifying local, component-level properties, and using combinations of interactive and machine-supported theorem proving for verifying that the machine-verified local properties imply the desired global property [CLM89, Jos90, Jos93, Kle93, GL94, DJS95, KV96, McM98, Sha98a]. This is discussed in [AHK98, DJHP98, BCC98].

Secondly, complex systems, and especially safety-critical ones, are in great need of formal verification. Also, the formal verification of crucial components of such systems has acquired higher industrial priority ever since INTEL lost an estimated 500 million US dollars due to a bug in its Pentium chip. Now compositional proof methods are reported which indicate that such verification tasks are being used for medium-size examples. Also in the case of the verification of *infinite state systems*, it therefore makes sense to complement the above mentioned combination of automatic and theorem-proving-based verification techniques by compositional reasoning supported by semi-automated proof checking methods such as, PVS [ORSvH95], as reported in [Hoo95, HvR97, Sha98a], and other methods, such as ACL2 [KM94], Nqthm [BM97], the Larch Prover [GG91], Nuprl [C$^+$86] and Coq [CH88], for which we refer to the contributions of Hooman and of Shankar to [dRLP98]. Hooman and Shankar encode a *semantical* characterisation of compositional verification in PVS, thus motivating the approach followed in Chapters 6, 7 and 8. For more information concerning this topic the reader is referred to the companion volume [HdRP$^+$00].

1.6.4 Compositionality, Completeness and Modularity

In specification the compositionality principle implies a *separation of concerns* between the use of (and reasoning about) a module, and its implementation. As Lamport says [Lam83c], it provides a *contract* between programmer and implementor:

- One can program a task by the use of modules of which one only knows their specification *without any knowledge about their implementation,* and
- one can implement a module solely on the basis of satisfying its specification *without any knowledge of its use.*

This separates *programming with modules on the basis of their specifications* from *implementing a module such that its specification is satisfied*.

To succeed in this contract between programmer and implementor, all those aspects of program execution which are required to define the effect of a construct must be explicitly addressed in semantics and assertion language alike, that is, one should be able to specify the observable behaviour of such modules. And this implies that one needs at least a compositional framework. This is especially required when specifying fixed components, which can be taken ready-made from the shelf, so to say.

In order to understand the exact relationship between the compositionality principle and Lamport's view of a specification as a contract between programmer and implementor, in [Zwi89, ZHLdR95] the terminology regarding compositionality is refined by distinguishing *compositionality,* which is needed when developing a program from its specification in top-down fashion, from *modularity* (also called *modular completeness* in [Zwi89]), which is needed when constructing programs from re-usable parts with fixed proven specifications in bottom-up fashion. Similar distinctions are made in the work of Abadi and Lamport [AL93, AL95].

Thus, *compositionality* refers to a *top-down approach,* stating that for every assertional specification of a compound construct *there exist* assertional specifications for its components such that these specifications imply the original specification without further information regarding the implementation of those components. The principle of *modularity* refers to a *bottom-up approach,* and requires that, whenever a property of a compound statement follows from *"a priori" given specifications of its fixed components,* then this should be deducible in the proof system in question. As proved in [Zwi89] this amounts to requiring compositionality plus a complete, so-called, *adaptation* rule (formulated, e.g., in Section 3.5.5).

The simplest known adaptation rule is the consequence rule, familiar from, e.g., Hoare and temporal logics. In general, given the fact that a certain *given* specification φ has been proved of a *given* component C, adaptation rules state when to "plug" this proof into another proof context in which another property, say ψ, of that same component C is needed. Completeness of such a rule amounts to requiring that, whenever some requested property ψ of C is implied by the already established specification φ of C, then the adaptation rule can be applied, resulting in a proof that component C satisfies the required property ψ.

Intuitively, the problem of establishing completeness of such a rule is as follows. Of course, when a given predicate φ implies another predicate ψ, $\varphi \rightarrow \psi$ can be proved relative to the set of true properties of the domain of interest (it is such a property itself). However, things are not that simple for program specifications. For example, when does validity of one Hoare triple $\{pre\}\ S\ \{post\}$ imply validity of another such triple $\{pre'\}\ S\ \{post'\}$, regardless of the identity of S?[12]

[12] Observe that here we abstract from the identity of S in order to formulate a property between Hoare-style specifications rather than Hoare triples. Formally, this requires introducing variables ranging over program statements [Zwi89].

Certainly not when the implications $pre \to pre'$ and $post \to post'$ hold, as follows immediately from the definitions of valid Hoare triples given in Section 1.5.4. Requiring $pre' \to pre$ and $post \to post'$ to hold, as expressed by the consequence rule, certainly establishes that validity of $\{pre\}\ S\ \{post\}$ implies validity of $\{pre'\}\ S\ \{post'\}$, regardless of the identity of S. However, the consequence rule does not yield a solution which is complete in that there exist pairs of Hoare triples such that validity of the one implies validity of the other, regardless of the identity of S, and yet this cannot be deduced using the consequence rule. This is especially the case for Hoare triples of concurrent program constructs (as discussed in Section 3.5.1), and leads to the introduction of so-called initialisation and auxiliary-variables rules.

This problem becomes even more acute in the case of assume-guarantee formalisms. Because in that case, specifying the observable behaviour of a component requires four predicates, instead of two. For synchronous communication a solution to this problem is given in [ZHLdR95].

Consequently, also when programming solely on the basis of given, fixed, specifications, a compositional framework is required in order to convince oneself of the correctness of one's programs in a formal set-up which has both the potential for application in the large and allows modules to be specified so as to provide a contract between programmer and implementor in Lamport's sense above.

The relationship between compositional proof systems, completeness and modularity is further discussed in a paper by Trakhtenbrot [Tra].

The paper by Finkbeiner, Manna and Sipma [FMS98] presents a formalism for modular specification and deductive verification of parameterised fair transition systems, with an extension to recursion.

1.6.5 On the Complexity of Compositional Reasoning

Take an automaton with 10 states and consider the parallel composition of 80 such automata. This results in a product automaton of 10^{80} states – that is, more states than the number of electrons within our universe!

Consider now the Internet. Certainly only 80 computers operating in parallel is chicken feed; many many more are operating in parallel! As we saw above, the complexity of the product automaton is *exponential* in the number of states of the constituting automata. Consequently, a description of the Internet using an enumeration of "in principle" possible states is completely unrealistic.

This example shows that an analysis using product automata may not be appropriate for analysing the parallel composition of programs (although it sets lower bounds for worst-case complexity). After all, human beings also operate in parallel, and do not perceive their human environment as being composed of a product automaton recording the changes inside that environment.

Using compositional specification methods, the parallel composition of processes can be specified using the *conjunction* of the specification of the separate processes. Hence the complexity of this description is *linear*, rather than exponential, in the number of these specifications.

Next, assume that inside these specifications *disjunctions* are used to list various possibilities and the conditions subject to which these arise. Then, using de Morgan's law about the distribution of disjunctions over conjunctions, such a linear description may still, in the worst case, lead to the same exponential number of separate cases to be considered. So what did we gain?

Return to the analogy with humans, above. What other humans observe are not the changes of state inside their fellow human beings but rather changes in their *(inter)faces* or attitudes, from which a lot of local internal changes have been eliminated. That is, one observes the *observable behaviour* of one's fellow human beings.

Similarly, compositional specifications record changes in observable behaviour between programs (or processes), i.e., the change of externally (i.e., to other programs) visible quantities, and not the change in values of internal quantities, such as local variables, communication along local channels and the like.

This implies immediately that compositional reasoning is most successful when applied to program constructs which allow a maximal amount of hiding of these internal quantities. Observe that this is not the case for pure shared-variable concurrency when no local variables are present.

Also, as already observed, compositional specifications express the *conditions* under which such external changes occur. Now, de Morgan's law still applies w.r.t. distributing disjunctions inside (compositional) specifications over conjunctions between specifications. However, due to the fact that compositional specifications also express the conditions under which the externally observable changes occur, provided the interaction between processes is not too tight, and internal execution mechanisms can be hidden to a considerable extent, only *a manageable amount of such combinations of externally visible state changes* (of the separate processes) needs to be considered, i.e., are *consistent* – one hopes at least so few that the number of consistent combinations has a manageable complexity. And this belief is supported by current programming practice.

The assumption behind this belief is that communicating computers, operating in parallel, are invented by humans, and, since humans cannot cope with exponential complexity, any artificial artifact *which works,* and consists of parallel components, must somehow be possible to characterise using a very much lower than exponential complexity in the number of processes.

1.6.6 Compositional or Noncompositional Methods?

In this section we try to answer the last question mentioned in Section 1.1:

> *Which style of proof is more appropriate, a compositional or a noncompositional one?*

In principle every programming construct and language can be given a denotational, hence compositional, semantics, defined using the language of mathematics. In a *compositional* semantics, the semantics of a compound construct is a function of the semantics of its top-level (constituent) components.[13] Obviously, no other information about the behaviour of a construct thus characterised needs to be provided, because its denotational semantics already contains all the necessary information! Therefore, in principle, reasoning about and specifying a program can always be done compositionally, using an appropriately formalised subset of the language of mathematics as specification language.

However, in verification actually-given correctness proofs are what count, rather than stating that they might be given in principle. As it turns out, formalisms for synchronous constructs have been characterised compositionally with most success, e.g., those used for characterising the behaviour of hardware – which is synchronous, as argued, e.g., by Gérard Berry [BB91, BG88] – as the success of the assume-guarantee paradigm for such formalisms demonstrates [Sha98b].

In general, compositional approaches for synchronous constructs and languages have been a success story, e.g., for their systematic top-down development from their specifications [HdR91, Hoo95, Hoo98, Oss83, Sta85, Jon94, KR93], for their machine-supported verification [Jos93, BGA98, BCC98, BST98, KV98, MR98, Mos98, OD98, PH98, dRLP98], and for structuring the tools providing this support [DJHP98, BB91, Jos90].

Once it comes to tightly-coupled asynchronous systems, such as those based on shared-variable concurrency, e.g., mutual exclusion algorithms, compositional systems are, in general, not so successful as noncompositional proof methods. Although the rely-guarantee paradigm constitutes an important step

[13] In [Old91a] a distinction is made between compositional semantics, in which the semantics of a compound construct is a function of its constituent components, and *denotational* semantics, which is compositional and, additionally, gives a least or greatest fixpoint interpretation to recursively defined programming constructs.

forward in the methodology of program verification, as yet we only know of a few compositionally derived protocols which need unadulterated shared-variable concurrency for their expression, i.e., in which both reading from and writing to the same shared variable is required by more than one process [Stø90a, dBHdR97b, Din99a]. Section 8.4.3 illustrates the compositional verification of a simple mutual exclusion algorithm using the R-G method. Notice that for deriving algorithms involving tightly-coupled shared-variable concurrency, heuristic methods based on the preservation of invariants within the noncompositional framework of Owicki and Gries have been quite successful [FvG99].

Another such situation, in which compositional methods lack success, arises when programs are defined using global invariants combining local quantities which cannot be easily stated as a conjunction of local predicates, see [CM84]. An example of such an invariant is $x+y<z$, where x, y and z are local variables which occur in different processes. Another example is a distributed greatest-common-divisor algorithm due to Nissim Francez and Mickey Rodeh [FR80], the correctness of which is proved in Section 4.4.2 using a noncompositional method.

Actually, such invariants can always be formulated as a conjunction of local quantities, as shown in Example 4.22. However, this leads to a massive multiplication of information, using so-called logical variables; these are variables which do not occur in any program, but only in their specifications.

Also, compositional reasoning does not lead to improvement when the complexity of the compositional semantics of the constructs concerned is simply too high. Rob Gerth and Marly Roncken give an example of such a case in [GR90], in which a compositional semantics is displayed requiring *nine different kinds of observables!* Yet in a number of such cases noncompositional reasoning can still be successfully applied. This holds especially for *object-oriented reasoning* [dB86, GdR86, Ame87, Ger89, dB91a, dB91b, ÁMdB00].

It turns out that noncompositional methods have been quite successful for axiomatising particular forms of communication and synchronisation in existing programming languages, as in CSP [Hoa78], OCCAM [INM88], ADA [Ada83], concurrent PASCAL and other monitor-based languages and object-oriented languages such as JAVA. Apart from the compositional proof methods for CSP and OCCAM, noncompositional methods are often the only ones available with compositional versions developed at a later stage, if at all [GdR84, dB86, GdR86, Ame87, Ger89, Zwi89, dB91a, dB91b, HdR91, dBH92, ÁMdB00]. This gives us yet another incentive to study both noncompositional and compositional proof methods and emphasise their close relationship.

Summarising, compositional reasoning should be applied whenever it successfully solves the problems of specifier, prover and implementor alike – such is the case, for instance, with formalisms for synchronous constructs. However, there are many cases where the formulation of the problem at hand (developing a program preserving a certain invariant), the inherent complexity of the semantics of a program construct (e.g., one requiring nine-tuples for its compositional characterisation), or the tight coupling of the processes concerned (typical for unadulterated applications of shared-variable concurrency), prevents a practical compositional solution. This applies especially to object-oriented reasoning.

1.7 From Noncompositional to Compositional Proof Methods – a historical perspective

No survey of compositional proof methods for program correctness, in particular for establishing the correctness of concurrent programs, is complete without pointing out the fundamental contributions of Edsger W. Dijkstra to concurrent programming and program correctness in general. This concerns, especially, his insistence on developing programs hand-in-hand with their proofs, the very point of compositional correctness proofs. Quoting from the Turing Award Citation read by M.D. McIlroy at the presentation of Dijkstra's 1972 Turing Award lecture [Dij72b]:

> The working vocabulary of programmers everywhere is studded with words originated or forcefully promulgated by E.W. Dijkstra – display, deadly embrace, semaphore, go-to-less programming, structured programming. But his influence on programming is more pervasive than any glossary can possibly indicate. The precious gift that this Turing Award acknowledges is Dijkstra's *style*: his approach to programming as a high, intellectual challenge; his eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; and his illuminating perception of problems at the foundations of program design. He has published about a dozen papers, both technical and reflective, among which are especially to be noted his philosophical addresses at IFIP [Dij62, Dij65a], his already classic papers on cooperating sequential processes [Dij65b, Dij68c], and his memorable indictment of the go-to-statement [Dij68b]. An influential series of letters by Dijkstra have recently surfaced as a polished monograph on the art of composing programs [Dij71].
>
> We have come to value good programs in much the same way as we value good literature. And at the centre of this movement, creating and reflecting patterns no less beautiful than useful, stands E.W. Dijkstra.

Another quotation, from Leslie Lamport's invited talk at the ACM Symposium on Principles of Distributed Computing in 1983, strengthens the impression that Edsger W. Dijkstra should be regarded as the founding father