Concurrency Verification

Cambridge Tracts in Theoretical Computer Science 54

Titles in the series

1. G. Chaitin *Algorithmic Information Theory*
2. L. C. Paulson *Logic and Computation*
3. M. Spivey *Understanding Z*
5. A. Ramsey *Formal Methods in Artificial Intelligence*
6. S. Vickers *Topology via Logic*
7. J.-Y. Girard, Y. Lafont & P. Taylor *Proofs and Types*
8. J. Clifford *Formal Semantics & Progmatics for Natural Language Processing*
9. M. Winslett *Updating Logical Databases*
10. K. McEvoy & J. V. Tucker (eds) *Theoretical Foundations of VLSI Design*
11. T. H. Tse *A Unifying Framework for Structured Analysis and Design Models*
12. G. Brewka *Nonmonotonic Reasoning*
14. S. G. Hoggar *Mathematics for Computer Graphics*
15. S. Dasgupta *Design Theory and Computer Science*
17. J. C. M. Baeten (ed) *Applications of Process Algebra*
18. J. C. M. Baeten & W. P. Weijland *Process Algebra*
19. M. Manzano *Extensions of First Order Logic*
21. D. A. Wolfram *The Clausal Theory of Types*
22. V. Stoltenberg-Hansen, I. Lindström & E. Griffor *Mathematical Theory of Domains*
23. E.-R. Olderog *Nets, Terms and Formulas*
26. P. D. Mosses *Action Semantics*
27. W. H. Hesselink *Programs, Recursion and Unbounded Choice*
28. P. Padawitz *Deductive and Declarative Programming*
29. P. Gärdenfors (ed) *Belief Revision*
30. M. Anthony & N. Biggs *Computational Learning Theory*
31. T. F. Melham *Higher Order Logic and Hardware Verification*
32. R. L. Carpenter *The Logic of Typed Feature Structures*
33. E. G. Manes *Predicate Transformer Semantics*
34. F. Nielson & H. R. Nielson *Two Level Functional Languages*
35. L. Feijs & H. Jonkers *Formal Specification and Design*
36. S. Mauw & G. J. Veltink (eds) *Algebraic Specification of Communication Protocols*
37. V. Stavridou *Formal Methods in Circuit Design*
38. N. Shankar *Metamathematics, Machines and Gödel's Proof*
39. J. B. Paris *The Uncertain Reasoner's Companion*
40. J. Dessel & J. Esparza *Free Choice Petri Nets*
41. J.-J. Ch. Meyer & W. van der Hoek *Epistemic Logic for AI and Computer Science*
42. J. R. Hindley *Basic Simple Type Theory*
43. A. Troelstra & H. Schwichtenberg *Basic Proof Theory*
44. J. Barwise & J. Seligman *Information Flow*
45. A. Asperti & S. Guerrini *The Optimal Implementation of Functional Programming Languages*
46. R. M. Amadio & P.-L. Curien *Domains and Lambda-Calculi*
47. W.-P. de Roever & K. Engelhardt *Data Refinement*
48. H. Kleine Büning & T. Lettman *Propositional Logic*
49. L. Novak & A. Gibbons *Hybrid Graph Theory and Network Analysis*
51. H. Simmons *Derivation and Computation*
52. A. S. Troelstra & H. Schwictenberg *Basic Proof Theory* (Second Edition)
53. P. Blackburn, M. de Rijke & Y. Venema *Modal Logic*
54. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel & J. Zwiers *Concurrency Verification*

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

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore,
São Paulo, Delhi, Dubai, Tokyo, Mexico City

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521806084

First published 2001

A catalogue record for this publication is available from the British Library

[Insert Library of Congress data if available from input material]

ISBN 978-0-521-80608-4 Hardback
ISBN 978-0-521-16932-5 Paperback

Contents

Preface

The subject of this work is the state-based verification of concurrent programs. It is published in two volumes. The leading theme of these volumes is the development of so-called *compositional* techniques for the verification of concurrent programs from noncompositional techniques. There are three reasons for this.

First of all, compositional verification techniques reduce the verification of large programs to the *independent* verification of their parts. Consequently, compositional verification represents one of the hopeful directions for the verification of really large programs.

Secondly, compositional techniques for the verification of concurrent programs originate from noncompositional techniques, and can, therefore, be better understood once a firm grasp of the latter has been obtained.

Thirdly, compositional techniques for program verification are well suited to top-down program development. However, there are many classes of programs for which noncompositional techniques lead to shorter and clearer proofs. Therefore, when faced with the difficult task of proving concurrent programs correct, one needs to master both kinds of techniques.

This explains why in the two volumes both noncompositional and compositional techniques are developed.

This volume offers a self-contained presentation from first principles of the main techniques for the state-based verification of concurrent programs, focussing on proofs of partial correctness, invariance properties and termination. It presents a mathematical semantically-oriented theory for the verification of concurrent programs, which is based on Floyd's inductive assertion method. It covers noncompositional as well as compositional methods, relating Floyd's method to the classical noncompositional Hoare logics for shared-variable concurrency and synchronous message passing, and presents an in-depth analysis of the two main compositional proof methods which are currently used:

Misra & Chandy's assumption-commitment paradigm for synchronous message passing, and Jones' rely-guarantee paradigm for shared-variable concurrency. Also a family of so-called *communication-closed-layer* transformation principles is developed. Whenever applicable, these transformation principles clarify the structure of correctness proofs considerably. This especially holds for certain classes of network protocols, e.g., those described in [Ray88].

The companion volume, *Compositional Theory of Concurrency,* presents a self-contained description of the main compositional Hoare logics for reactive systems as well as for real-time distributed message passing, for both synchronous and asynchronous communication. It illustrates these compositional techniques by correctness proofs for a number of industrially-inspired medium-size applications, and shows how to obtain machine support for the application of these techniques using the Prototype Verification System PVS.

The material contained in the two volumes offers an integrated account of the subject matter of around 15 dissertations and approximately 100 papers, and leads up to state-of-the-art research in the area of compositional methods for concurrent program verification.

Since many themes in the two volumes have not been discussed before in any textbook, we first list the new ones below.

The leading new themes of this volume are:

1. A clear separation between the *mathematical* theory of program verification, which is semantic in nature, and *syntactically*-formulated axiomatic approaches to program verification.
2. A detailed account of the development from *noncompositional* to *compositional* proof methods for concurrent program verification; this development is part of the above-mentioned mathematical theory, and includes comprehensive accounts of the *assumption-commitment* and *rely-guarantee* paradigms.
3. Partial-order based *transformation principles* for concurrency, constituting the *communication-closed-layers paradigm.*

The leading themes of the companion volume are:

4. Compositional Hoare logics for reactive and distributed real-time systems.
5. Applications of these compositional techniques to the verification of *hybrid* systems, *arbitration* and *atomic broadcast* protocols, *control systems for chemical batch processing*, and a *stable storage* medium.
6. Electronic tool support for the correct construction of distributed real-time systems using the *Prototype Verification System* PVS.

Next, each of these themes is briefly discussed.

1. The mathematical theory of concurrency developed in this volume is based on *Floyd's inductive assertion method*. The main advantages we see in such a mathematical theory are:

   (i) it highlights essential concepts without syntactic overhead,
   (ii) most semantic steps can be directly verified by dedicated tool support, and
   (iii) it allows simple soundness and semantic completeness proofs.

   We often mix Floyd's method and Hoare logic, for instance when developing a compositional theory of concurrency in a context of inductive assertions. Then inductive-assertion-based methods for the sequential parts of a program are combined with compositional proof rules for deducing properties of the parallel composition of these parts. For all these methods and logics rigorous proofs of their soundness and (semantic) completeness are given,[1] together with many examples of their application.

2. That the hierarchical decomposition of programs into smaller ones is imperative to master the complexity of large programs is now generally recognised. In 1965 Edsger W. Dijkstra formulated this principle [Dij65b], which consists of *reducing* the development and verification problem of a program to *that of its constituent (i.e., top-level) subprograms* by starting from its top-level specification and *verifying that specification on the basis of the specifications of those subprograms.* The development and verification of the latter then proceed in essentially the same way, until no further decomposition is necessary.

   Essential for this strategy is that the specification of a large program is verified *on the basis of the specifications of its constituent subprograms, only*, i.e., *without any knowledge of the interior construction of those subprograms* [Zwi89]. And this is the principle of *compositional program verification.*

   To make this verification strategy possible, systems and their parts are specified using predicates over only their observable behaviour. 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 these components.

   To be precise, compositional verification that a program P satisfies an assertional specification φ involves two kinds of proof techniques:

[1] In the experience of the senior author literally *every* alleged proof method for concurrency, which reached his desk and had not been proven complete, turned out to be incomplete, and every such proof method, which had not been proven sound, turned out to be unsound.

(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 define *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]

The characteristic feature of a compositional proof rule is its so-called *compositional proof (reduction) 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*!

Observe that the latter is imposed by the condition *"without any knowledge of the interior construction of those subprograms"* in the formulation of compositional program verification.

Consequently, a *compositional proof rule for a program operator* $op^{lang}(P_1, \ldots, P_n)$ has, essentially, the following form:

$$\frac{\text{for } i = 1, \ldots, n,\ P_i \text{ satisfies } \varphi_i,\quad op^{spec}(\varphi_1, \ldots, \varphi_n, \varphi)}{op^{lang}(P_1, \ldots, P_n) \text{ satisfies } \varphi,}$$

where $\varphi_1, \ldots, \varphi_n, \varphi$ express assertional specifications, and $op^{spec}(\varphi_1, \ldots, \varphi_n)$ expresses a *compositional proof (reduction) step*.

Apart from rules such as the above, one needs additional proof rules to *adapt* one specification to another. Since these rules do not depend on the structure of the actual programs involved, they reduce to the case $n = 1$ in the formulation of compositional proof rules above, and can, therefore, also be regarded as compositional proof rules.

Now a *compositional proof method* for establishing program correctness consists of basic rules, dealing with constructs which are not decomposed any further, and compositional proof rules, which deal with constructs which are decomposed, and with adaptation of specifications.

In the present volume this compositional theory is formulated as an extension of Floyd's inductive assertion method, and includes the first textbook treatments of the assumption-commitment (A-C) and rely-guarantee (R-G) paradigms. In general, compositional proof techniques have the advantage

that they allow a systematic top-down development of programs from their specification, which is correct by construction, as illustrated by many examples in this and in the companion volumes. Moreover, the A-C and R-G paradigms allow for the development of *open* systems, i.e., systems whose environment is not yet known, but possibly specified; this is also illustrated.

Another advantage of compositional techniques is that as soon as traditional techniques suffice to establish the correctness of a component of a process P, these techniques can be "plugged into" the compositional proof strategy described above. For instance, in case the components of P can be successfully checked using *automatic verification tools* (e.g., for model checking), the top-level compositional verification of P only requires, additionally, a number of compositional proof steps (in the above sense) to be carried out [KL93, DJS95]. If applicable, this is a very practical way of formally verifying really complex systems.

3. Partial-order-based transformation principles for concurrency unify such different theories as, e.g.,

   (i) Lipton's theory of left and right movers for reasoning about synchronisation primitives [Lip75],
   (ii) Elrad & Francez' principle of "Communication-Closed Layers" for transforming distributed programs [EF82], and
   (iii) Mazurkiewicz' trace theory [Maz89], one of the first satisfactory partial-order theories for reasoning about truly concurrent systems.

   These transformation principles embody the point of view that, in order to explain and clarify how a network of processes functions, as opposed to its mere verification, *the structure of its correctness proof should reflect the structure of its original design process* rather than that of the resulting final program. These principles for explaining and clarifying complex network protocols have been demonstrated on the basis of many correctness proofs, notably for Gallager, Humblet & Spira's distributed spanning-tree algorithm [GHS83].

4. The compositional Hoare logics for reactive systems discussed in the companion volume include, amongst others, Hoare logics for Misra & Chandy's *assumption-commitment* formalism [MC81] for distributed communication, Jones' *rely-guarantee* formalism [Jon83] for shared-variable concurrency, and Pandya & Joseph's *presupposition-affirmation* formalism [PJ91], which are all proved to be sound (completeness proofs are only discussed in this volume).

   Moreover, a logic is presented for reasoning compositionally about distributed real-time reactive systems, which is subsequently applied and pro-

vided with PVS-based machine support, as described in the second half of the companion volume.

5. In the last three chapters of that volume the emphasis is on illustrating the systematic top-down derivation of distributed programs from their specifications. The resulting programs are correct by construction. This process of program derivation is therefore called the *verify-while-develop paradigm*. This paradigm can be regarded as the "inversion" of a compositional proof method, in which a development step coincides with the application of a compositional rule in which its conclusion and hypotheses are interchanged. 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 above definition of compositional proof techniques we stipulated that in a compositional proof step "no additional knowledge about the underlying execution mechanism of the relevant parts is allowed". For, without that clause, reasoning about a particular design step might have involved reasoning about future design steps, and we want these two stages to be independent. This also explains why compositional techniques can be viewed as the proof-theoretical analogue of hierarchically-structured program development.
6. A compositional formalism is developed for specifying distributed real-time systems, in which program constructs and assertional specifications are combined within a single "mixed" framework. This formalism is formally defined within PVS. The resulting machine-checked theory supports systematic top-down derivation of distributed algorithms, as explained above, and is illustrated by the correct construction of a distributed real-time control system for chemical batch processing.

   This does not imply that machine-checked proofs are free from error. E.g., an error might be introduced by the use of inconsistent axioms, a bug in the proof-checker or a bug in the underlying hardware. Yet the probability of such errors leading to the machine outputting that a faulty system is correct, is small when compared with hand-checked proofs, and decreases as proof-checkers become more experienced.

Occasionally we refer to properties which in general cannot be proved in a compositional set-up, such as termination properties of semaphores [Pnu77, Pnu85]. These do not constitute the main focus of our work, and are, amongst others, discussed in [Fra86] and Manna & Pnueli's manuscript for the *Progress* book [MP99]. These properties depend on assumptions of so-called

weak and strong fairness, which in essence imply termination of the constructs involved, provided corresponding fairness properties of the underlying scheduling mechanisms are satisfied. As such, weak and strong fairnesses are *abstractions* of *real-time* properties of these mechanisms, since they do not state *when* a construct is executed. Since for practical applications the execution speed of such constructs obviously matters, the principal focus of our companion volume is on applications of compositional theories for real-time concurrency.

The structure of this volume

This volume consists of five parts.

Part I consists of Chapter 1. In this chapter such central questions are answered as:

- 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 is more appropriate, a compositional or a noncompositional one?

In order to answer these questions, three concurrent algorithms are discussed, in increasing order of difficulty: Peterson's mutual exclusion algorithm, a concurrent garbage collector due to Dijkstra, Lamport and others, and a distributed mutual-exclusion algorithm due to Szymanski. This discussion leads up to the conclusion of Dijkstra that *"To believe that correct solutions of such problems can be found without a very careful justification is optimism on the verge of foolishness."*

Next the approach taken in this volume is explained. It is a state-based, property-oriented, dual-language, and semantically-oriented approach. The concept of compositionality is explained in relation to the verify-while-develop paradigm, machine verification, and the problem how to specify program modules; also the complexity of compositional reasoning as well as its advantages and disadvantages are discussed. Chapter 1 ends with a short account of the history of the development from noncompositional to compositional methods for program verification.

Part II consists of Chapters 2–5.

In Chapters 2, 3 and 4 semantical formulations are given of Floyd's inductive assertion method for, respectively, a simple model of sequential programming, that of *sequential transition systems*, for shared-variable concurrency in the style of Owicki & Gries, and for synchronous message passing in the styles of

Apt, Francez & de Roever and of Levin & Gries. This set-up of interpreting the inductive assertion method semantically allows for a simple and intuitive formulation of program verification methods, which is based on the systematic generation of so-called *verification conditions*. These are the basic mathematical properties of the underlying state spaces of a program that should be satisfied for that program to be correct. Therefore, this approach leads to the formulation of a *mathematical basis* for state-based program verification. The focus here is on proving partial correctness, absence of deadlock, absence of runtime errors, and termination properties. All proposed program verification methods are proved to be sound and complete. The examples discussed include the correctness proofs for various mutual-exclusion algorithms, amongst others, that of Szymanski (discussed in Chapter 1) and Lamport's ticket algorithm, and a correctness proof for Francez & Rodeh's distributed greatest-common-divisor algorithm.

In Chapters 2–4 the kind of completeness investigated is that of *semantic* completeness. In semantic completeness proofs one disregards the expressibility of predicates in any formal language, and only focusses on their mathematical content, i.e., one regards them as boolean functions. Chapter 5 investigates the expressibility of these boolean functions in the language of first-order predicate logic over the standard model of the natural numbers, establishing that all boolean functions which have been used within the verification methods discussed in Chapters 2–4 can be expressed in this logic. This leads to so-called relatively-complete syntactical formulations of the main verification methods discussed, in which all valid properties of the natural numbers are assumed to be axioms. These formulations are needed for establishing relative completeness of the Hoare logics discussed in later chapters.

Part III consists of Chapters 6–8. It discusses various compositional proof methods for the verification of concurrent programs, and culminates in comprehensive accounts of Misra & Chandy's assumption-commitment paradigm (in order to reason compositionally about synchronous message passing) and Jones' rely-guarantee paradigm (for reasoning compositionally about shared-variable concurrency) [MC81, Jon81, Jon83] in, respectively, Chapters 7 and 8. Chapter 6 contains a general introduction to the subject of compositionality and concurrency, and in particular, to the A-C and R-G paradigms. For all these methods soundness and (semantic) completeness proofs are given. The examples discussed include a top-down derivation of a distributed priority queue, and correctness of mutual-exclusion as well as array-search algorithms.

Part IV consists of Chapters 9–11, and discusses Hoare logics for the programming models discussed in Part II. Soundness and relative completeness

of these logics is deduced from the corresponding soundness and semantic completeness results from Part II, in combination with the expressibility results from Chapter 5.

In Chapter 9 Hoare logic is presented as a *structured* syntactic formulation of the various inductive assertion methods formulated in Part II, reflecting the algebraic structure of the programs whose proof is given. In particular, the semantics of a program is defined as the semantics of its corresponding transition system; therefore, programs can be regarded as a structured notation for transition systems. Moreover, for every program operator we introduce within the inductive assertion method an operator upon proofs such that the proof of a program displays the same structure as the program itself. As a consequence, a correctness proof of a program in Hoare logic corresponds to a proof using the inductive assertion method for its associated transition system, and vice versa, with the added difference that a proof in Hoare logic now reflects the algebraic structure of the program in question. This leads to an identification of concepts which, until now, were regarded as being different.

In Chapter 10 a so-called *proof-outline logic* is presented for nested shared-variable concurrency. A proof outline is a *systematic annotation* of the control points of a program *with assertions*, which satisfy the verification conditions generated by the inductive assertion method. Because proof outlines annotate program text, they utilise additional information about the underlying execution mechanism, and therefore any logic based on them is noncompositional. They satisfy the following property: whenever the assertions associated with the initial control points of a (concurrent) program are satisfied, and that program is executed, the states arising during its computation satisfy the assertions annotating the control points encountered. Proof-outline logics therefore capture properties of intermediate states arising during program execution. Consequently, they are appropriate for proving correctness of concurrent programs, because the interaction between the processes of such programs depends on the values of their intermediate states. As illustration of these concepts, correctness is proved of the concurrent garbage collector from Chapter 1.

In Chapter 11 we present the noncompositional Hoare logics of Apt, Francez & de Roever and of Levin & Gries for synchronous message passing. The logic of the latter is applied in order to derive a compositional parallel composition rule for synchronous message passing. The examples include a correctness proof of a set-partitioning algorithm.

The last part of this volume is Part V. It consists of Chapter 12, and introduces various communication-closed-layer transformation principles for concurrency (both for shared-variable concurrency and for a simple form of

asynchronous message passing). As already stated, these transformation principles express the point of view that, in order to clarify how a network of distributed processes functions, the structure of its correctness proof should reflect the structure of its original design process rather than that of the resulting final program. Whenever applicable, these principles lead to a much clearer proof structure which has the additional advantage of clarifying the resulting network algorithm. This is illustrated on the basis of correctness proofs for a running example in this volume, that of a simple set-partitioning algorithm, and for the two-phase-commit protocol.

The structure of the companion volume

Next we give a brief description of the structure of our companion volume. The latter can be studied independently from this volume.

The companion volume consists of three parts: Part I presents a concise self-contained introduction to Hoare logics for sequential programming, shared-variable concurrency and synchronous message passing. In Part II various compositional Hoare logics for concurrency are presented, and in Part III a number of industrially-inspired medium-size applications are described together with techniques for machine support for checking compositional verification principles for concurrency using PVS.

Its principal focus is the illustration through many examples of compositional techniques in correctness proofs for concurrent programs. As a result, no attention is paid to completeness proofs (the latter being a focus of the present volume). However, we still give soundness proofs to explain why these techniques work.

Part I consists of Chapters 1–3, and discusses Hoare logics for a simple programming language for guarded commands, and its classical noncompositional extensions to shared-variable concurrency and synchronous message passing.

Part II consists of Chapters 4–7.

Chapter 4 presents a compositional Hoare logic for proving partial correctness of programs which communicate through synchronous message passing and feature nested parallelism. The soundness proof for this logic illustrates the problems which must be solved to obtain such compositional Hoare logics in general, and serves as a contrast with its considerably simpler semantical characterisation within an inductive-assertion-style framework which is presented in Chapter 7 of this volume.

Chapter 5 presents various compositional Hoare logics for reactive programs which communicate through synchronous message passing. The purpose of

reactive programs is primarily to maintain an ongoing relationship with the environment, rather than to produce a final value upon termination [HP85]. These Hoare logics are variants of:

(i) Misra & Chandy's assumption-commitment and Pandya & Joseph's presupposition-affirmation formalisms, and
(ii) so-called trace-invariant logics.

In these logics the communication interface of a process is specified using invariants over the communication histories of that process. The focus here is on proving both invariance properties, such as *deadlock freedom*, and certain *progress* properties, such as chatter freedom and divergence freedom, which can be proved compositionally. *Chatter freedom* ensures that the number of consecutive communications in a computation is finite, whereas *divergence freedom* ensures finiteness of the number of consecutive internal actions. In case the relation between an action and its resulting reaction within a reactive program is expressed by execution of a loop body, it is important that this loop body terminates. To prove this, establishing chatter and divergence freedom of that loop body are important, because together these properties imply that no infinite computations are generated – this is called *weak* total correctness. Once the latter has been proved, establishing total correctness requires additional deadlock (and runtime-error) freedom proofs.

As usual, soundness proofs are provided.

Chapter 6 presents a sound rely-guarantee formalism for reasoning compositionally about shared-variable concurrency.

Semantic versions of these proof systems for establishing partial correctness are proved semantically complete in this volume.

In Chapter 7 an assumption-commitment-style Hoare logic is developed for reasoning compositionally about real-time synchronous and asynchronous message passing.

Part III consists of Chapters 8–10, and has been discussed under points 5 and 6 on page xiv.

Chapter 8 applies the compositional proof system for distributed real-time of Chapter 7 to hybrid systems and bus protocols. In particular, a system for chemical batch processing is derived, as well as a distributed real-time arbitration protocol inspired by the IEEE 896 Futurebus specification.

In Chapter 9:

- a mixed formalism is defined in which programs and specifications are combined to a unified framework,

- subsequently, this mixed formalism is formulated inside PVS (for Prototype Verification System) [ORS92], and
- applied to the machine-supported verification of the chemical batch processing system from Chapter 8, and a membership protocol.

Chapter 10 extends the formalism of Chapter 7 for reasoning compositionally about distributed real-time to fault tolerance. In particular, a correctness proof is given for the reliable storage of data, called stable storage, which uses multiple disks.

Instructions for classroom use

The material in these textbooks has been developed and tried out during classroom use for over 10 years in the Netherlands and Germany: at the Eindhoven University of Technology, at the University of Utrecht, and at the Christian-Albrechts University at Kiel. The courses given on their basis vary from quarter courses of 18 hours duration to semester courses of up to 52 hours duration, and include numerous shorter compact courses given at various schools. Outlines of possible courses are discussed below, using the chapter dependencies given in Figures 0.1 and 0.2.

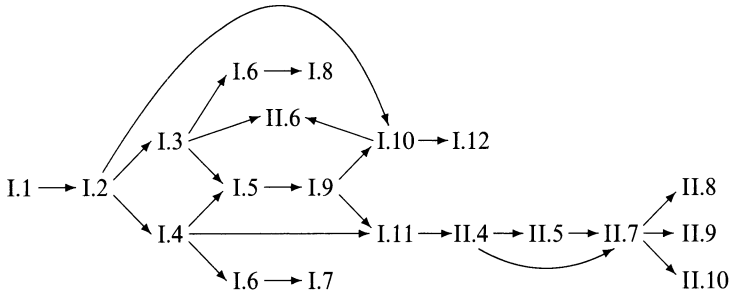

Fig. 0.1. Chapter dependencies: I.j refers to Chapter j of this volume, whereas II.k refers to Chapter k of the companion volume.

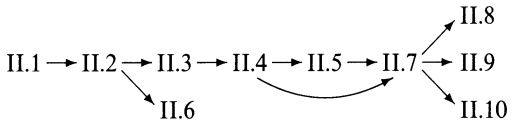

Fig. 0.2. Chapter dependencies for a course exclusively based on the companion volume.

1. An 18-hours introductory course on the mathematical foundations of program verification, covering Chapters I.2, I.3, I.4 and I.9 (possibly dropping the sections on deadlock-freedom, runtime-error freedom and termination proofs).
2. A one-semester course on the mathematical foundations of compositional program verification, covering Parts I, II and III of this volume.
3. A one-semester course on the verification of programs communicating via shared variables, covering Sections 2–6 of Chapter I.1, and Chapters I.2, I.3, (I.5), I.9, I.10, I.12 and/or II.6, discussing Chapter I.5 only in the exercise sessions.
4. A one-semester course on the verification of distributed message passing programs, covering Sections 2–5 of Chapter I.1, and Chapters I.2, I.4, (I.5), I.9, I.11, II.4 and II.5, discussing Chapter I.5 only in the exercise sessions.
5. A course on compositional logics for program verification, exclusively based on the companion volume (which can be studied independently).
6. A 52-hours course on verification of concurrency, covering this volume and Chapters II.4, II.5 and II.6, with or without Chapters I.5 or I.12.

Corrections

It is inevitable that these books contain errors. Please be so kind as to e-mail any errors or remarks to bkmail2@informatik.uni-kiel.de. A list with corrections can be found under: http://www.informatik.uni-kiel.de/inf/deRoever/ .

Technical Notes

Some sections are called *technical notes;* these can be skipped upon first reading, and are aimed at the already technically-skilled reader.

Acknowledgements

This book contains pictures of many researchers whose work is discussed. We thank them for their permission to publish these, and also those people who provided these pictures. In particular, we are grateful to Manfred Broy for putting his picture archive at our disposal; Michael van Emde Boas for his photograph of Peter van Emde Boas and Theo Janssen; Peter van Emde Boas for his shots of Edsger W. Dijkstra and David Park; Sean Floyd for helping us to obtain pictures of his father Robert W. Floyd; Gerhard Pfeifer for his pictures of flying cranes; Henk Thomas, the photographer of Folia, the weekly

of the University of Amsterdam, for his picture of Peter van Emde Boas; the editorial staff of this newspaper for the permission to publish it; and the University of Jena for its picture of Gottlob Frege.

David Tranah and the helpful staff at Cambridge University Press deserve our gratitude for helping us improve and, finally, publish this book. In particular, we owe special thanks to Keith Westmoreland for his careful copy editing and Zoe Naylor for managing to accommodate our request concerning the cover design.

We would like to thank our students and colleagues, especially, Erika Ábrahám-Mumm, Henning Arndt, Roy Bartsch, Kai Baukus, Steve Brooks, Pierre Colette, Antonio Cau, Jaco de Bakker, Edsger W. Dijkstra, Michael Felsberg, Christian Gebken, Oliver Granert, Denise Hodgeson-Möckel, Ralf Huuck, Lasse Kliemann, Lars Kühne, Marcel Kyas, Ben Lukoschus, Jan Lukoschus, Helge Marquardt, Oliver Matz, Jan Meyer, Paritosh Pandya, Jan Paul, Amir Pnueli, Lasse Rempe, Sven Riesenberg, Fred Schneider, Markus Schneider, Carsten Scholz, Dirk Scholz, Natarajan Shankar, Michael Siegel, Karsten Stahl, Änne Straßner, Jan Vianen, Marcus Wieschalla, and Andreas Wortmann for their help and interest, without which writing this book would have been impossible.

We especially thank Ben Lukoschus, without whose TEXpertise the different parts of this book would never have been integrated into a polished monograph with a uniform typography.

Willem-Paul's special thanks go towards his wife Corinne for her professional advice and for creating such a convivial and stimulating atmosphere when receiving his colleagues at home.

Writing this book would have been impossible without the love, support, and understanding of our partners and children during the many years we have been working on it, and without the engagement and support of our teachers. We dedicate this book to all of them.

Christian-Albrechts-Universität zu Kiel	Willem-Paul de Roever
Rijksuniversiteit Utrecht	Frank de Boer
Katholieke Universiteit Nijmegen	Ulrich Hannemann
Katholieke Universiteit Nijmegen	Jozef Hooman
Université Joseph Fourier, Grenoble	Yassine Lakhnech
Universiteit Twente	Mannes Poel
Universiteit Twente	Job Zwiers