Formal Methods: State of the Art and New Directions

Paul Boca • Jonathan P. Bowen
Jawed I. Siddiqi
Editors

Formal Methods: State of the Art and New Directions

Springer

Editors

Dr. Paul Boca
Hornbill Systems Ltd
Ares, Odyssey Business Park
West End Road
Ruislip
United Kingdom HA4 6QD
paul.boca@googlemail.com

Prof. Jawed I. Siddiqi
Sheffield Hallam University
Informatics Research Group
City Campus
Sheffield
United Kingdom S1 1WB
j.i.siddiqi@shu.ac.uk

Jonathan P. Bowen
Museophile Ltd
Oak Barn
Sonning Eye Reading, Oxon
United Kingdom RG4 6TN
jpbowen@gmail.com

ISBN 978-1-84882-735-6 e-ISBN 978-1-84882-736-3
DOI 10.1007/978-1-84882-736-3
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number:2009942261

Cover design: WMXDesign GmbH

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Dedicated to Peter Landin (1930–2009)

Foreword

The Formal Aspects of Computing Science (FACS) Specialist Group of the British Computer Society set up a series of evening seminars in 2005 to report on advances in the application of formal design and analysis techniques in all the stages of software development. The seminars attracted an audience from both academia and industry, and gave them the opportunity to hear and meet pioneers and key researchers in computing science. Normally it would be necessary to travel abroad and attend an international conference to be in the presence of such respected figures; instead, the evening seminar programme, over a period of three years, brought the keynote speakers of the conference to the British Computer Society headquarters, for the convenience of an audience based in London. Several speakers from the period 2005–2007 kindly developed their talks into full papers, which form the basis of this volume.

I am delighted to welcome the publication of such an excellent and comprehensive series of contributions. They are now available in book form to an even wider audience, including developers interested in solutions already available, and researchers interested in problems which remain for future solution.

Sir Tony Hoare

Preface

They envy the distinction I have won; let them therefore, envy my toils, my honesty, and the methods by which I gained it.

— Sallust (86–34 BC)

Formal methods are a powerful technique for helping to ensure the correctness of software. The growth in their use has been slow but steady and they are typically applied in critical systems where safety or security is paramount. Their use requires both expertise and effort, but this is rewarded if they are applied judiciously. Given the pervasiveness of Microsoft products, it must be seen as good news that they are increasingly using formal methods in key parts of their software development, particularly to check the interoperability of third-party software with Windows. We believe formal methods are here to stay and will gain further traction in the future.

In 2005, the British Computer Society Formal Aspects of Computing Science (BCS-FACS) Specialist Group initiated a seminar series. Leading specialists and authorities on the use of formal methods, both in theory and practice, have delivered talks on a variety of topics covering important concerns in formal methods. The contributors have been chosen largely for their expertise and quality of exposition. This book forms a permanent written record of these talks over the first two years of the series. It provides an authoritative collection of views, aimed at advanced researchers and professionals with an interest in the field.

The target readership for this book includes researchers and practitioners in research laboratories, both within academia and industry. It could also serve as advanced MSc level reading material in Computer Science, Informatics and Software Engineering courses with a formal methods component. The range of topics is aimed at researchers, lecturers, graduates and practitioners who are software engineers interested in applying formal methods to software design. Researchers will benefit from a state-of-the-art account in a number of areas of formal methods. Lecturers will gain insight that could be useful background for their courses. Graduates will be exposed to material by leading experts that is intended to be inspirational for their research. Practitioners will discover the current concerns of academia in this area. All the chapters are designed to be self-contained and may be read separately if desired, and they are intended to provide a broad introduction together with some more in-depth material.

This book covers many aspects of the use of formality in computer science. Issues such as requirements, specification, design, applications, modelling, handling complexity and understanding the meaning of programs are presented, each by an expert in their field. This illustrates the various different levels of abstraction that can benefit from a formal approach. No one technique is best in all circumstances and it is a matter of judgement to decide on the best approach to adopt in a particular situation; this is illustrated here through the range of uses of formality.

We thank all the speakers in the evening seminar series, including those not appearing here in written form. Without their support, encouragement and willingness to deliver seminars, the series would never have succeeded. We thank the reviewers of the papers for their timely and insightful comments: Egon Börger, Richard Bornat, Michael Butler, Muffy Calder, John Cooke, Tim Denvir, Mark d'Inverno, Chris George, Martin Henson, Joe Kiniry, Peter Landin,[1] Farhad Mehta, Peter Mosses, Mike Poppleton, Steve Reeves, Wolfgang Reisig, Abdolbaghi Rezazadeh, Peter Ryan, Simon Thompson, Charles Wallace and Margaret West.

The editors thank the staff at Springer for their support and encouragement during the production of this book. Beverley Ford helped in the establishment of the book and Helen Desmond was very patient and helpful.

Some of the evening seminars are held in collaboration with other organizations, including the BCSWomen Specialist Group, Formal Methods Europe, and the London Mathematical Society. We are grateful to these organizations for their continued support. We thank Praxis High Integrity Systems for sponsoring the seminars, and the British Computer Society for providing the splendid central London venue in Covent Garden. The seminar series continues, with around six seminars arranged per year.

Please do visit `http://www.fmsand.info` for information on upcoming seminars and presentation slides. We look forward to seeing readers at future events.

Middlesex, UK — Paul Boca
Oxon, UK — Jonathan P. Bowen
Sheffield, UK — Jawed Siddiqi

[1] Sadly Peter Landin died on 3 June 2009 prior to the publication of this book.

Biographies

Dines Bjørner is a Danish computer scientist, specialising in research into formal methods. He worked with Hans Bekič, Peter Lucas, Cliff Jones and others on the Vienna Development Method (VDM) at IBM in Vienna (and elsewhere). Later he was involved with producing the RAISE (Rigorous Approach to Industrial Software Engineering) formal method with tool support.

Before retiring in March 2007, Bjørner was a professor at the Technical University of Denmark (DTU) in Lyngby. Amongst other things, he was responsible for establishing the United Nations University International Institute for Software Technology (UNU-IIST) in Macau during the 1990s. His magnum opus on software engineering (in three volumes) appeared in 2005/6. To support VDM, Bjørner co-founded VDM-Europe, which subsequently became Formal Methods Europe, an organization that supports conferences and related activities. In 2003, he instigated the associated ForTIA Formal Techniques Industry Association. Dines Bjørner is a knight of the Order of the Dannebrog and was awarded the John von Neumann Medal in Budapest, Hungary in 1994. He is a Fellow of the IEEE (2004) and ACM (2005).

Michael Jackson is a British computer scientist and independent computing consultant in London, England. He is also a visiting research professor at the Open University in the UK, and a visiting fellow at the University of Newcastle.

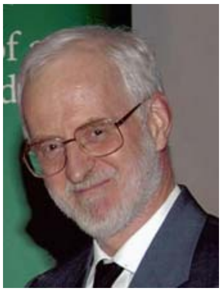
Jackson was educated at Harrow School, where he was taught by Christopher Strachey and wrote his first program under Strachey's guidance. He then studied classics at Oxford University (known as "Greats"), where he was a fellow student with C. A. R. Hoare, two years ahead of him. They had a shared interest in logic, which was studied as part of Greats at Oxford. Subsequently he studied mathematics at Cambridge.

In the 1970s, Jackson developed Jackson Structured Programming (JSP). In the 1980s, with John Cameron, he developed Jackson System Development (JSD). Then, in the 1990s, he developed the Problem Frames Approach. In collaboration with Pamela Zave, he created *Distributed Feature Composition*, a virtual architecture for specification and implementation of telecommunication services.

Jackson received the Stevens Award for Software Development Methods in 1997. In 1998, he received the IEE Achievement Medals Senior Award for outstanding contribution to informatics; in the same year, he was the first recipient of the British Computer Society Lovelace Medal.

Martin Henson is a professor in the School of Computer Science and Electronic Engineering at the University of Essex, UK. Previously he was head of department. His research interests cover formal methods, including logics for specification (especially using the Z notation), refinement and program development. He has been an Honorary Professor of Computer Science at the University of Waikato in New Zealand, where he has worked with Steve Reeves.

Henson was a member of the UK EPSRC-funded RefineNet network. He was programme co-chair of the ZB2002 conference (on the Z notation and the B-Method) and organized a Summer School on Specification Logics with Dines Bjørner in Stara Lesna, Slovakia. More recently, he has worked for the Commission for Academic Accreditation in the United Arab Emirates, helping in the accreditation of computer science degree programmes.



Egon Börger between 1965 and 1971 studied philosophy, logic and mathematics at the Sorbonne, Paris (France), Université Catholique de Louvain and Institut Supérieur de Philosophie de Louvain (Belgium), University of Münster (Germany). He received his Doctoral degree (1971) and Habilitation (1976) in mathematics from the university of Münster. He has been a lecturer and professor in logic and computer science at the universities of Salerno (Italy) 1972–1976, Münster 1976–1978, Dortmund (Germany) 1978–1985, Udine (Italy) 1982/83 and since 1985 Pisa (Italy).

Börger is a pioneer of applying logical methods in computer science. He is co-founder of the international conference series CSL. He is also one of the founders of the Abstract State Machines (ASM) Method for accurate and controlled design and analysis of computer-based systems and co-founder of the series of international ASM workshops. He contributed to the theoretical foundations of the method and initiated its industrial applications in a variety of fields, in particular programming languages, system architectures, requirements and software (re-)engineering, control systems, protocols and web services. He is one of the leading scientists in ASM-based modelling and verification technology, which he has crucially shaped by his activities. In 2007, he received the *Humboldt Research Award*.

Peter Ryan is a professor in the Laboratory of Algorithmics, Cryptology and Security (LACS) at the University of Luxembourg. Prior to taking up this post, he was a professor in the Centre for Software Reliability (CSR) at the School of Computing Science at Newcastle University in the UK. Prior to joining the CSR, Ryan conducted research in formal methods and information assurance at GCHQ, CESG, DERA, SRI Cambridge in the UK and the Software Engineering Institute, Carnegie Mellon University, in the USA. Before moving into information assurance, he was a theoretical physicist and holds a PhD in Theoretical Physics from the University of London for research in quantum gravity.

Ryan's contributions to the field of information assurance include pioneering the use of process algebra to formulate the key security concept of non-interference (absence of information flow) and leading the *Modelling and Analysis of Security protocols* project that demonstrated the effectiveness of process algebra and model checking in the analysis of security protocols. He has made contributions in the area of verifiable voting systems, in particular, creating the Prêt à Voter verifiable election scheme. He is also interested in classical and quantum cryptography.

Muffy Calder has been at the Department of Computing Science, University of Glasgow, Scotland, since January 1988. She was Head of Department of Computing Science for four years, from 2003 to 2007. Previously, she worked at the Departments of Computing Science and Mathematics, University of Stirling and Computer Science, University of Edinburgh. Her research is in modelling and reasoning about the behaviour of complex software and biochemical systems using mathematics and automated reasoning tools.

Calder has a BSc from the University of Stirling and a PhD from the University of St Andrews on *The Imperative Implementation of Algebraic Data Types*. She is a Fellow of the Royal Society of Edinburgh, a Fellow of the Institution of Electrical Engineers and a Fellow of the British Computer Society. She has had short spells in industry at British Ship Research Association and Burroughs Computers, and was a visiting scientist at DEC SRC (Digital Equipment Corporation Systems Research Centre) in California, USA, and a short-term BT Research Fellow at the BT Labs at Martlesham, UK. Calder was a member of the Scottish Science Advisory Committee, which reports to the Scottish Executive, for five years, and currently she is chair of the UKCRC (UK Computing Research Committee). She is a member of TOP (Technical Opportunities Panel) of the EPSRC.

Richard Bornat is a British author and researcher in the field of computer science. He is Professor of Computer Programming at Middlesex University. Previously he was a professor at Queen Mary, University of London. Bornat's research interests include program proving in separation logic. His focus is on the proofs themselves, as opposed to the logical underpinnings. Much of the work involves discovering ways to state the properties of independent modules, in a manner that makes their composition into useful systems conducive.

Bornat, in conjunction with Bernard Sufrin of the Oxford University Computing Laboratory, developed Jape, a proof calculator; he has been involved in research on the usability of this tool for exploration of novel proofs. His PhD students have included Samson Abramsky in the early 1980s. Bornat has published a book entitled *Understanding and Writing Compilers: A Do It Yourself Guide*, which provides extensive resources on compiler development, and more recently a book entitled *Proof and Disproof in Formal Logic: An Introduction for programmers.*

Peter Mosses contributed to the development of denotational semantics in the early 1970s while a graduate student and postdoc in the Programming Research Group, Oxford University. His doctoral studies were supervised by Christopher Strachey; he completed his dissertation (on *Mathematical Semantics and Compiler Generation*) shortly before Strachey's untimely death in 1975.

Mosses moved to Aarhus, Denmark, in 1976. His main contributions in semantics of programming languages have been regarding modularity and tool support. He has developed two frameworks for modular semantic description: action semantics (together with David Watt) and a modular variant of Plotkin's structural operational semantics. He has also contributed to the area of algebraic specification: he has developed a framework called unified algebras, and was overall coordinator of CoFI (the common framework initiative), which developed the algebraic specification language CASL.

Mosses has been a member of IFIP WG 2.2 (Formal Description of Programming Concepts) since 1984, and of WG 1.3 (Foundations of System Specification) since 1994. He returned to the UK in 2005, to a chair at Swansea University, Wales. He recently became a member of the UK Computing Research Committee (UKCRC).

Afterword

The chapters in this volume consist of fundamental contributions in the field from leading researchers; they span a wide variety of concepts, theories, practices, techniques, etc. The simplest manner to view these works is as methods for software construction. For brevity, they have been labelled as formal methods; perhaps, more accurately they could be described as formal aspects of computing science applied to software development – a characterisation that aligns itself to the mission of the FACS specialist group.

In the spirit of simplicity and for our purposes, we can characterise software development into three key stages. First is the early stage of establishing the *requirements*; in some cases, it will also incorporate specification. Secondly, the intermediate stage involves the activities of *design, modelling and implementation.* Finally, the late stage involves *verification and validation.* Such a characterisation has little practical merit, but will suffice, and has been used for the sequencing and pairing of the chapters in this book.

From its inception, two of the key challenges in software engineering, continuing to the present day, are the early "upstream" activity of describing the requirements of the system to be constructed and the associated late "downstream" activity of establishing whether constructed system meets the requirements.

The first pair of papers, respectively by Dines Bjørner and Michael Jackson, both address these activities; they are often embraced within the activity that is termed as requirements engineering, and as such their primary contribution is the establishment of accurate requirements.

Classical engineering, with a long established tradition, subdivides itself into a variety of specialisms such as electronic, mechanical, civil, aeronautical, control, chemical, etc. Yet software engineering, a relatively new discipline, has the daunting, if not impossible, task of constructing software for a whole host of applications, or perhaps more appropriately domains. Is this a desirable situation? Both authors focus on systems that operate in the physical world and they have questioned the merits of this generic approach; they both point out the importance of understanding and describing the domains for which we are required to build systems.

Bjørner's paper addresses these two key challenges directly by advocating domain engineering via three key concepts: domain, the main description and domain model. A domain can be viewed simply as an application or perhaps more accurately as the universe of discourse. The domain description is both an informal narrative as well as a mathematical text that formalises the description. The model is simply that which is designated by the domain description. The hallmark of this approach is the benefits it affords to both verification and validation. When developers wish to prove properties, assert predicates and build theories over the domain (i.e., perform verification), then the model can be viewed as a set of mathematical structures that satisfies the formal description. However, when developers wish to consider aspects of the system as it operates in the real world (i.e., perform validation), they could ask stakeholders whether the description of the system corresponds to their conceptualisation/view of it in the domain that is designated; i.e., does it correspond to the real world sufficiently accurately for the purpose at hand?

The ubiquitous characterisation due to Barry Boehm of verification as "are we building the system right?" and validation as "are we building the right system?" has been repeatedly used to reinforce the distinction between the two.

Jackson's paper picks up the term verification, which he sees as establishing a formal relationship between two formal descriptions, in contrast to validation, which he views as establishing whether a machine that meets the specification is the machine we would like to have. He considers it more fruitful to focus on their similarities in that they both reason about formal descriptions; this accords with Bjørner's view. Jackson, however, argues that formal reasoning inevitably relies on assumptions and that descriptions only reason about approximations to the realities they describe. He intentionally introduces the term "wide verification" to provoke recognition that validation conducted in pursuit of system dependability can and should exploit the same kind of reasoning as verification.

The theme of Jackson's chapter is to characterise the nature of this wide verification, particularly as it relates to the system dependability, because, as he argues, we are not simply in the business of reasoning about the software, but also about the environment in which the system operates. He does this through the work he has established on problem frames, a hallmark of which is to assist the developer in describing the system by putting a sharp focus on the problem domain.

Wide verification corresponds to establishing whether a system operating in the physical world (in Bjørner's terms the domain) satisfies the requirements. Both Bjørner and Jackson advocate formalisation of the domain; however, for Jackson such a formalisation is necessarily an approximation based on certain fundamental assumptions about the physical world. Like Bjørner, he advocates the view that the reasoning can be carried out using techniques similar to those used in verification.

Both Jackson's and Bjørner's works focus on the sort of system in which software plays a central part but other parts such as humans together with natural and engineered aspects are all drawn from the environment; for this reason, the notion of dependable systems must go beyond the properties of the software alone to considerations of its operation (such as safety) in the real world.

In practical terms, Bjørner's and Jackson's approaches share common concerns. Bjørner's domain engineering and Jackson's problem frames and wide verification both enable the developer to describe aspects of the real world, as well as provide a sound basis for formal analysis of the requirements, to address issues of both verification and validation. They focus attention on the fact that systems operate in the physical world and that it is imperative to capture a description of this real world formally at an appropriate level of abstraction. In so doing, any reasoning about the system blurs the distinction between verification and validation. It does not restrict itself to the narrow focus of verification on correctness but to wider properties that are pertinent in the physical world. Neither is it a call for the validation activity to replace verification, but for contributions from both activities to enable developers to capitalise on the strengths of each.

The next two chapters by Martin Henson and Egon Börger focus on an intermediate activity in that they propose methods for software design through the construction of, in Henson's case, specifications and, in Börger's case, models.

Henson describes the specification logic $\nu\mathtt{Z}$, which follows the traditional benefits of formal methods in that the specification (i.e., the formal description) is constructed so as to enable the developer to reason about the specification. He asserts that $\nu\mathtt{Z}$ specifications, unlike traditional Z specifications, are not simply definitions, but that they are theories created so that the specification "correctness" can be addressed as soon as it is created. He examines simple applications through a series of case studies that illustrate the methodological differences between Z and $\nu\mathtt{Z}$, particularly to highlight the deficiencies in the former and the strengths of the latter. Specifically, $\nu\mathtt{Z}$ encourages the definition of operators, which is almost impossible in Z, hence encouraging structuring based on not only "horizontal" modularity, but also "vertical" modularity. The methodological principles established are used in depth on a couple of examples to examine the differences between Z and $\nu\mathtt{Z}$. He argues the characterisation in $\nu\mathtt{Z}$ is closer to a property-based rather than a model-based classification and hence the ability to reason about specifications themselves by exploring properties is facilitated through the theories constructed in $\nu\mathtt{Z}$.

The chapter by Börger in terms of categorisation fits firmly across the whole intermediate stage: it bridges the gap from the problem characterisation to its implementation. Börger's Abstract State Machine (ASM) approach provides an integrated framework for the developer to choose and use appropriate combinations of concepts, notations and techniques. It focuses primarily on modelling through the construction of a ground model that can be refined via design decisions based on real-world constraints and properties. The subsequent analysis that is carried out can be viewed in terms of two major components: verification and validation. Börger illustrates the application of this integrated framework through three examples to construct ground models from which a number of refinement steps and their associated analysis, based on verification, can be carried out to prove correctness. This integrated framework claims the benefit of supporting developers via a multistep process, at the same time allowing them freedom to select any mathematical concepts, notations and techniques that can be combined in a semantically coherent manner.

The next two chapters both address the intermediate stage of modelling and the associated validation. The common features of these are that their approach involves an eclectic combination of formal techniques; a further similarity is that they both apply their approach to novel complex real-world problems.

Peter Ryan's chapter addresses the challenging problem of designing an electronic voting system that can provide high level of auditability, accuracy and privacy (ballot secrecy), with minimal trust assumptions. Like all socio-technical systems, the problem is rarely purely technical. There is an immediate tension and hence the need for a trade-off between auditability and secrecy, while maintaining accuracy. The importance of the validity of the outcome of the election process is paramount, whether electronic or otherwise; therefore, system integrity (or more precisely accuracy) is a primary concern. The chapter not only provides an excellent exposition of the technicalities of human-centred issues involved in electronic voting, but also even more importantly ends up raising vital socio-technical questions that are at the heart of such complex systems.

Ryan's approach to accuracy is through public key cryptography; security in such schemes depends on factorisation of large primes, where availability and secrecy are achieved via a voter verifiable scheme, both of which are elaborated on within the chapter. He shows how to develop and reason about high assurance (accurate) and transparent (voter verifiable) voting schemes; this achieved through a series of proposals that are assessed to ascertain how these schemes are impacted by various threats. The presentation is augmented by a discussion of the enhancements required to counter these threats.

The next chapter by Muffy Calder et al. is about formal modelling and reasoning that has been developed and applied to engineered communications systems such as networks and protocols. However, the authors have found their techniques suitable for modelling and reasoning about naturally evolved networks such as biochemical pathways – a huge complex real-world system. Their approach is based on probabilistic process algebra (PEPA), which provides a comprehensive means to model and reason about systems. Their modelling technique uses formal languages allowing the developer to model the system using a high-level abstraction, which has formal mechanisms for composition, communication and interaction. Two principal benefits accrue from this: the accuracy of the problem description and the ability to be able to reason about them.

The hallmark of Calder's method is that the reasoning carried out is transparent as opposed to the traditional approach involving quantitative methods based on ordinary differential equations. The traditional approach, involving ordinary differential equations to model pathways, does not provide an explicit correspondence between the topology and the model, whereas in the algebraic formulation of the model of the pathways, there is a direct correspondence between that topology and the model; hence, the interactions between the pathway components are clear and they are easier to derive and modify. Calder et al. demonstrate the effectiveness of their approach and its application in recasting a much-cited published case study that had been "solved" using ordinary differential equations in terms of their approach; their analysis revealed an error, primarily because they were able to reason about the constructed model from different perspectives.

Both Ryan's and Calder's papers demonstrate the effectiveness of applying established formal techniques involving modelling and reasoning about modelling to two complex and novel applications. In Ryan's case of the challenging application of electronic voting, this is a human-centred system that requires meeting both social and technical needs. In contrast, Calder's novel application of modelling biochemical signals provides a radical alternative computational discrete model perspective on problems that are traditionally addressed using well-established techniques of continuous differential equations.

These two chapters provided an exposition of the power of formality to conquer two challenging complex real-world problems. Similarly, the chapters by Bornat and Mosses also address complex problems, but within the discipline of computer science. Bornat's paper concerns the problem of designing and reasoning about concurrent and parallel programs. Programming, which is at the heart of computer science, continues to be a challenging activity, and concurrent and parallel programming is especially challenging. Mosses' chapter focuses on programming languages. He points out that while the syntax of programming languages can be described quite precisely using formal grammars, describing the formal semantics still presents many challenges.

Computer scientists are familiar with the difficulties associated with concurrent programming and appreciate the benefit of Dijkstra's simple dictum of keeping processes independent. Nevertheless, they also realise that once these processes communicate, difficulties due to problems of deadlock, livelock, safety, etc., naturally arise. Bornat's chapter explores how communication should happen properly and his choice of formalism to address proper communication of concurrent processes is "separation logic". He does this through the development of Hoare logic adapted to deal with concurrency and pointers; however, because it is a partial correctness logic, it cannot deal with liveness and termination.

One can say that pointers are to data structures what the goto statement is to control structures. However, while the use of goto statements has largely disappeared from programming practice, pointers have proved to be more persistent and useful. Because of this, it is important to pay attention to the semantics of pointers so that we can reason about them effectively. Bornat's chapter provides a selective exposition of separation logic on a series of problems associated with pointers and concurrency, involving the use of the frame rule to reason about pointers, disjoint concurrency rule to reason about binary trees, and conditional critical regions to reason about a pointer transferring buffer program. All these examples, however, involve heap cells rather than variables; in order to address this, Bornat enhances separation logic by treating semaphores as variables to deal with the classical problem of readers and writers.

From the illustrative examples in which Bornat applies and enhances separation logic, he presents us the building blocks for its use to reason about the challenging problem – a hot topic in computer science – of non-blocking concurrency, and in particular to Simpson's four-slot algorithm.

Mosses' focus, similar to Bornat's, is essential to programming because understanding the intended meaning of a programming language is the key to understanding programs. This requires us to have a description of a programming language or, to be more precise, the formal syntax and semantics. For Mosses, the motivation is obvious: without this description the programmer will not be able to reason about the behaviour of a program. Describing the syntax of programming languages has been comprehensively addressed by formal grammars; however, their semantics still remains a challenge.

Denotational semantics is a long-established theoretical approach to the formal description of program behaviour. However, in practice, it has drawbacks and other approaches have been advocated; these include operational semantics, the use of monads and the technique advocated by Mosses, that of constructive semantics. Mosses provides a comprehensive review of each of these approaches to the semantic description of programming languages.

Mosses does not confine the scope of his work to the provision of a formal description; he has the more ambitious task of an online repository of such descriptions. Moreover, constructive semantics has the following major differences/advantages: it allows the semantics of individual constructs to be formulated independently and provides explicit large-scale reuse of the independent parts, as well as supporting modular structuring. These properties yield significant benefits. For the online repository, commonly occurring constructs could be used to index existing language descriptions, as well as forming the basis for composing descriptions. The effort required to develop complete language descriptions can be reduced significantly because descriptions of common constructs are already available and can simply be cited; only those for novel constructs that are not already present in the repository remain to be described.

Much of this Afterword has presented material from the chapters in this book in a condensed form. Any misrepresentations of the ideas of the chapter authors or implied connections between the works are solely the responsibility of the editors.

Middlesex, UK	Paul Boca
Oxon, UK	Jonathan P. Bowen
Sheffield, UK	Jawed Siddiqi

Contents

1

Domain Engineering

Dines Bjørner

Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com

Abstract Before software can be designed we must know its requirements. Before requirements can be expressed we must understand the domain. So it follows, from our dogma, that we must first establish precise descriptions of domains; then, from such descriptions, "derive" at least domain and interface requirements; and from those and machine requirements design the software, or, more generally, the computing systems.

The preceding was an engineering dogma. Now follows a science dogma:

Just as physicists have studied this universe for centuries (and more), and will continue to do so for centuries (or more), so it is about time that we also study such man-made universes as air traffic, the financial service industry, health care, manufacturing, "the market", railways, indeed transportation in general, and so forth. Just in-and-by-themselves. No need to excuse such a study by stating only engineering concerns. To understand is all. And helps engineering.

In the main part of this chapter, Sect. 1.4, we shall outline what goes into a domain description.[1] We shall not cover other domain stages, such as stakeholder identification (etc.), domain acquisition, analysis of domain acquisition units, domain verification and domain validation. That is, before we can acquire domain knowledge, we must know what are suitable structures of domain descriptions. Thus, we shall outline ideas of modelling: (1) the intrinsics (of a domain), (2) the support technologies (of ...), (3) the management and organisation (of ...), (4) the rules and regulations (including [licence or contract] scripts) (of ...) and (5) the human behaviours (of a domain).

Before delving into the main part we shall, however, first overview what we see as basic principles of describing phenomena and concepts of domains.

At the basis of all modelling work is abstraction. Mathematics is a reliable carrier of abstraction. Hence, our domain modelling will be presented as informal, yet short and precise, that is, both concise narratives as well as formal specifications.

[1] We shall use the following three terms: description when we specify what there is (as for domains), prescription when we specify what we would like there to be (as for requirements) and specification when we specify software (or hardware) designs.

1.1 Introduction

1.1.1 Application Cum Business Domains

We shall understand "domain" as "application (or business) domain": a "universe of discourse", "an area of human and societal activity" for which eventually some support in the form of computing and (electronic) communication may be desired. Once computing and communication, that is, hardware and software, that is, a machine, has been installed, then the environment for that machine is the former, possibly business process re-engineered,[2] domain and the new domain includes the machine. The machine interacts with its possibly business process re-engineered (former) domain. But we can speak of a domain without ever thinking, or having to think about, computing (etc.) applications.

Examples of domains are: (1) air traffic, (2) airports, (3) the financial service industry (clients, banks, brokers, securities exchange, portfolio managers, insurance companies, credit card companies, financial service industry watchdogs, etc.), (4) freight logistics, (5) health care, (6) manufacturing and (7) transportation (air lines, railways, roads (private automobiles, buses, taxis, trucking), shipping).

These examples could be termed "grand scale" and reflect infrastructure components of a society. Less "grand" examples of domains are: (8) the interconnect-cabling and the electronic and electro-mechanical boxes of, for example, electronic equipment (9) the interlocking of groups of rail points (switches) of a railway station (10) an automobile (with all its mechanical, electro-mechanical and electronic parts and the composition of all these parts, and with the driver and zero, one or more passengers) or (11) a set of laws (or just rules and regulations) of a business enterprise.

In all of these latter cases, we usually include the human or technological monitoring and control of these domains.

1.1.2 Physics, Domains and Engineering

Physics has been studied for millenia. Physicists continue to unravel deeper and deeper understandings of the physically observable universe around us. Classical engineering builds on physics. Every aeronautical and aerospace, or chemical, or civil, or electrical and electronics, or mechanical and control engineer is expected to know all the laws of physics relevant to their field and much more, and is expected to model, using various branches of mathematics (calculus, statistics, probability theory, graph theory, combinatorics, etc.) phenomena of the domain in which their engineering artifacts are placed (as well as, of course, these artifacts themselves).

[2] By "business process re-engineered domain", we mean a domain where some facets have been altered without the changes containing computing or communication. The changes will normally involve interfacing to the machine being sought.

Software engineers sometimes know how to model their own artifacts (compilers, operating systems, database management systems, data communication systems, web services, etc.), but they seldom, if ever, are expected to model and they mostly cannot, i.e., do not know how to model, the domain in which their software operates.

1.1.3 So, What is a Domain?

We shall use the three terms "domain", "domain description" and "domain model" quite extensively in this chapter.

Above we briefly characterised the term "domain". By a domain, we shall understand the same as an application (or business) domain: a universe of discourse, an area of human and societal activity, etc.

By a domain description, we mean a pair of descriptions: an informal, natural, but probably a professional technical language narrative text which describes the domain as it is and a formal, mathematical text which, supposedly hand-in-hand with the narrative, formalises this description.

By a domain model, we mean that which is designated by a domain description. Two views can be taken here. Either we may claim that the domain description designates the domain, i.e., an actual universe of discourse "out there", in reality. Or we may – more modestly – say that the domain description denotes a possibly infinite and possibly empty set of mathematical structures which (each) satisfy the formal description. The former view is taken when the domain describer is validating the description by asking domain stakeholders whether the description corresponds to their view (conceptualisation) of the domain (in which they work). The latter view is taken when the domain describer is building a domain theory, that is, proves theorems that hold of predicates over the mathematical domain model.

1.1.4 Relation to Previous Work

Considerations of domains in software development have, of course, always been there. Jackson, already in [42] and especially in the work [43–46, 75] leading up to the seminal [47], has again and again emphasised the domain, or, as he calls it, the environment aspects. In classical data analysis – in preparation for the design of data-based information systems – we find a strong emphasis on domain analysis. In the work on ontology, in relation to software engineering, we likewise find this emphasis, notably on the idiosyncratic [67]. We are not claiming that domain analysis as part of our domain engineering approach is new. We claim it is new to put emphasis on describing the domain formally in isolation from any concern for requirements, let alone software. Of course, as one then goes on to develop requirements prescriptions from domain descriptions and software from requirements prescriptions, discrepancies and insufficiencies of the base domain description may very well be uncovered – and must be repaired. We appreciate the approach taken in Jackson's Problem Frame approach [47] in alternating between concerns of domain, requirements and software design.

1.1.5 Structure of the Chapter

In Sect. 1.2, we briefly express the dogma behind the concept of domain engineering and its role in software development. We, likewise, briefly outline basic stages of development of a domain description, i.e., of domain engineering. Section 1.3 outlines an ontology of concepts that we claim appear in any interesting model: entities, functions (or relations), events and behaviours. A new contribution here, perhaps, is our treatment of the modelling of entities: atomic in terms of their attributes and composite in terms of their attributes, their sub-entities and the composition, which we refer to as the mereology of these sub-entities. Section 1.4 forms the major part of this chapter. We present some high level pragmatic principles for decomposing the task of describing a domain – in terms of what we call domain facets – and we illustrate facet modelling by some, what you might think as low level descriptions. Sections 1.3 and 1.4 focus on some aspects of domain abstraction. Section 1.5 comments on a number of abstraction principles and techniques that are not covered in this chapter. Basically this chapter assumes a number of (other) abstraction principles and techniques which we cover elsewhere – notably in [5,6] and [7, Chaps. 2–10]. The first part of Sect. 1.6 very briefly indicates how one may meaningfully obtain major parts of a requirements prescription from a domain description. Section 1.6 also contains a discussion of the differences between domain engineering and requirements engineering. Section 1.7 takes up the motivation and justification for domain engineering.

In this chapter, we primarily express the formalisations in the RAISE [26] specification language, RSL [24]. We refer to [5, 6] for a comprehensive coverage of formal abstractions and models.

Two remarks are now in order. Firstly, there are other specification (cum development or design) languages, such as Alloy [41], ASM [13, 62], B [1], CafeOBJ [19,22,23], CASL [4,16,55], VDM-SL [9,10,21] and Z [37,68,69,74]. But, secondly, none of these suffice. Each, including RSL, has its limitations in what it was supposed to express with ease. So one needs to combine and integrate any of the above formal notations with, for example, the notations of duration calculus (DC) [76,77], message sequence charts (MSCs) [38–40] or live sequence charts (LSCs) [17,35,49], Petri Nets [48,57,59–61], statecharts (SCs) [31–34,36], temporal logic of actions (TLA+) [50,51,54], etc.

The chapters 12–15 of [6] present an extensive coverage of Petri Nets, MSCs and LSCs, SCs and DC, respectively. The present chapter presents an essence of Chaps. 5, 6 and 11 of [7].

The book "Logics of Specification Languages" [8] covers the following formal notations: ASM, B, CafeOBJ, CASL, DC, RAISE, VDM-SL, TLA+

and Z. Reference [8] represents an extensive revision of the following published papers: [15,19,25,37,54,55,62].

•••

A word of warning: This chapter only covers one aspect of domain engineering, namely that of domain facets. There are a number of other aspects of software engineering which underlie professional domain engineering – such as (1) general principles of abstraction and modelling, (2) special principles of modelling languages and systems and (3) other special principles of modelling domains. These are extensively covered in [5,6] and [7, Chaps. 2–10], respectively.

1.2 Domain Engineering: The Engineering Dogma

- *Before software can be designed we must know its requirements.*
- *Before requirements can be expressed we must understand the domain.*
- *So it follows, from our dogma, that we must*
  - *first establish precise descriptions of domains;*
  - *then from such descriptions, "derive" at least domain and interface requirements;*
  - *and from those and machine requirements design the software, or, more generally, the computing systems.*

That is, we propose – what we have practised for many years – that the software engineering process be composed – and that it be iterated over, in a carefully monitored and controlled manner – as follows:

- Domain engineering
- Requirements engineering
- Software design

Each with its many stages and many steps.

We see the domain engineering process as composed from and iterated over the following stages:[3]

1. Identification of and regular interaction with stakeholders
2. Domain (knowledge) acquisition
3. Domain analysis
4. Domain modelling
5. Domain verification
6. Domain validation
7. Domain theory formation

In this chapter, we shall only look at the principles and techniques of domain modelling, that is, item 4. To pursue items 2–3, one must know what goes into a domain description, i.e., a domain model.

[3] The requirements engineering stages are listed in Sect. 1.6.1.

- *A major part of the domain engineering process is taken up by finding and expressing suitable abstractions, that is, descriptions of the domain.*
- *Principles for identifying, classifying and describing domain phenomena and concepts are, therefore, needed.*

This chapter focuses on presenting some of these principles and techniques.

1.3 Entities, Functions, Events and Behaviours

In the domain we observe phenomena. From repeated observations we form (immediate, abstract) concepts. We may then lift such immediate abstract concepts to more general abstract concepts.

Phenomena are manifest. They can be observed by human senses (seen, heard, felt, smelt or tasted) or by physical measuring instruments (mass, length, time, electric current, thermodynamic temperature, amount of substance, luminous intensity). Concepts are defined.

We shall analyse phenomena and concepts according to the following simple, but workable classification: *entities*, *functions* (over entities), *events* (involving changes in entities, possibly as caused by function invocations, i.e., *actions*, and/or possibly causing such), and *behaviours* as (possibly sets of) sequences of actions (i.e., function invocations) and events.

1.3.1 Entities

- *By an* **entity**, *we shall understand something that we can point to, something that manifests, or a concept abstracted from such phenomena or concepts.*

Entities are either atomic or composite. The decision as to which entities are considered is a decision solely taken by the describer.

Atomic Entities

- *By an* **atomic entity**, *we intuitively understand an entity which "cannot be taken apart" (into other, the sub-entities).*

Attributes – Types and Values

With any entity we can associate one or more attributes.

- *By an* **attribute**, *we understand a pair of a* **type** *and a* **value**.

Example 1. Atomic Entities:

Entity: person		Entity: bank account	
Type	Value	Type	Value
Name	Dines Bjørner	Number	212,023,361,918
Weight	118 pounds	Balance	1,678,123 Yen
Height	179 cm	Interest rate	1.5 %
Gender	Male	Credit limit	400,000 Yen

•

"Removing" attributes from an entity destroys its "entity-hood", that is, attributes are an essential part of an entity.

Mereology

- *By* **mereology**, *we shall understand a theory of part-hood relations. That is, of the relations of part to whole and the relations of part to part within a whole.*

The term mereology seems to have been first used in the sense we are using it by the Polish mathematical logician Stanisław Leśniewski [53, 70].

Composite Entities

- *By a composite entity we intuitively understand an entity (1) which "can be taken apart" into sub-entities, (2) where the composition of these is described by its* **mereology** *and (3) which further possess one or more attributes.*

Example 2. Transport Net, A Narrative:

Entity: transport net		
Subentities:	Segments	
	Junctions	
Mereology:	"set" of one or more s(egment)s and	
	"set" of two or more j(unction)s	
such that each s(egment) is delimited by two j(unctions)		
and such that each j(unction) connects one or more s(egments)		
Attributes		
	Types	Values
	Multimodal	Rail, roads
	Transport net of	Denmark
	Year surveyed	2006

•

To put the above example of a composite entity in context, we give an example of both an informal narrative and a corresponding formal specification:

Example 3. Transport Net, A Formalisation: A transport net consists of one or more segments and two or more junctions. With segments [junctions], we can associate the following attributes: segment [junction] identifiers, the identifiers of the two junctions to which segments are connected [the identifiers of the one or more segments connected to the junction] and the mode of a segment [the modes of the segments connected to the junction].

```
type
    N, S, J, Si, Ji, M
value
    obs_Ss: N → S-set,     obs_Js: N → J-set
    obs_Si: S → Si,        obs_Ji: J → Ji
    obs_Jis: S → Ji-set,   obs_Sis: J → Si-set
    obs_M: S → M,          obs_Ms: J → M-set
axiom
    ∀ n:N • card obs_Ss(n) ≥ 1 ∧ card obs_Js(n) ≥ 2
    ∀ n:N • card obs_Ss(n) ≡ card {obs_Si(s)|s:S • s ∈ obs_Ss(n)}
    ∀ n:N • card obs_Js(n) ≡ card {obs_Ji(c)|j:J • j ∈ obs_Js(n)}
    ...
type
    Name, Country, Year
value
    obs_Name: N → Name, obs_Country: N → Country, obs_Year: N → Year
```

Si, Ji, M, Name, Country and Year are not entities. They are names of attribute types and, as such, designate attribute values. N is composite, S and J are considered atomic.[4] •

States

- *By a domain* **state***, we shall understand a collection of domain entities chosen by the domain engineer.*

The pragmatics of the notion of state is that states are recurrent arguments to functions and are changed by function invocations.

1.3.2 Functions

- *By a* **function***, we shall understand something which when applied to some argument values yield some entities called the result value of the function (application).*
- *By an* **action***, we shall understand the same things as applying a state-changing function to its arguments (including the state).*

Function Signatures

By a function signature we mean the name and type of a function.

```
type
    A, B, ..., C, X, Y, .., Z
```

[4] As remarked by a referee: "using cardinality to express injectivity seems obscure, reveals a symptom rather than a cause and is useless in proof". I agree.

```
value
  f: A × B × ... × C → X × Y × ... × Z
```

The last line above expresses a schematic function signature.

Function Descriptions

By a function description, we mean a function signature and something which describes the relationship between function arguments and function results.

Example 4. Well Formed Routes:

```
type
  P = Ji × Si × Ji          /* path: triple of identifiers */
  R′ =  P*                  /* route: sequence of connected paths */
  R = {| r:R′ • wf_R(r) |}  /* subtype of R′: those r′s satisfying wf_R(r) */
value
  wf_R: R′ → Bool
  wf_R(r) ≡
    ∀ i:Nat•{i,i+1}⊆inds r⇒let (,,ji′)=r(i),(ji″,,)=r(i+1) in ji′=ji″ end
```

•

The last line above describes the route wellformedness predicate. [The meaning of the "(,," and ",,)" is that the omitted path components "play no role".]

1.3.3 Events

- *By an* **event**, *we shall understand an instantaneous change of state not directly brought about by some explicitly willed action in the domain, but either by "external" forces or implicitly as a non-intended result of an explicitly willed action.*

Events may or may not lead to the initiation of explicitly issued operations.

Example 5. Events: A "withdraw" from a positive balance bank account action may leave a negative balance bank account. A bank branch office may have to temporarily stop actions, i.e., close due to a bank robbery. •

Internal events: The first example above illustrates an internal action. It was caused by an action in the domain, but was not explicitly the main intention of the "withdraw" function.

External events: The second example above illustrates an external action. We assume that we have not explicitly modelled bank robberies!

Formal modelling of events: With every event we can associate an event label. An event label can be thought of as a simple identifier. Two or more event labels may be the same.

1.3.4 Behaviours

- *By a* **behaviour***, we shall understand a structure of actions (i.e., function invocations) and events. The structure may typically be a set of sequences of actions and events.*

We here refrain from stating whether the "set of sequences" is supposed to model interleaved concurrency, as when we express concurrency in terms of CSP, or whether it is supposed to model "true" concurrency, as when we express concurrency in terms of Petri Nets. Such a statement is required or implied, however, whenever we present a particular model.

A behaviour is either a simple behaviour or is a concurrent behaviour, or if the latter, can be either a communicating behaviour or not.

- *By a* **simple behaviour***, we shall understand a sequence of actions and events.*

Example 6. Simple Behaviours: The opening of a bank account, the deposit into that bank account, zero, one or more other such deposits, a withdrawal from the bank account in question, etc. (deposits and withdrawals), ending with a closing of the bank account. Any prefix of such a sequence is also a simple behaviour. Any sequence in which one or more events are interspersed is also a simple behaviour. •

- *By a* **concurrent behaviour***, we shall understand a set of behaviours (simple or otherwise).*

Example 7. Concurrent Behaviours: A set of simple behaviours may result from two or more distinct bank clients, each operating their own, distinct, that is, non-shared accounts. •

- *By a* **communicating behaviour***, we shall understand a set of two or more behaviours where otherwise distinct elements (i.e., behaviours) share events.*

The sharing of events can be identified via the event labels.

Example 8. Communicating Behaviours: To model that two or more clients can share the same bank account, one could model the bank account as one behaviour and each client as a distinct behaviour. Let us assume that only one client can open an account and that only one client can close an account. Let us further assume that sharing is brought about by one client, say the one who opened the account, identifying the sharing clients. Now, in order to make sure that at most one client accesses the shared account at any one time (in any one "smallest" transaction interval), one may model "client access to account" as a pair of events such that during the interval between the first (begin transaction) and the second (end transaction) event, no other client can share events with the bank account behaviour. Now the set of behaviours of the bank account and one or more of the client behaviours is an example of a communicating behaviour. •

Formal modelling of behaviours: Communicating behaviours, the only really interesting behaviours, can be modelled in a great variety of ways: from set-oriented models in B, RSL, VDM or Z, to models using, for example, CSP (as for example "embedded" in RSL) or to diagram models using, for example, Petri nets, message sequence charts (MSCs) or live sequence charts (LSCs), or statecharts (SCs).

1.3.5 Discussion

The main aim of Sect. 1.3 is to ensure that we have a clear understanding of the modelling concepts of entities, functions, events and behaviours. To "reduce" the modelling of phenomena and concepts to these four is, of course, debatable. Our point is that it works and that further classification, as in, for example, Sowa's [67], is not necessary or rather is replaced by how we model attributes of, for example, entities[5] and how we model facets, as we shall call them. The modelling of facets is the main aim of this chapter.

1.4 Domain Facets

- *By a domain* **facet***, we shall understand one among a finite set of generic ways of analysing a domain: a view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain.*

The barrier here is the "finite set of generic ways". Thus, there is an assumption, a conjecture to be possibly refuted. Namely the postulate that there is a finite number of facets. We shall offer the following facets: intrinsics, support technology, management and organisation, rules and regulations (and scripts) and human behaviour.

1.4.1 Intrinsics

- *By domain* **intrinsics***, we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed earlier and treated, in some detail, below), with such domain intrinsics initially covering at least one specific, hence named, stakeholder view.*

Example 9. Railway Net Intrinsics: We narrate and formalise three railway net intrinsics.

- From the view of *potential train passengers*, a railway net consists of lines, l:L, with names, ln:Ln, stations, s:S, with names sn:Sn, and trains, tn:TN, with names tnm:Tnm. A line connects exactly two distinct stations.

[5] For such issues as static and dynamic attributes, dimensionality, tangibility, time and space, etc., we refer to Jackson's [45] or [7, Chap. 10].

- From the view of *actual train passengers*, a railway net – in addition to the above – allows for several lines between any pair of stations and within stations, provides for one or more platform tracks, tr:Tr, with names, trn:Trn, from which to embark on or alight from a train.
- From the view of *train operating staff*, a railway net – in addition to the above – has lines and stations consisting of suitably connected rail units. A rail unit is either a simple (i.e., linear, straight) unit, or is a switch unit, or is a simple crossover unit, or is a switchable crossover unit, etc. Simple units have two connectors. Switch units have three connectors. Simple and switchable crossover units have four connectors. A path, p:P, through a unit, is a pair of connectors of that unit. A state, $\sigma : \Sigma$, of a unit is the set of paths, in the direction of which a train may travel. A (current) state may be empty: The unit is closed for traffic. A unit can be in any one of a number of states of its state space, $\omega : \Omega$.

A summary formalisation of the three narrated railway net intrinsics could be:

- *Potential train passengers*:

```
scheme N0 =
   class
      type
         N, L, S, Sn, Ln, TN, Tnm
      value
         obs_Ls: N → L-set, obs_Ss: N → S-set
         obs_Ln: L → Ln, obs_Sn: S → Sn
         obs_Sns: L → Sn-set, obs_Lns: S → Ln-set
      axiom
         ...
   end
```

  N, L, S, Sn and Ln designate nets, lines, stations, station names and line names. One can observe lines and stations from nets, line and station names from lines and stations, pair sets of station names from lines and lines names (of lines) into and out from a station from stations. Axioms ensure proper graph properties of these concepts.
- *Actual train passengers*:

```
scheme N1 = extend N0 with
   class
      type
         Tr, Trn
      value
         obs_Trs: S → Tr-set, obs_Trn: Tr → Trn
      axiom
         ...
   end
```

The only additions are that of track and track name types, related observer functions and axioms.

- *Train operating staff*:

```
scheme N2 = extend N1 with
    class
        type
            U, C
            P′ = U × (C×C)
            P = {| p:P′ • let (u,(c,c′))=p in (c,c′)∈ ∪ obs_Ω(u) end |}
            Σ = P-set
            Ω = Σ-set
        value
            obs_Us: (N|L|S) → U-set
            obs_Cs: U → C-set
            obs_Σ: U → Σ
            obs_Ω: U → Ω
        axiom
            ...
    end
```

Unit and connector types have been added as have concrete types for paths, unit states, unit state spaces and related observer functions, including unit state and unit state space observers. The reader is invited to compare the three narrative descriptions with the three formal descriptions, line by line. •

Different stakeholder perspectives, not only of intrinsics, as here, but of any facet, lead to a number of different models. The name of a phenomenon of one perspective, that is, of one model, may coincide with the name of a "similar" phenomenon of another perspective, that is, of another model and so on. If the intention is that the "same" names cover comparable phenomena, then the developer must state the comparison relation.

Example 10. Comparable Intrinsics: We refer to Example 9. We claim that the concept of nets, lines and stations in the three models of Example 9 must relate. The simplest possible relationships are to let the third model be the common "unifier" and to mandate

- That the model of nets, lines and stations of the *potential train passengers* formalisation is that of nets, lines and stations of the *train operating staff* model
- That the model of nets, lines, stations and tracks of the *actual train passengers* formalisation is that of nets, lines, stations of the *train operating staff* model

Thus, the third model is seen as the definitive model for the stakeholder views initially expressed. •

Example 11. Intrinsics of Switches: The intrinsic attribute of a rail switch is that it can take on a number of states. A simple switch $(\,^{c|}\,Y^{c/}_{c}\,)$ has three connectors:

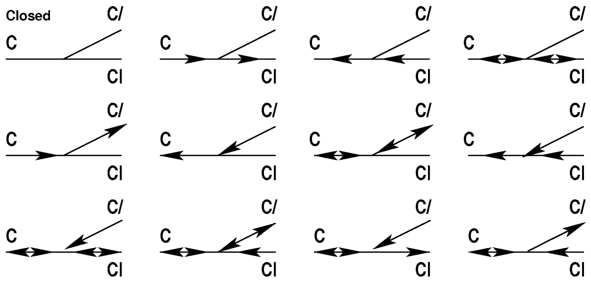


Fig. 1.1. Possible states of a rail switch

$\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$ (Fig. 1.1). So we have that a possible state space of such a switch could be ω_{g_s}:

$\{\{\},$
$\{(c,c_|)\},\{(c_|,c)\},\{(c,c_|),(c_|,c)\},$
$\{(c,c_/)\},\{(c_/,c)\},\{(c,c_/),(c_/,c)\},\{(c_/,c),(c_|,c)\},$
$\{(c,c_|),(c_|,c),(c_/,c)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c),(c,c_|)\},\{(c,c_/),(c_|,c)\}\}$

The above models a general switch ideally. Any particular switch ω_{p_s} may have $\omega_{p_s}\subset\omega_{g_s}$. Nothing is said about how a state is determined: who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (i.e., invisible, intangible) signals up or down the rail, away from the switch. •

Conceptual vs. Actual Intrinsics

In order to bring an otherwise seemingly complicated domain across to the reader, one may decide to present it piecemeal.[6] First, one presents the very basics, the fewest number of inescapable entities, functions and behaviours. Then, in a step of enrichment, one adds a few more (intrinsic) entities, functions and behaviours. And so forth. In a final step, one adds the last (intrinsic) entities, functions and behaviours. In order to develop what initially may seem to be a complicated domain, one may decide to develop it piecemeal. We basically follow the presentation steps: Steps of enrichment – from a big lie via increasingly smaller lies, till one reaches a truth!

[6] That seemingly complicated domain may seem very complicated, containing hundreds of entities, functions and behaviours. Instead of presenting all the entities, functions, events and behaviours in one "fell swoop", one presents them in stages: first, around seven such (entities, functions, events and behaviours), then seven more, etc.

On Modelling Intrinsics

Domains can be characterised by intrinsically being entity or function or event or behaviour intensive. Software support for activities in such domains typically amounts to database systems, computation-bound systems, real-time embedded systems, respectively distributed process monitoring and control systems. Modelling the domain intrinsics in respective cases can often be done in property-oriented specification languages (like CafeOBJ or CASL), model-oriented specification languages (like B, VDM-SL, RSL or Z), event-based languages (like Petri nets or CSP) and respectively process-based specification languages (like MSCs, LSCs, SCs or CSP).

1.4.2 Support Technologies

- *By a domain* **support technology***, we shall understand ways and means of implementing certain observed phenomena or certain conceived concepts.*

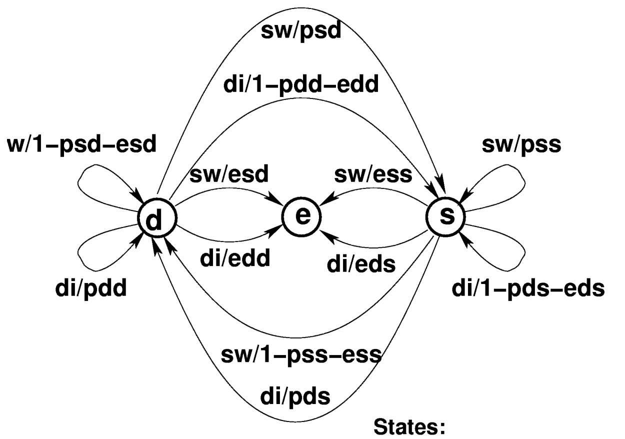
Example 12. Railway Support Technology: We give a rough sketch description of possible rail unit switch technologies.

1. In "ye olde" days, rail switches were "thrown" by manual labour, i.e., by railway staff assigned to and positioned at switches.
2. With the advent of reasonably reliable mechanics, pulleys and levers and steel wires, switches were made to change state by means of "throwing" levers in a cabin tower located centrally at the station (with the lever then connected through wires etc., to the actual switch).
3. This partial mechanical technology then emerged into electromechanics and cabin tower staff was "reduced" to pushing buttons.
4. Today, groups of switches, either from a station arrival point to a station track, or from a station track to a station departure point, are set and reset by means also of electronics, by what is known as interlocking (for example, so that two different routes cannot be open in a station if they cross one another). •

It must be stressed that Example 12 is just a rough sketch. In a proper narrative description, the software (cum domain) engineer must describe, in detail, the subsystem of electronics, electromechanics and the human operator interface (buttons, lights, sounds, etc.).

An aspect of supporting technology includes recording the state-behaviour in response to external stimuli.

Example 13. Probabilistic Rail Switch Unit State Transitions: Figure 1.2 indicates a way of formalising this aspect of a supporting technology. Figure 1.2 intends to model the probabilistic (erroneous and correct) behaviour of a switch when subjected to settings (to switched (s) state) and re-settings (to direct (d) state). A switch may go to the switched state from the direct state when subjected to a switch setting s with probability psd. •



States:
s: Switched state
d: Direct (reverted) state
e: Error state

Input stimuli:
sw: Switch to switched state
di: Revert to direct state

Probabilities: 0 <= p.. <= 1
pss: Switching to switched state from switched state
psd: Switching to switched state from direct state
pds: Reverting to direct state from switched state
pds: Reverting to direct state from direct state
esd: Switching to error state from direct state
edd: Reverting to error state from direct state
ess: Switching to error state from switched state
eds: Reverting to error state from switched state

Fig. 1.2. Probabilistic state switching

The next example shows another aspect of support technology: the technology must guarantee certain of its own behaviours, so that software designed to interface with this technology, together with the technology, meets dependability requirements.

Example 14. Railway Optical Gates: Train traffic (itf:iTF), intrinsically, is a total function over some time interval, from time (t:T) to continuously positioned (p:P) trains (tn:TN).

Conventional optical gates sample, at regular intervals, the intrinsic train traffic. The result is a sampled traffic (stf:sTF). Hence, the collection of all optical gates, for any given railway, is a partial function from intrinsic to sampled train traffics (stf).

We need to express quality criteria that any optical gate technology should satisfy – relative to a necessary and sufficient description of a closeness predicate. The following axiom does that:

> For all intrinsic traffics, itf, and for all optical gate technologies, og, the following must hold: Let stf be the traffic sampled by the optical gates. For all time points, t, in the sampled traffic, those time points must also be in the intrinsic traffic, and for all trains, tn, in the intrinsic traffic

at that time, the train must be observed by the optical gates and the actual position of the train and the sampled position must somehow be check-able to be close or identical to one another.

Since units change state with time, n:N, the railway net needs to be part of any model of traffic.

```
type
  T, TN
  $P = U^*$
  $\mathsf{NetTraffic} == \mathsf{net}{:}\mathsf{N}\ \ \mathsf{trf}{:}(\mathsf{TN} \underset{m}{\rightarrow} \mathsf{P})$
  $\mathsf{iTF} = \mathsf{T} \rightarrow \mathsf{NetTraffic}$
  $\mathsf{sTF} = \mathsf{T} \underset{m}{\rightarrow} \mathsf{NetTraffic}$
  $\mathsf{oG} = \mathsf{iTF} \overset{\sim}{\rightarrow} \mathsf{sTF}$
value
  [close] c: $\mathsf{NetTraffic} \times \mathsf{TN} \times \mathsf{NetTraffic} \overset{\sim}{\rightarrow} \mathbf{Bool}$
axiom
  $\forall$ itt:iTF, og:OG • let stt = og(itt) in
    $\forall$ t:T • t $\in$ dom stt $\Rightarrow$
      $\forall$ Tn:TN • tn $\in$ dom trf(itt(t))
        $\Rightarrow$ tn $\in$ dom trf(stt(t)) $\wedge$ c(itt(t),tn,stt(t)) end
```

Check-ability is an issue of testing the optical gates when delivered for conformance to the closeness predicate, i.e., to the axiom. •

On Modelling Support Technologies

Support technologies in their relation to the domain in which they reside typically reflect real-time embeddedness. As such, the techniques and languages for modelling support technologies resemble those for modelling event and process intensity, while temporal notions are brought into focus. Hence, typical modelling notations include event-based languages (like Petri nets or CSP), respectively process-based specification languages (like MSCs, LSCs, SCs, or CSP), as well as temporal languages (like the duration calculus (DC) and temporal logic of actions (TLA+).

1.4.3 Management and Organisation

Example 15. Train Monitoring, I: In China, as an example, rescheduling of trains occurs at stations and involves telephone negotiations with neighbouring stations ("up and down the lines"). Such rescheduling negotiations, by phone, imply reasonably strict management and organisation (M&O). This kind of M&O reflects the geographical layout of the rail net. •

- *By domain* **management***, we shall understand such people (such decisions) (1) who (which) determine, formulate and thus, set standards (cf. rules and regulations, Sect. 1.4.4) concerning strategic, tactical and operational decisions; (2) who ensure that these decisions are passed on to (lower) levels of management and to floor staff; (3) who make sure that such orders, as they were, are indeed carried out; (4) who handle undesirable deviations in the carrying out of these orders cum decisions and (5) who "backstop" complaints from lower management levels and from floor staff.*
- *By domain* **organisation***, we shall understand the structuring of management and non-management staff levels; the allocation of strategic, tactical and operational concerns to within management and non-management staff levels and hence, the "lines of command": who does what, and who reports to whom, administratively and functionally.*

Example 16. Railway Management and Organisation: Train Monitoring, II: I single out a rather special case of railway management and organisation. Certain (lowest-level operational and station-located) supervisors are responsible for the day-to-day timely progress of trains within a station and along its incoming and outgoing lines, and according to given timetables. These supervisors and their immediate (middle-level) managers (see below for regional managers) set guidelines (for local station and incoming and outgoing lines) for the monitoring of train traffic, and for controlling trains that are either ahead of or behind their schedules. By an incoming and an outgoing line, we mean part of a line between two stations, the remaining part being handled by neighbouring station management. Once it has been decided, by such a manager, that a train is not following its schedule, based on information monitored by non-management staff, then that manager directs that staff (1) to suggest a new schedule for the train in question, as well as for possibly affected other trains; (2) to negotiate the new schedule with appropriate neighbouring stations, until a proper reschedule can be decided upon by the managers at respective stations and (3) to enact that new schedule.[7] A (middle-level operations) manager for regional traffic, i.e., train traffic involving several stations and lines, resolves possible disputes and conflicts. •

The above, albeit a rough-sketch description, illustrated the following management and organisation issues: (1) There is a set of lowest-level (as here: train traffic scheduling and rescheduling) supervisors and their staff; (2) they are organised into one such group (as here: per station); (3) there is a middle-level (as here: regional train traffic scheduling and rescheduling) manager (possibly with some small staff), organised with one per suitable (as here: railway) region and (4) the guidelines issued jointly by local and regional (...) supervisors

[7] That enactment may possibly imply the movement of several trains incident upon several stations: the one at which the manager is located, as well as possibly at neighbouring stations.

and managers imply an organisational structuring of lines of information provision and command.

Conceptual Analysis, First Part

People staff enterprises and enterprises are the components of infrastructures with which we are concerned and for which we develop software. The larger these enterprises – these infrastructure components – the more need there is for management and organisation. The role of management is roughly, for our purposes, twofold. Firstly, to perform strategic, tactical and operational work and to set strategic, tactical and operational policies, and to see to it that they are followed. Secondly, to react to adverse conditions, that is, to unforeseen situations, and to decide how they should be handled, i.e., conflict resolution.

Policy setting should help non-management staff operate normal situations – those for which no management interference is needed. And management acts as a backstop for problems: management takes these problems off the shoulders of non-management staff.

To help management and staff know who is in charge of policy setting and problem handling, a clear conception of the overall organisation is needed. Organisation defines lines of communication within management and staff, and between these. Whenever management and staff have to turn to others for assistance, they usually, in a reasonably well-functioning enterprise, follow the command line: the paths of organigrams - the usually hierarchical box and arrow/line diagrams.

Methodological Consequences

The *management and organisation* model of a domain is a partial specification; hence, all the usual abstraction and modelling principles, techniques and tools apply. More specifically, management is a set of predicate functions, or of observer and generator functions. These either parameterise other, the operations functions, that is, determine their behaviour, or yield results that become arguments to these other functions.

Organisation is, thus, a set of constraints on communication behaviours. Hierarchical, rather than linear, and matrix structured organisations can also be modelled as sets (of recursively invoked sets) of equations.

Conceptual Analysis, Second Part

To relate classical organigrams to formal descriptions, we first show such an organigram (Fig. 1.3) and then schematic processes which – for a rather simple scenario – model managers and the managed!

Based on such a diagram and modelling only one neighbouring group of a manager and the staff working for that manager, we get a system in which

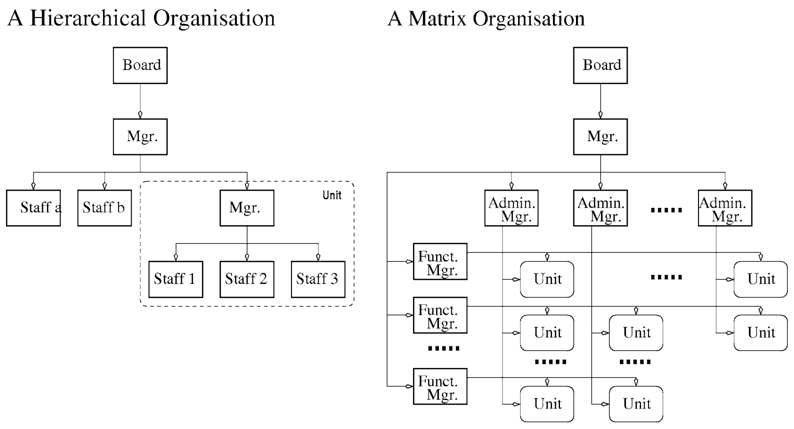


Fig. 1.3. Organisational structures

one manager, mgr, and many staff, stf, coexist or work concurrently, i.e., in parallel. The mgr operates in a context and a state modelled by ψ. Each staff, stf(i), operates in a context and a state modelled by sσ(i).

```
type
    Msg, $\Psi$, $\Sigma$, Sx
    S$\Sigma$ = Sx $\underset{m}{\to}$ $\Sigma$
channel
    { ms[i]:Msg | i:Sx }
value
    s$\sigma$:S$\Sigma$, $\psi$:$\Psi$

    sys: Unit → Unit
    sys() ≡ ∥ { stf(i)(s$\sigma$(i)) | i:Sx } ∥ mg($\psi$)
```

In this system, the manager, mgr, (1) either broadcasts messages, m, to all staff via message channel, ms[i]. The manager's concoction, m_out(ψ), of the message, msg, has changed the manager state; or (2) is willing to receive messages, msg, from whichever staff, i, the manager sends a message. Receipt of the message changes, m_in(i,m)(ψ), the manager state. In both cases, the manager resumes work as from the new state. The manager chooses – in this model – which of the two things (1 or 2) to do by a so-called non-deterministic internal choice ($\sqcap$).

```
    mg: $\Psi$ → in,out {ms[i]|i:Sx} Unit
    mg($\psi$) ≡
1.  (let ($\psi'$,m)=m_out($\psi$) in ∥{ms[i]!m|i:Sx};mg($\psi'$)end)
        $\sqcap$
2.  (let  $\psi'$=$\sqcap${let m=ms[i]? in m_in(i,m)($\psi$) end|i:Sx} in mg($\psi'$) end)
```

```
    m_out: $\Psi \rightarrow \Psi \times$ MSG,
    m_in: Sx $\times$ MSG $\rightarrow \Psi \rightarrow \Psi$
```

And in this system, staff i, stf(i), (1) either is willing to receive a message, msg, from the manager and then to change, st_in(msg)(σ), state accordingly, or (2) to concoct, st_out(σ), a message, msg (thus changing state) for the manager and send it ms[i]!msg. In both cases, the staff resume work as from the new state. The staff member chooses – in this model – which of the two "things" (1 or 2) to do by a non-deterministic internal choice ($\sqcap$).

```
    st: i:Sx $\rightarrow \Sigma \rightarrow$ **in**,**out** ms[i] **Unit**
    stf(i)($\sigma$) $\equiv$
1.  (**let** m = ms[i]? **in** stf(i)(stf_in(m)($\sigma$)) **end**)
        $\sqcap$
2.  (**let** ($\sigma'$,m) = st_out($\sigma$) **in** ms[i]!m; stf(i)($\sigma'$) **end**)

    st_in: MSG $\rightarrow \Sigma \rightarrow \Sigma$,
    st_out: $\Sigma \rightarrow \Sigma \times$ MSG
```

Both manager and staff processes recurse (i.e., iterate) over possibly changing states. The management process non-deterministically, internal choice, "alternates" between "broadcast" issuing orders to staff and receiving individual messages from staff. Staff processes likewise non-deterministically, internal choice, alternate between receiving orders from management and issuing individual messages to management.

The conceptual example also illustrates modelling stakeholder behaviours as interacting (here CSP-like) processes.

On Modelling Management and Organisation

Management and organisation basically span entity, function, event and behaviour intensities and thus, typically require the full spectrum of modelling techniques and notations – summarised in the two "On Modelling ..." paragraphs at the end of the two previous sections.

1.4.4 Rules and Regulations

- *By a domain* **rule**, *we shall understand some text (in the domain) which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their function.*
- *By a domain* **regulation**, *we shall understand some text (in the domain) which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention.*

Example 17. Trains at Stations:

- Rule: In China, the arrival and departure of trains at and from railway stations, respectively, is subject to the following rule:

  In any 3 min interval at most, one train may either arrive to or depart from a railway station.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

•

Example 18. Trains Along Lines:

- Rule: In many countries, railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving along the line, then:

  There must be at least one free sector (i.e., without a train) between any two trains along a line.
- Regulation: *If it is discovered that the above rule is not obeyed*, then there is some regulation which prescribes administrative or legal management and/or staff action, as well as some correction to the railway traffic.

•

A Meta-Characterisation of Rules and Regulations

At a meta-level, i.e., explaining the general framework for describing the syntax and semantics of the human-oriented domain languages for expressing rules and regulations, we can say the following: There are, abstractly speaking, usually three kinds of languages involved when expressing rules and regulations (respectively when invoking actions that are subject to rules and regulations). Two languages, Rules and Reg, exist for describing rules and regulations, respectively; and one, Stimulus, exists for describing the form of the (always current) domain action stimuli.

A syntactic stimulus, sy_sti, denotes a function, se_sti:STI: $\Theta \rightarrow \Theta$, from any configuration to a next configuration, where configurations are those of the system being subjected to stimulations. A syntactic rule, sy_rul:Rule, stands for, i.e., has as its semantics, its meaning, rul:RUL, a predicate over current and next configurations, $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$, where these next configurations have been brought about, i.e., caused, by the stimuli. These stimuli express that if the predicate holds, then the stimulus will result in a valid next configuration.

```
type
   Stimulus, Rule, $\Theta$
   STI = $\Theta \rightarrow \Theta$
   RUL = $(\Theta \times \Theta) \rightarrow \mathbf{Bool}$
```

```
value
    meaning: Stimulus → STI
    meaning: Rule → RUL

    valid: Stimulus × Rule → Θ → Bool
    valid(sy_sti,sy_rul)(θ) ≡ meaning(sy_rul)(θ,(meaning(sy_sti))(θ))

    valid: Stimulus × RUL → Θ → Bool
    valid(sy_sti,se_rul)(θ) ≡ se_rul(θ,(meaning(sy_sti))(θ))[-3pt]
```

A syntactic regulation, sy_reg:Reg (related to a specific rule), stands for, i.e., has as its semantics, its meaning, a semantic regulation, se_reg:REG, which is a pair. This pair consists of a predicate, pre_reg:Pre_REG, where Pre_REG = $(\Theta \times \Theta) \rightarrow$ **Bool**, and a domain configuration-changing function, act_reg:Act_REG, where Act_REG = $\Theta \rightarrow \Theta$, that is, both involving current and next domain configurations. The two kinds of functions express: If the predicate holds, then the action can be applied.

The predicate is almost the inverse of the rules functions. The action function serves to undo the stimulus function.

```
type
    Reg
    Rul_and_Reg = Rule × Reg
    REG = Pre_REG × Act_REG
    Pre_REG = Θ × Θ → Bool
    Act_REG = Θ → Θ
value
    interpret: Reg → REG[-3pt]
```

The idea is now the following: Any action of the system, i.e., the application of any stimulus, may be an action in accordance with the rules, or it may not. Rules, therefore, express whether stimuli are valid or not in the current configuration. And regulations, therefore, express whether they should be applied and, if so, with what effort.

More specifically, there is usually, in any current system configuration, given a set of pairs of rules and regulations. Let (sy_rul,sy_reg) be any such pair. Let sy_sti be any possible stimulus. And let θ be the current configuration. Let the stimulus, sy_sti, applied in that configuration result in a next configuration, θ', where θ' = (meaning(sy_sti))(θ). Let θ' violate the rule, ∼valid(sy_sti,sy_rul)(θ), then if predicate part, pre_reg, of the meaning of the regulation, sy_reg, holds in that violating next configuration, pre_reg(θ,(meaning(sy_sti))(θ)), then the action part, act_reg, of the meaning of the regulation, sy_reg, must be applied, act_reg(θ), to remedy the situation.

```
axiom
    ∀ (sy_rul,sy_reg):Rul_and_Regs •
        let se_rul = meaning(sy_rul),
              (pre_reg,act_reg) = meaning(sy_reg) in
```

```
∀ sy_sti:Stimulus, θ:Θ •
    ∼valid(sy_sti,se_rul)(θ)
        ⇒ pre_reg(θ,(meaning(sy_sti))(θ))
            ⇒ ∃ nθ:Θ • act_reg(θ)=nθ ∧ se_rul(θ,nθ)
end
```

It may be that the regulation predicate fails to detect applicability of regulations actions. That is, the interpretation of a rule differs, in that respect, from the interpretation of a regulation. Such is life in the domain, i.e., in actual reality.

On Modelling Rules and Regulations

Usually rules (as well as regulations) are expressed in terms of domain entities, including those grouped into "the state", functions, events and behaviours. Thus, the full spectrum of modelling techniques and notations may be needed. Since rules usually express properties, one often uses some combination of axioms and wellformedness predicates. Properties sometimes include temporality and hence, temporal notations (like DC or TLA) are used. And since regulations usually express state (restoration) changes, one often uses state changing notations (such as found in B, RSL, VDM-SL and Z). In some cases, it may be relevant to model using some constraint satisfaction notation [2] or some Fuzzy Logic notations [72].

1.4.5 Scripts and Licensing Languages

- *By a domain* **script***, we shall understand the structured, almost, if not outright, formally expressed, wording of a rule or a regulation that has legally binding power, that is, which may be contested in a court of law.*

Example 19. A Casually Described Bank Script: We deviate, momentarily, from our line of railway examples, to exemplify one from banking. Our formulation amounts to just a (casual) rough sketch. It is followed by a series of four large examples. Each of these elaborates on the theme of (bank) scripts.

The problem area is that of how repayments of mortgage loans are to be calculated. At any one time, a mortgage loan has a balance, a most recent previous date of repayment, an interest rate and a handling fee. When a repayment occurs, the following calculations shall take place: (1) the interest on the balance of the loan since the most recent repayment, (2) the handling fee, normally considered fixed, (3) the effective repayment, being the difference between the repayment and the sum of the interest and the handling fee, and the new balance, being the difference between the old balance and the effective repayment.

We assume repayments to occur from a designated account, say a demand/deposit account. We assume that bank to have designated fee and interest income accounts.

(1) The interest is subtracted from the mortgage holder's demand/deposit account and added to the bank's interest (income) account. (2) The handling fee is subtracted from the mortgage holder's demand/deposit account and added to the bank's fee (income) account. (3) The effective repayment is subtracted from the mortgage holder's demand/deposit account and also from the mortgage balance. Finally, one must also describe deviations such as overdue repayments, too large, or too small repayments, and so on. •

Example 20. A Formally Described Bank Script:

First we must informally and formally define the bank state:

There are clients (c:C), account numbers (a:A), mortgage numbers (m:M), account yields (ay:AY) and mortgage interest rates (mi:MI). The bank registers, by client, all accounts (ρ:A_Register) and all mortgages (μ:M_Register). To each account number, there is a balance (α:Accounts). To each mortgage number, there is a loan (ℓ:Loans). To each loan is attached the last date that interest was paid on the loan.

value
  r, r':**Real axiom** ...
type
  C, A, M, Date
  $AY' = \mathbf{Real}$, AY = $\{|$ ay:AY$'$ • $0 < ay \leq r$ $|\}$
  $MI' = \mathbf{Real}$, MI = $\{|$ mi:MI$'$ • $0 < mi \leq r'$ $|\}$
  Bank$'$ = A_Register $\times$ Accounts $\times$ M_Register $\times$ Loans
  Bank = $\{|$ β:Bank$'$ • wf_Bank(β)$|\}$
  A_Register = C $\overrightarrow{m}$ **A-set**
  Accounts = A $\overrightarrow{m}$ Balance
  M_Register = C $\overrightarrow{m}$ **M-set**
  Loans = M $\overrightarrow{m}$ (Loan $\times$ Date)
  Loan,Balance = P
  P = **Nat**

Then we must define well-formedness of the bank state:

value
  ay:AY, mi:MI

  wf_Bank: Bank $\rightarrow$ **Bool**
  wf_Bank(ρ,α,μ,ℓ) $\equiv \cup$ **rng** ρ = **dom** $\alpha \land \cup$ **rng** μ = **dom** ℓ
axiom
  ay<mi [$\land$ ...]

We – perhaps too rigidly – assume that mortgage interest rates are higher than demand/deposit account interest rates: ay<mi.

Operations on banks are denoted by the commands of the bank script language. First the syntax:

```
type
  Cmd = OpA | CloA | Dep | Wdr | OpM | CloM | Pay
  OpA == mkOA(c:C)
  CloA == mkCA(c:C,a:A)
  Dep == mkD(c:C,a:A,p:P)
  Wdr == mkW(c:C,a:A,p:P)
  OpM == mkOM(c:C,p:P)
  Pay == mkPM(c:C,a:A,m:M,p:P,d:Date)
  CloM == mkCM(c:C,m:M,p:P)
  Reply = A | M | P | OkNok
  OkNok == ok | notok
value
  period: Date × Date → Days [for calculating interest]
  before: Date × Date → Bool [first date is earlier than last date]
```

And then the semantics:

```
  int_Cmd(mkPM(c,a,m,p,d))($\rho$,$\alpha$,$\mu$,$\ell$) ≡
    let (b,d′) = $\ell$(m) in
    if $\alpha$(a)≥p
      then
        let i = interest(mi,b,period(d,d′)),
            $\ell'$ = $\ell$ † [m↦$\ell$(m)−(p−i)]
            $\alpha'$ = $\alpha$ † [a↦$\alpha$(a)−p,$a_i$↦$\alpha(a_i)$+i] in
        (($\rho$,$\alpha'$,$\mu$,$\ell'$),ok) end
      else
        (($\rho$,$\alpha'$,$\mu$,$\ell$),nok)
    end end
    pre c ∈ dom $\mu$ ∧ a ∈ dom $\alpha$ ∧ m ∈ $\mu$(c)
    post before(d,d′)

    interest: MI × Loan × Days → P
```

•

The idea about scripts is that they can somehow be objectively enforced: that they can be precisely understood and consistently carried out by all stakeholders, eventually leading to computerisation. But they are, at all times, part of the domain.

Licensing Languages

A special form of script is increasingly appearing in some domains, notably the domain of electronic, or digital media, where these licences express that the licensor permits the licensee to render (i.e., play) works of a proprietary nature, CD ROM-like music, DVD-like movies, etc. while obligating the licensee to pay the licensor on behalf of the owners of these, usually artistic works. We refer to [11,29,58,65] for papers and reports on license languages.

On Modelling Scripts

Scripts (as are licenses) are like programs (respectively like prescriptions program executions). Hence, the full variety of techniques and notations for modelling programming (or specification) languages apply [18,30,63,66,71,73]. Reference [6, Chaps. 6–9] covers pragmatics, semantics and syntax techniques for defining languages.

1.4.6 Human Behaviour

- *By domain* **human behaviour**, *we shall understand any of a quality spectrum of carrying out assigned work: from (1) careful, diligent and accurate, via (2) sloppy dispatch and (3) delinquent work, to (4) outright criminal pursuit.*

Example 21. Banking – or Programming – Staff Behaviour: Let us assume a bank clerk, "in ye olde" days, when calculating, say mortgage repayments (cf. Example 19).

We would characterise such a clerk as being *diligent,* etc., if that person carefully follows the mortgage calculation rules and checks and double-checks that calculations "tally up", or lets others do so. We would characterise a clerk as being *sloppy* if that person occasionally forgets the checks alluded to above. We would characterise a clerk as being *delinquent* if that person systematically forgets these checks. And we would call such a person a *criminal* if that person intentionally miscalculates in such a way that the bank (and/or the mortgage client) is cheated out of funds which, instead, may be diverted to the cheater.

Let us, instead of a bank clerk, assume a software programmer charged with implementing an automatic routine for effecting mortgage repayments (cf. Example 20).

We would characterise the programmer as being *diligent* if that person carefully follows the mortgage calculation rules, and throughout the development, verifies and tests that the calculations are correct with respect to the rules. We would characterise the programmer as being *sloppy* if that person forgets certain checks and tests when otherwise correcting the computing program under development. We would characterise the programmer as being *delinquent* if that person systematically forgets these checks and tests. And we would characterise the programmer as being a *criminal* if that person intentionally provides a program which miscalculates the mortgage interest, etc., in such a way that the bank (and/or the mortgage client) is cheated out of funds. •

Example 22. A Human Behaviour Mortgage Calculation: Example 20 gave a semantics to the mortgage calculation request (i.e., command) as a diligent bank clerk would be expected to perform it. To express, that is, to model, how sloppy, delinquent or outright criminal persons (staff?) could behave, we must modify the int_Cmd(mkPM(c,a,m,p,d′))(ρ,α,μ,ℓ) definition.

```
int_Cmd(mkPM(c,a,m,p,d))($\rho$,$\alpha$,$\mu$,$\ell$) ≡
    let (b,d′) = $\ell$(m) in
    if q($\alpha$(a),p) /* $\alpha$(a)≤p∨$\alpha$(a)=p∨$\alpha$(a)≤p∨... */
        then
            let i = $f_1$(interest(mi,b,period(d,d′))),
                $\ell'$ = $\ell$ † [m↦$f_2$($\ell$(m)−(p−i))]
                $\alpha'$ = $\alpha$ † [a↦$f_3$($\alpha$(a)−p),$a_i$↦$f_4$($\alpha$($a_i$)+i),
                                $a_{\text{“staff”}}$↦$f_{\text{“staff”}}$($\alpha$($a_{\text{“staff”}}$)+i)] in
            (($\rho$,$\alpha'$,$\mu$,$\ell'$),ok) end
        else
            (($\rho$,$\alpha'$,$\mu$,$\ell$),nok)
    end end
    pre c ∈ dom $\mu$ ∧ m ∈ $\mu$(c)

q: P × P $\xrightarrow{\sim}$ Bool
$f_1$,$f_2$,$f_3$,$f_4$,$f_{\text{“staff”}}$: P $\xrightarrow{\sim}$ P  [typically: $f_{\text{“staff”}}$ = λp.p]
```

•

The predicate q and the functions f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ of Example 22 are deliberately left undefined. They are being defined by the “staffer” when performing (including programming) the mortgage calculation routine.

The point of Example 22 is that one must first define the mortgage calculation script precisely as one would like to see the diligent staff (programmer) perform (including correctly program) it before one can “pinpoint” all the places where lack of diligence may “set in”. The invocations of q, f_1, f_2, f_3, f_4 and $f_{\text{“staff”}}$ designate those places.

The point of Example 22 is also that we must first domain-define, “to the best of our ability”, all the places where human behaviour may play other than a desirable role. If we cannot, then we cannot claim that some requirements aim at countering undesirable human behaviour.

A Meta-Characterisation of Human Behaviour

Commensurate with the above, humans interpret rules and regulations differently and not always consistently – in the sense of repeatedly applying the same interpretations.

Our final specification pattern is therefore:

```
type
    Action = $\Theta \xrightarrow{\sim} \Theta$-infset
```

```
value
    hum_int: Rule → $\Theta$ → RUL-infset
    action: Stimulus → $\Theta$ → $\Theta$
    hum_beha: Stimulus × Rules → Action → $\Theta \overset{\sim}{\to} \Theta$-infset
    hum_beha(sy_sti,sy_rul)($\alpha$)($\theta$) as $\theta$set
        post
            $\theta$set = $\alpha(\theta)$ ∧ action(sy_sti)($\theta$) ∈ $\theta$set
             ∧ ∀ $\theta'$:$\Theta$•$\theta'$ ∈ $\theta$set ⇒
                ∃ se_rul:RUL•se_rul ∈ hum_int(sy_rul)($\theta$)⇒se_rul($\theta$,$\theta'$)
```

The above is, necessarily, sketchy: There is a, possibly infinite, variety of ways of interpreting some rules. A human, in carrying out an action, interprets applicable rules and chooses one which that person believes suits some (professional, sloppy, delinquent or criminal) intent. “Suits” means that it satisfies the intent, i.e., yields **true** on the pre/post-configuration pair, when the action is performed – whether as intended by the ones who issued the rules and regulations or not. We do not cover the case of whether an appropriate regulation is applied or not.

The above-stated axioms express how it is in the domain, not how we would like it to be. For that we have to establish requirements.

On Modelling Human Behaviour

To model human behaviour is, “initially”, much like modelling management and organisation but only “initially”. The most significant human behaviour modelling aspect is then that of modelling non-determinism and looseness, even ambiguity. So a specification language that allows non-determinism and looseness (like CafeOBJ and RSL) is preferred.

1.4.7 Completion

Domain acquisition results in typically up to thousands of units of domain descriptions. Domain analysis subsequently also serves to classify which facet any one of these description units primarily characterises. But some such “compartmentalisations” may be difficult and may be deferred till the step of “completion”. It may then be “at the end of the day”, that is, after all of the above facets have been modelled, that some description units are left as not having been described, not deliberately, but “circumstantially”. It then behoves the domain engineer to fit these “dangling” description units into suitable parts of the domain description. This “slotting in” may be simple, and all is fine. Or it may be difficult. Such difficulty may be a sign that the chosen model, the chosen description, in its selection of entities, functions, events and behaviours to model – in choosing these over other possible selections of phenomena and concepts – is not appropriate. Another attempt must be made. Another selection, another abstraction of entities, functions, etc., may need to be chosen. Usually, however, after having chosen the abstractions of the intrinsic phenomena and concepts, one can start checking whether "dangling" description units can be fitted in "with ease".

1.4.8 Integrating Formal Descriptions

We have seen that to model the full spectrum of domain facets, one needs not one, but several specification languages. No single specification language suffices. It seems highly unlikely and appears not to be desirable to obtain a single, "universal" specification language capable of "equally" elegantly, suitably abstractly modelling all aspects of a domain. Hence, one must conclude that the full modelling of domains shall deploy several formal notations. The issues are then the following: which combinations of notations to select, and how to make sure that the combined specification denotes something meaningful. The ongoing series of "Integrating Formal Methods" conferences [3, 12, 14, 27, 64] is a good source for techniques, compositions and meaning.

1.5 On Modelling

A number of remarks may be in order – especially given the terseness of many of the statements and examples given in the previous two sections.

1.5.1 Abstractions

In abstraction, we conscientiously omit some properties deciding instead to focus on other properties. Whenever an abstraction is presented, one should carefully discuss the abstraction choice. This has not been done in this chapter.

1.5.2 Models

We model both domain phenomena and domain concepts. The latter are abstractions of phenomena. However, in modelling a phenomenon, we "make it into" a concept. (So models of domain concepts could be said to be meta-concepts.) The point is: our domain description designates a model, or more generally, a class of models, each of which satisfies the description. These models are not the domain, only an abstract and only a model of it.

1.5.3 Real-World Constraints

Published models of, for example, railways,[8] contain axioms that reflect the physical constraints of the world of physics: trains arrive after they have departed, if a train appears in the traffic at times t and t' then it appears at

[8] `http://www.railwaydomain.org/PDF/tb.pdf`

all time in between these times, etc. We have not dealt with this aspect of modelling here – but refer to [7, Chap. 10], whose modelling principles and techniques are covered "in depth" in [5, 6].

1.5.4 Type Invariants

We have shown a few examples. Usually as part of axioms that govern type, or as subtype definitions, or as explicitly defined functions over types, say A, then usually named wf_A. Again we have not dwelled on this modelling aspect here – but refer to [5, Chaps. 13–18], which also cover principles and techniques of modelling type invariants.

1.5.5 Vagaries of Domains

Care has to be taken to model the domain not only as we would prefer the properties of its entities, functions, events and behaviours to be, but as they are. Namely stochastically varying, unpredictable, erroneous, etc. Some modelling aspects of this, notably regarding human behaviour, have been mentioned – and again we refer to the full book [5–7] for a more comprehensive treatment.

1.5.6 Monitoring of Domain States

In Sect. 1.4.6, we gave an example of the varieties of human behaviour – the second version of int_Cmd(mkPM(c,a,m,p,d)) (ρ,α,μ,ℓ). We did not follow up on any monitoring (audit of bank accounts) regarding this behaviour. We could and should, i.e., must model monitoring (and related control), if there is a notion of monitoring (etc.) in the domain. We have here assumed that such a monitoring, viz. a bank audit, currently might be a natural task for computing and would, hence, not describe it as part of the domain, but as part of a domain requirements prescription.

1.5.7 Incompleteness and Inconsistency

It is unavoidable that early iterations of domain description are incomplete, in fact may remain so throughout. It may not be detrimental to the overall objective of the emerging domain description as long as the "lacuna" of incompletenesses are well identified and acceptable. It is also difficult to avoid that early iterations of domain description are inconsistent. Domain description analysis, in the form of dispatching proof obligations raised by the formalisation, is one source of discovering inconsistencies. Another earlier source is that of validation in which two forms of inconsistencies may be identified. One is in which an inconsistency has its roots in conflicting statements about the domain from stakeholders supposedly of a same tightly knit group. In that case, the domain engineer must resolve it through mediation within such a group. The other source has the inconsistency in conflicting statements about the domain from stakeholders of different groups. Here, not the domain engineer, but stakeholder management must intervene and produce a consistent description.

1.6 From Domain Models to Requirements Models

One role for $\mathcal{D}$omain descriptions is to serve as a basis for constructing $\mathcal{R}$equirements prescriptions. The purpose of constructing $\mathcal{R}$equirements prescriptions is to specify properties (not implementation) of a $\mathcal{M}$achine. The $\mathcal{M}$achine is the hardware (equipment) and the software that together implements the requirements. The implementation relation is:

$$\mathcal{D}, \mathcal{M} \models \mathcal{R}$$

The $\mathcal{M}$achine is proven to implement the $\mathcal{R}$equirements in the context of (assumptions about) the $\mathcal{D}$omain. That is, proofs of correctness of the $\mathcal{M}$achine with respect to the $\mathcal{R}$equirements often refer to properties of the $\mathcal{D}$omain.

The $\mathcal{D}, \mathcal{M} \models \mathcal{R}$ formula expresses a context in which the individual phases of software development takes place. Awareness of this contextualisation was the basis for the *ProCoS* project [52, 56] and is also at the basis of the later work by Jackson et al. [28, 75].

1.6.1 Requirements Engineering Stages

As for domain engineering, requirements engineering is pursued in a number of iterated and sometimes concurrent stages:

1. Identification of and interaction with requirements stakeholders
2. Domain requirements development, modelling and analysis
3. Interface requirements development, modelling and analysis
4. Machine requirements development, modelling and analysis
5. Further analysis work
6. Requirements validation
7. Requirements satisfiability and feasibility study

In this section, we shall only cover the modelling parts of stages 2–4.

• • •

In the next three subsections, we very briefly sketch the relationships between a domain description and a requirements prescription. I introduce the notions of domain requirements (elsewhere called functional requirements), interface requirements (elsewhere sometimes called user requirements) and machine requirements (elsewhere called non-functional requirements). We find our labels semantically less "coded".

1.6.2 Domain Requirements

By domain requirements, we understand requirements (prescription) which use only terms from the domain description – and which additionally, in the requirements narrative, uses the terms shall or must (and probably not should or may).

First, in a concrete sense, you copy the domain description and call it a requirements prescription. Then that requirements prescription is subjected to a number of operations: (1) removal (projection away) of all those aspects not needed in the requirements; (2) instantiation of remain aspects to the specifics of the client's domain; (3) making determinate what is unnecessarily or undesirably non-deterministic in the evolving requirements prescription; (4) extending it with concepts not feasible in the domain and (5) fitting these requirements to those of related domains (say monitoring and control of public administration procedures). The result is called a domain requirements.

1.6.3 Interface Requirements

By interface requirements, we understand requirements (prescription) which use terms from both the domain description and the machine terminology – in addition to the terms shall or must. Interface requirements deal with the entities, functions (relations), events and behaviours that are shared between the domain and the machine.

From the domain requirements, one now constructs the interface requirements. First, one selects all phenomena and concepts, entities, functions, events and behaviours shared with the environment of the machine (hardware + software) being requirements specified.[9] Then prescribe how each shared phenomenon and concept is being initialised and updated: entity initialisation and refreshment, function initialisation and refreshment (interactive monitoring and control of computations) and the physiological man-machine and machine-machine implements.

1.6.4 Machine Requirements

By machine requirements, we understand requirements (prescription) which basically only use terms from the machine terminology – in addition to the terms shall or must. Machine requirements may, however, refer, generically, to entities, functions, events and behaviours of the domain.

Finally one deals with machine requirements performance, dependability, maintainability, portability, etc., where dependability addresses such issues as availability, accessability, reliability, safety, security, etc.

[9] A domain phenomenon or concept that is also to "appear" in the machine, in some form or another, is said to be shared.

1.6.5 Domain Descriptions vs. Requirements Prescriptions

On the background of the previous subsections (domain, interface and machine requirements), we can now characterise some differences between domain descriptions and requirements prescriptions.

Indicative vs. Putative

A narrative domain description is indicative. It uses such verbs as *is* and *are.* A domain description tells *what there is.*

A narrative requirements prescription is putative. It uses such verbs as *shall* and *must.* A prescription tells *what there will be.*

This distinction is not captured by the formalisations that we have shown. One could think of introducing some keywords for that purpose, but mathematically the meaning of such keywords is doubtful.

Computability

A formal domain description may very well denote mathematical entities, which cannot be represented computationally, or mathematical functions (or relations) that are not computable. The purpose of a domain and interface requirements construction (from a domain description) is to render possibly non-computable entities and possibly non-computable functions computable (as the objective of software design is to make entity representations and function computations data structure-wise and algorithmically efficient).

Stability

It is often claimed that requirements prescriptions are unstable: that they change repeatedly during software development. We claim, in contrast, that once a domain description has reached a relative maturity, i.e., being relatively stable, then using the domain and interface requirements "derivation" principles and techniques hinted at here (and covered in more details in [7, Chaps. 17–24]) then the resulting requirements prescriptions will be more, or even far more stable.

So, we have decomposed the "requirements instability" problem into three problems: (1) the "domain instability", (2) the "domain description instability" and (3) the "requirements prescription instability" problems. The domain instability problem, (1), has two roots: (a) the inherent changes of domain support technologies, management and organisation, rules and regulations facets, etc. and (b) the inherent inability of any group of people (i.e., the stakeholders and the domain engineers) to express and hence, capture all of a domain. As for (a), the scope of facet changes is more well-defined and their identification seems easier – so we may now be able to better cope with that part of the (former) "requirements instability" problem. As for (b), all there is to say is:

"such is life!" The domain description instability problem, (2), is due to misunderstandings, inconsistencies and incompletenesses of the stakeholders and the domain engineers' understanding of "their" domain. By having decomposed the overall (former) "requirements instability" problem into smaller, more well-defined problem areas and by decomposing the domain description problem into several facets as well as forcing their formalisation, we can hope to now be better able to deal with this, the (2), problem. Finally, the (new, redefined) "requirements instability" problem area, (3), is now of a size which is potentially much "smaller" than the former "requirements instability" problem area. Any actual "requirements prescription instability" is now due to a wrong decision by the stakeholders and domain engineers as to the desired projection, instantiation, determination, extension or fitting of the domain description at hand.

Overall we claim that the requirements engineering process – as we see it – is now based on a rather dramatically different basis than what is otherwise reported in the literature and certainly different from what is mostly practised in industry. Therefore, generic claims of "requirements instability" based on what we obviously consider an antiquated requirements engineering approach must be refined to claims of the kind: (a), (b), (2) or (3).

Domain Engineering vs. Requirements Engineering Stages

The domain engineering phase involves the stages of (D1) identification of and regular interaction with stakeholders, (D2) domain (knowledge) acquisition, (D3) domain analysis, (D4) domain modelling, (D5) domain verification, (D6) domain validation and (D7) domain theory formation. The requirements phase involves the stages of (R1) identification of and regular interaction with stakeholders, (R2–R4) domain, interface and machine requirements development, modelling and analysis, (R5) further analysis work, (R6) requirements validation and (R7) requirements satisfiability and feasibility study. Phases (D1,R1), (D5,R5) and (D6,R6) have similar names, but their inputs and hence, their procedural steps (principles, techniques and tool) are different. The same is true, of course, for all phases, but we can say that each of the requirements engineering phases (R2), (R3) and (R4) is "similar" to a combination of domain engineering phases (D2), (D3) and (D4). Space considerations prevent us from further clarifications.

1.7 Why Domain Engineering?

1.7.1 Two Reasons for Domain Engineering

We believe that one can identify two almost diametrically opposed reasons for the pursuit of domain descriptions. One is utilitarian, concrete, commercial and engineering goal-oriented. It claims that domain engineering will lead to better software and development processes that can be better monitored and controlled. The other is science-oriented. It claims that establishing domain theories is a necessity, that it must be done, whether we develop software or not.

We basically take the latter, the science-oriented one, while, of course, noting the former, the engineering consequences. We will briefly look at these.

1.7.2 An Engineering Reason for Domain Modelling

In a recent e-mail, in response undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote:[10]

"There are many unique contributions that can be made by domain modelling.

1. The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.
2. They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.
3. They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.
4. They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.
5. They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided".

All of these issues are dealt with, one-by-one and in depth, in Vol. 3 of my three volume book [7].

1.7.3 On a Science of Domains

Domain Theories

Although not dealt with in this chapter, the concept of domain theories must now be mentioned.

- *By a* **domain theory**, *we shall understand a domain description together with lemmas, propositions and theorems that can be proved about the description – and hence, can be claimed to hold in the domain.*

To create a domain theory, the specification language must possess a proof system. It appears that an essence of possible theorems of – that is, laws about – domains can be found in laws of physics. For a delightful view of the law-based nature of physics – and hence possibly also of man-made universes

[10] E-Mail to Dines Bjørner, CC to Robin Milner et al. July 19, 2006.

we refer to Richard Feynman's Lectures on Physics [20]. However, while laws of physics are conjectures and may basically (still) be refuted, laws of man-made domains are less conjectural,[11] but sometimes subject to change.[12]

A Scientific Reason for Domain Modelling

So, inasmuch as the above-listed issues of Sect. 1.7.2 are of course of utmost engineering importance, scientific issues are more important and should first and foremost be understood. There is no excuse for not trying to first understand. Whether that understanding can be "translated" into engineering tools and techniques is then another matter. But then, of course, it is nice that clear and elegant understanding also leads to better tools and hence, better engineering. It usually does.

1.8 Conclusion

1.8.1 Summary

We have introduced the scientific and engineering concepts of domain theories and domain engineering and have brought but a mere sample of the principles, techniques and tools that can be used in creating domain descriptions.

1.8.2 Grand Challenges of Informatics

To establish a reasonably trustworthy and believable theory of a domain, say the transportation or just the railway domain, it may take years, possibly 10–15! It is similar for domains such as the financial service industry, the market (of consumers and producers, retailers, wholesaler, distribution cum supply chain), health care and so forth.

The current author urges younger scientists to get going! It is about time.

Acknowledgements

The author thanks Springer for allowing this chapter, which is based on Chaps. 5 and 11 of [7], to be included in the present book. The author thanks Paul Boca, Jonathan P. Bowen and Jawed Siddiqi for the invitation to present this chapter in the BCS-FACS seminar series in the summer of 2005. The author thanks Tony Hoare for permission to quote his characterisation of domain engineering (Sect. 1.7.2).

[11] In early versions of a domain description, one that has yet to be validated by domain stake holders, one might very well claim that the model is conjectural.

[12] Typically we would wish the intrinsics part of a domain model to be invariant, to reflect innate laws of the domain. And: typically management and organisation and also rules and regulations are contain or reflect laws that may be valid for a time – only to be replaced by other "laws".

References

1. J.-R. Abrial: *The B Book: Assigning Programs to Meanings* (Cambridge University Press, Cambridge, MA, 1996)
2. K.R. Apt: *Principles of Constraint Programming* (Cambridge University Press, Cambridge, MA, 2003)
3. K. Araki, A. Galloway, and K. Taguchi, editors. *IFM 1999: Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 1945, New York, June 1999. Springer (Proceedings of 1st International Conference on IFM)
4. M. Bidoit and P.D. Mosses: CASL *User Manual* (Springer, Berlin, 2004)
5. D. Bjørner: *Software Engineering, Vol. 1: Abstraction and Modelling* (Springer, Berlin, 2006)
6. D. Bjørner: *Software Engineering, Vol. 2: Specification of Systems and Languages* (Springer, Berlin, 2006)
7. D. Bjørner: *Software Engineering, Vol. 3: Domains, Requirements and Software Design* (Springer, Berlin, 2006)
8. D. Bjørner, M. Henson, editors: *Logics of Specification Languages* (Springer, Berlin, 2007)
9. D. Bjørner, C.B. Jones, editors: *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, vol. 61 (Springer, Berlin, 1978)
10. D. Bjørner, C.B. Jones, editors: *Formal Specification and Software Development* (Prentice-Hall, Hemel Hempstead, 1982)
11. D. Bjørner, A. Yasuhito, C. Xiaoyi and X. Jianwen: *A Family of License Languages*. Technical Report, JAIST, Graduate School of Information Science, Nomi, Ishikawa (2006)
12. E.A. Boiten, J. Derrick, G. Smith, editors. *IFM 2004: Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 2999, London, April 4–7 2004 Springer (Proceedings of 4th International Conference on IFM. ISBN 3-540-21377-5)
13. E. Börger, R. Stärk: *Abstract State Machines. A Method for High-Level System Design and Analysis* (Springer, Berlin, 2003)
14. M.J. Butler, L. Petre, K. Sere, editors. *IFM 2002: Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 2335, Turku, May 15–18 2002 Springer (Proceedings of 3rd Intrenational Conference on IFM. ISBN 3-540-43703-7)
15. D. Cansell and D. Méry. Logical foundations of the B method. *Computing and Informatics*, 22:257–283, 2003
16. CoFI (The Common Framework Initiative): CASL *Reference Manual*, Lecture Notes in Computer Science (IFIP Series), vol. 2960 (Springer, Berlin, 2004)
17. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001
18. J. de Bakker: *Control Flow Semantics* (The MIT Press, Cambridge, MA, 1995)
19. R. Diaconescu, K. Futatsugi and K. Ogata. CafeOBJ: Logical foundations and methodology. *Computing and Informatics*, 22:1001–1025, 2003
20. R. Feynmann, R. Leighton, M. Sands: *The Feynmann Lectures on Physics*, vols. I–II (Addison-Wesley, California Institute of Technology, Reading, MA, 1963)
21. J.S. Fitzgerald and P.G. Larsen: *Developing Software using VDM-SL* (Cambridge University Press, Cambridge, MA, 1997)
22. K. Futatsugi and R. Diaconescu: *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification* (World Scientific Publishing, Singapore, 1998)

23. K. Futatsugi, A. Nakagawa and T. Tamai, editors. *CAFE: An Industrial-Strength Algebraic Formal Method*, Amsterdam, The Netherlands, 2000. Elsevier (Proceedings from an April 1998 Symposium, Numazu)
24. C.W. George, P. Haff, K. Havelund et al: *The RAISE Specification Language* (Prentice-Hall, Hemel Hampstead, 1992)
25. C.W. George and A.E. Haxthausen. The logic of the RAISE specification language. *Computing and Informatics*, 22(3–4):323–350, 2003
26. C.W. George, A.E. Haxthausen, S. Hughes et al: *The RAISE Method* (Prentice-Hall, Hemel Hampstead, 1995)
27. W. Grieskamp, T. Santen, B. Stoddart, editors. *IFM 2000: Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 1945, Schloss Dagstuhl, November 1–3, 2000. Springer (Proceedings of 2nd International Conference on IFM)
28. C.A. Gunter, E.L. Gunter, M.A. Jackson and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000
29. C.A. Gunter, S.T. Weeks and A.K. Wright: Models and languages for digtial rights. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)* (IEEE Computer Society Press, Maui, 2001), pp. 4034–4038
30. C. Gunther: *Semantics of Programming Languages* (The MIT Press, Cambridge, MA, 1992)
31. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 3:231–274, 1987
32. D. Harel. On visual formalisms. *Communications of the ACM*, 33(5):514–530, 1988
33. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 7:31–42, 1997
34. D. Harel, H. Lachover, A. Naamad et al. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering*, 4:403–414, 1990
35. D. Harel and R. Marelly: *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine* (Springer, Berlin, 2003)
36. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996
37. M.C. Henson, S. Reeves and J.P. Bowen. Z Logic and its consequences. *Computing and Informatics*, 22(3–4):381–415, 2003
38. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992
39. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996
40. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999
41. D. Jackson: *Software Abstractions Logic, Language, and Analysis* (The MIT Press, Cambridge, MA, April 2006)
42. M.A. Jackson: *Principles of Program Design* (Academic Press, New York, 1969)
43. M.A. Jackson. Problems, methods and specialisation. *Software Engineering Journal*, 9(6):249–255, 1994
44. M.A. Jackson: Problems and requirements (software development). In: *Second IEEE International Symposium on Requirements Engineering (Cat. No. 95TH8040)* (IEEE Computer Society Press, 1995), pp. 2–8
45. M.A. Jackson: *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices* (Addison-Wesley, Reading, MA, 1995)

46. M.A. Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997
47. M.A. Jackson: *Problem Frames – Analyzing and Structuring Software Development Problems* (Addison-Wesley, Edinburgh, 2001)
48. K. Jensen: *Coloured Petri Nets*, vol. 1: Basic Concepts (234 pages + xii), vol. 2: Analysis Methods (174 pages + x), vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science* (Springer, Heidelberg 1985, revised and corrected second version: 1997)
49. J. Klose, H. Wittke: An automata based interpretation of live sequence charts. In: *TACAS 2001*, T. Margaria, W. Yi, editors, (Springer, Berlin, 2001), pp. 512–527
50. L. Lamport. The temporal logic of actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1995
51. L. Lamport: *Specifying Systems* (Addison-Wesley, Boston, MA, 2002)
52. H. Langmaack, W.P. de Roever, J. Vytopil, editors: *Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, vol. 863 (Springer, Heidelberg, 1994)
53. E. Luschei: *The Logical Systems of Lesniewski* (Amsterdam, The Netherlands 1962)
54. S. Merz. On the logic of TLA+. *Computing and Informatics*, 22(3–4):351–379, 2003
55. T. Mossakowski, A.E. Haxthausen, D. Sanella and A. Tarlecki. CASL – The common algebraic specification language: semantics and proof theory. *Computing and Informatics*, 22(3–4):285–3321, 2003
56. E.-R. Olderog: *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship* (Cambridge University Press, Cambridge, MA, 1991, paperback 2005)
57. C.A. Petri: *Kommunikation mit Automaten* (Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962)
58. R. Pucella and V. Weissman: A logic for reasoning about digital rights. In: *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)* (IEEE Computer Society Press, Maui, 2002), pp. 282–294
59. W. Reisig: *Petri Nets: An Introduction*, *EATCS Monographs in Theoretical Computer Science*, vol. 4 (Springer, Berlin, 1985)
60. W. Reisig: *A Primer in Petri Net Design* (Springer, Berlin, 1992)
61. W. Reisig: *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets* (Springer, Heidelberg, 1998)
62. W. Reisig. The expressive power of abstract state machines. *Computing and Informatics*, 22(3):209–219, 2003
63. J.C. Reynolds: *The Semantics of Programming Languages* (Cambridge University Press, Cambridge, MA, 1999)
64. J.M. Romijn, G.P. Smith, J.C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 3771, Eindhoven, The Netherlands, December 2005. Springer (Proceedings of 5th International Conference on IFM. ISBN 3-540-30492-4)
65. P. Samuelson. Digital rights management {and, or, vs.} the law. *Communications of ACM*, 46(4):41–45, 2003
66. D.A. Schmidt: *Denotational Semantics: A Methodology for Language Development* (Allyn & Bacon, Boston, MA, 1986)

67. J.F. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations* (Pws Pub Co, August 17, 1999)
68. J.M. Spivey: *Understanding Z: A Specification Language and its Formal Semantics*, *Cambridge Tracts in Theoretical Computer Science*, vol. 3 (Cambridge University Press, Cambridge, MA, 1988)
69. J.M. Spivey: *The Z Notation: A Reference Manual*, 2nd edn (Prentice Hall International Series in Computer Science, 1992)
70. J.T.J. Srzednicki, Z. Stachniak, editors: *Lesniewski's Lecture Notes in Logic* (Dordrecht, Kluwer, 1988)
71. R. Tennent: *The Semantics of Programming Languages* (Prentice-Hall, New York, 1997)
72. F. Van der Rhee, H. Van Nauta Lemke and J. Dukman. Knowledge based fuzzy control of systems. *IEEE Transactions on Automatic Control*, 35(2):148–155, 1990
73. G. Winskel: *The Formal Semantics of Programming Languages* (The MIT Press, Cambridge, MA, 1993)
74. J.C.P. Woodcock and J. Davies: *Using Z: Specification, Proof and Refinement* (Prentice Hall International Series in Computer Science, 1996)
75. P. Zave and M.A. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997
76. C.C. Zhou and M.R. Hansen: *Duration Calculus: A Formal Approach to Real–time Systems* (Springer, Heidelberg, 2004)
77. C.C. Zhou, C.A.R. Hoare and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992

2

Program Verification and System Dependability

Michael Jackson

The Open University and The University of Newcastle, Newcastle upon Tyne, UK
jacksonma@acm.org

Abstract Formal verification of program correctness is a long-standing ambition, recently given added prominence by a "Grand Challenge" project. Major emphases have been on the improvement of languages for program specification and program development, and on the construction of verification tools. The emphasis on tools commands general assent, but while some researchers focus on *narrow verification* aimed only at program correctness, others want to pursue *wide verification* aimed at the larger goal of system dependability. This paper presents an approach to system dependability based on *problem frames* and suggests how this approach can be supported by formal software tools. Dependability is to be understood and evaluated in the physical and human problem world of a system. The complexity and non-formal nature of the problem world demand the development and evolution of normal designs and normal design practices for specialised classes of systems and subsystems. The problem frames discipline of systems analysis and development that can support normal design practices is explained and illustrated. The role of formal reasoning in achieving dependability is discussed and some conceptual, linguistic and software tools are suggested.

2.1 Introduction

The earliest work [26] on program verification – by von Neumann and Goldstine and then Turing – began in the mid-1940s, as soon as the electronic digital computer became a practical possibility. These pioneers recognised immediately what generations of programmers after them discovered by humbling practical experience: Writing programs is easy; writing correct programs is difficult. Discussing the process of deriving a program from a "meaningful text ... [in] the language of mathematics", von Neumann and Goldstine [13] wrote:

> "Since coding is not a static process of translation, but rather the technique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and

one that represents a new branch of formal logics. We propose to show in the course of this report how this task is mastered".

Inevitably this early work was concerned with machine code and the design of program control flow was typically depicted in flowcharts. The fundamental approach of von Neumann and Goldstine, and slightly later of Turing [40], was to consider values of program variables, capturing the deduced properties and relationships of those values by assertions tied to locations in the flowchart. Assertions tied to the start and finish locations are, in effect, preconditions and postconditions, respectively, and so constitute a program specification in a sense that was to become usual. Reasoning about these and the other assertions in the program flow could give a convincing proof that the program was correct, in the sense that it satisfied its specification. As early as 1961, McCarthy proposed that correctness proofs could be checked by a computer program [35]. In 1967, Floyd [11] proposed the construction of a *verifying compiler* that would check program correctness as a part of the compilation process. A workable prototype verifier was constructed by King in 1969 [28].

The need to program at a higher level of abstraction became evident from the earliest years, and gave rise to work on imperative programming languages (and functional languages too). An important part of the motivation was the vision of automatic translation of mathematical specifications into program texts. It was implicitly assumed that such a specification, correctly reflecting the developers' needs and intentions, would be available, often in the form of a set of equations to be solved. Fortran was named to reflect its role as a formula translator, and Algol as a language for the expression of algorithms. Subroutines, structured control flow, parameterised procedure call, data types and other programming abstractions were introduced, clarified and refined. To some extent, these abstractions corresponded to mathematical concepts and also provided a basis for program structure – what might be called software architecture in the small. They made it easier both to develop programs and to analyse and verify their properties. Programmers accepted – at first, not always willingly – some apparent restrictions on the patterns of machine-code instructions that could be executed; the gain was an increased clarity and reliability with which the programmer's intentions could be expressed and their consequences in execution could be forecast and analysed. Compilers and interpreters, in various configurations, eventually provided reliable paths from the more abstract program text to a concrete efficient execution that correctly conformed to the programming language semantics. It was recognised that ease and reliability of program construction depend heavily on the quality of the programming language and its suitability to the task in hand.

Further, the basis of program verification, whether in a programming discipline for achieving correctness by construction or in the analysis of a given program text, was clearly seen to lie in the semantics of the programming language constructs. Equally, the semantics of those constructs could be at least partially defined by the axioms to be used in reasoning about programs [17]. Either way the verification task was intimately related to the constructs of the language in which the object of verification is written. Substantial progress was made that would surely have been impossible had machine code continued in use as the usual programming language. Jones's paper [26] on program verification – *The Early Search for Tractable Ways of Reasoning about Programs* – might well be complemented by another paper: *The Early Search for Tractable Programs to Reason about.*

2.1.1 The Grand Challenge Project

This tradition of program construction and verification has been given greater prominence by a recently announced project. The project was conceived in response to the BCS initiative to identify Grand Challenges in Computing Research [20]; it has been referred to by its proponents under various titles: *The Verifying Compiler*, *Verified Software* or *Dependable Systems Evolution* [43]. A subsidiary proposal aims to form a *Verified Software Repository* containing verified software texts, challenge problems, models, specifications, case studies and other associated artefacts. The central thrust of the whole project is to work towards an ideal of correct software by developing and using intellectual, linguistic and software tools for program construction and analysis, including a sound and complete theory of programming [19] and enhanced languages [29] for specification and programming. The project appears to emphasise specifications that are embedded in the program text as assertions over program variables, such specifications having the advantage that they can be added very gradually to legacy programs in the hope of reducing their budget of existing errors. However, it also embraces specifications expressed in separate documents describing required properties of program execution. In all its aspects the approach is primarily formal and the uncompromising goal is program correctness with respect to given formal specifications.

The variety of titles that have been used indicates an ambivalence in the project goals. Sometimes the project scope is presented as what we will call *wide verification*, aiming to achieve overall dependability of critical systems in which software is only one part among many. In [43] the proposal mentions the need for dependability in systems for avionics, e-voting, stockbroking, telephone switching and e-business. The Mondex purse system, nuclear power station control, administrative systems for the UK Passport Office and Child Support Agency, airline reservations systems, systems for Mars landers and the control system for the Rotterdam storm-surge barrier have also been cited in project documents as illustrative examples of the need for system dependability. At other times, the scope is clearly restricted to what we will call *narrow verification* – limited to verifying the computer programs against their specifications. In [19] the project vision is described like this:

> "We envision a world in which computer programmers do not make mistakes and certainly never deliver them to their customers. At present, programming is the only modern engineering profession which spends half its time on detecting and correcting mistakes made in the other half of its time. The resulting delays to delivery are notorious. We envision a world in which computer programs are always the most reliable component of any system or device that contains them. At present, devices frequently have to be switched off and on (re-booted) to clear a software error that prevents them from working".

Similarly, in [29] the question "How do we ensure that our models and properties properly reflect the requirements on the system?" is posed, but is immediately rejected: "we will largely ignore [this] question..., since it is too large and important to be included in our grand challenge". The goal is to improve the software; for this purpose the other parts of the system are not of direct interest.

From the early stages of the project, this ambivalence was disturbing to some participants. In the words of [18], the project was in danger of choosing to "concentrate solely on solutions that can be found by the known hard scientific methods of modelling and deduction, ones that can be wholly represented and solved within the silicon package of a computer". The phrase "silicon package" is very appropriate; in the past, much of the admired work on program correctness has carefully restricted its scope by eschewing even simple input-output operations, which threaten to pierce the veil separating the program from its non-formal environment. The motivation to narrow, rather than widen, the scope was memorably expressed by Dijkstra [9]:

> The choice of functional specifications – and of the notation to write them down in – may be far from obvious, but their role is clear: it is to act as a logical "firewall" between two different concerns. The one is the "pleasantness problem", i.e., the question of whether an engine meeting the specification is the engine we would like to have; the other one is the "correctness problem", i.e., the question of how to design an engine meeting the specification.
> ...the two problems are most effectively tackled by...psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem.

In this view, the role of program specifications is to separate the quasi-formal world within the silicon package of the computer – which, suitably abstracted, is the subject matter of computing science – from the non-formal world outside it – which falls under other, less formal, disciplines. The non-formal nature of the human and physical world outside the computer, it is held, precludes the use of the kind of formal reasoning that is appropriate to the development of programs to satisfy formal specifications.

2.1.2 The Theme of This Paper

The term *wide verification*, used in the preceding section, is unconventional. *Verification*, understood as establishing a formal relationship between two formal descriptions, is commonly contrasted with *validation*, which is taken to be the inevitably non-formal determination that "an engine meeting the specification is the engine we would like to have".

This suggested contrast is less convincing than it appears at first sight. Validation, like verification, must reason about formal descriptions. In both cases, the formal descriptions are only imperfectly true. In validation, the necessary descriptions of the human and physical world are explicit formalisations of a non-formal reality. The same is also true of a program text, describing a physical execution that depends on the uninterrupted correct functioning of the computer and its operating system. Effective formal reasoning for a practical purpose always depends on a fundamental assumption: that the formal descriptions we reason about are good enough approximations to the realities they describe.

Choice of the term *wide verification*, then, is intentionally provocative, inviting a recognition that validation conducted in pursuit of system dependability can exploit essentially the same kind of reasoning as program verification. This paper addresses the questions: What is the nature of wide verification in this sense? Is it different from narrow verification and, if so, how does it differ? How are formal and non-formal concerns related? What is system dependability, and how can it be achieved?

Dependability is to be understood and evaluated, not in the software but in the environment – the physical and human problem world of the system. In the immediately following section, the relationship between the problem world and the software is discussed and some existing approaches to wide verification are briefly presented. The several sections following present and illustrate an approach to system dependability based on *problem frames*. Problem frames [21–23,31] bring the problem world into sharp focus at the centre of the developers' view. The complexity and non-formal nature of the problem world demand the development and evolution of specialised *normal designs* and *normal design practices* for software-intensive systems and subsystems, analogous to those found in established branches of engineering. It is an important advantage of the problem frames discipline that it can support the evolution and use of normal design practices. It also induces system structures that help to expose deep concerns and failure risks. Presentation of the problem frames discipline itself in this paper is intentionally very sketchy: the central purpose is to explain and illustrate the larger themes of normal design, structures, concerns and risks.

The role of formal reasoning tools in this context is then discussed. Some conceptual, linguistic and software tools are suggested for checking adherence to normal design practice. Additionally, such tools can help to identify likely sites of faults that can lead to system failures, including some failures specifically arising from the inherently non-formal nature of the problem world. Software-intensive systems cannot be proved correct; but appropriate software tools can make a large contribution to improving their dependability.

2.2 Dependability and the Problem World

The systems mentioned earlier in connection with the need for dependability – systems for avionics, e-voting, stockbroking, telephone switching, e-business, e-cash, nuclear power station control, control of the Rotterdam storm-surge barrier and others – are all software-intensive systems. They are critical systems of which software forms a central part, but only one part among many. The other parts are drawn from the physical world – human, natural and engineered – and it is in those other parts that the systems' purposes are defined, their effects are felt and their dependability is to be evaluated.

For a system to control traffic at a complex road intersection, for example, the purpose is safe, convenient and efficient movement of vehicles and pedestrians. The system's dependability is measured, not by properties of the software, but instead by the incidence of avoidable collisions, pedestrian injuries, or unnecessary traffic delays or congestion. For the system to control the Rotterdam storm-surge barrier, the purpose is to ensure that the barrier – which takes eleven hours to open or close – is closed in stormy weather to protect the harbour and is otherwise open. For a system to control the lifts in a building, the purpose is to provide safe and efficient transport of the building's users from floor to floor. In every case, the criterion of a successful system is its effect in the problem world: that is, in the physical world outside the computer.

2.2.1 Reasoning About the Machine and the World

The fundamental principle of the problem frames approach [21–23, 31] is to recognise that in a software-intensive system the software can – and should – be regarded as describing a physical *machine* introduced into the physical *problem world.* The machine consists of all or part of one or more digital computers, their behaviour specialised by the software to serve a particular purpose. As software developers we are concerned only to produce the software parts of the machine, the physical digital computers being the product of hardware engineers. It is because the hardware engineers have done their work so well that we can regard our programs as reliably defining significant behavioural properties of the physical machines.

The purpose, or *requirement*, of the system is to bring about and maintain some condition or behaviour in the problem world. Essentially, this is to be achieved by the machine, which interacts with the problem world at an interface of *shared phenomena*, such as events, states and encoded values held in shared states or transmitted in shared events. (An event, for example, is shared if both the machine and the problem world participate in the event, and state is shared in the sense associated with shared storage in programming.) However, the requirement is not, in general, concerned with these shared phenomena, but with other phenomena that lie at some distance from the interface, deeper into the problem world. The specified behaviour of the machine at the interface must be such as to ensure satisfaction of the requirement. This is possible only because certain given *problem world properties* relate the interface phenomena to the phenomena in terms of which the requirement is expressed.

This view is depicted in Fig. 2.1. The machine is endowed with certain properties $\mathcal{M}$ by the software, and by the underlying mechanisms of its physical construction, operating system and compilers and interpreters. The machine has some behaviour at its interface A with the environment, which we may imagine to be described by a specification $\mathcal{S}$. The requirement $\mathcal{R}$ is expressed in terms of the problem world phenomena B. The problem world properties $\mathcal{W}$ are those that the problem world possesses independently of the machine's possible behaviours. They are expressed in terms of the phenomena A and B, and may also refer to other phenomena C, which are "hidden" phenomena of the problem world, neither shared with the machine nor mentioned in the requirement.

Dependability of such a system means dependable satisfaction of the requirement $\mathcal{R}$. If we assume that the problem world and its properties $\mathcal{W}$ are given – that is, we are engaged in software engineering rather than in a much broader enterprise in which we are free to modify or replace parts of the problem world – then as software engineers our task is to develop a machine which, once installed in the problem world, will dependably ensure satisfaction of $\mathcal{R}$. Verifying formally that we have succeeded would constitute *wide verification*, proving that $\mathcal{M},\mathcal{W} \models \mathcal{R}$.

Taking this point of view, we can identify the goal of *narrow verification*: it is to prove that the machine, as specialised by the software, satisfies the specification $\mathcal{S}$. The specification $\mathcal{S}$ may be total with respect to the problem world: that is, it specifies machine behaviour at the interface A for all possible

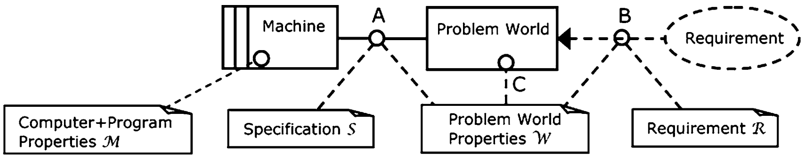

Machine
A
Problem World
B
Requirement
C
Computer+Program Properties $\mathcal{M}$
Specification $\mathcal{S}$
Problem World Properties $\mathcal{W}$
Requirement $\mathcal{R}$

Fig. 2.1. A software-intensive system

problem worlds, provided only that they satisfy the syntax of interface A. For example, if the interface at A were a keyboard, a total specification would specify the machine behaviour for all possible sequences of key presses. In this case, narrow verification would prove the entailment $\mathcal{M} \models \mathcal{S}$. Alternatively, the specification $\mathcal{S}$ may be partial with respect to the problem world, specifying machine behaviour only for problem worlds that satisfy some semantic constraint $\mathcal{V}$ at the interface A such that $\mathcal{W} \Rightarrow \mathcal{V}$. To give a programming example, to specify that a function programmed in an imperative language is not re-entrant is to specify its behaviour at its call interface only for those problem worlds that do not invoke the function again before it has returned the result of a previous invocation. In this case, narrow verification would prove the entailment $\mathcal{M}, \mathcal{V} \models \mathcal{S}$.

Dijkstra's introduction of the functional specification as a firewall between the pleasantness problem and the correctness problem aims to separate wide verification into two distinct tasks. For the correctness problem, it is necessary to prove the entailment $\mathcal{M} \models \mathcal{S}$; for the pleasantness problem, it is necessary to prove $\mathcal{S},\mathcal{W} \models \mathcal{R}$. Both proofs together would constitute wide verification, proving that $\mathcal{M},\mathcal{W} \models \mathcal{R}$. (It is worth noting here that existence of a formal specification $\mathcal{S}$ is essential to narrow verification, but not necessarily essential to wide verification.)

At first sight, wide verification is simply an enlargement of the scope of narrow verification. Expression of the requirement $\mathcal{R}$ and the problem world properties $\mathcal{W}$ is based on consciously chosen formalisations of the problem world. Formalisation of a physical problem world is necessarily an approximation; but approximation is unavoidable if we are to reason at all: it underlies the analytical modelling that is the foundation of much design work in the established branches of engineering. In short, up to the approximation due to our formalisations, it seems that we may apply techniques similar, or even identical, to those that have demonstrated their value in program verification. Significant work has been done on these lines, and some examples are given in the next section.

2.2.2 Some Approaches to Reasoning About the World

One important group of approaches to extending the scope of formal reasoning to include the problem world includes what are often called formal methods. For example, the specification languages Z [15] and VDM [25] and their successors have been applied to development problems of many kinds, both in research studies and in practical projects. In these approaches, the whole system, including both the machine and the problem world, is structured in terms of descriptions of a state and a set of operations on it. The central emphasis is on formal logic and on the discrete mathematics appropriate to the chosen states and operations [44]. This work does not fit easily into the view depicted in Fig. 2.1, chiefly because two related distinctions that we have picked out as important – between the machine and the problem world, and between the specification and the requirement – are not explicitly articulated: in some published problem treatments, it is difficult to know exactly what is being described in particular descriptions. Nonetheless, these approaches have been influential both in motivating and studying the formalisation of significant problem worlds and in applying formal reasoning to the results.

In [34] the distinction between machine and problem world was made explicit for simple process-control problems. The approach taken introduces the problem world phenomena and properties into the program text itself in the form of auxiliary function-valued *reality variables*. For example, in a train control application, there might be a function-valued variable $p(t)$ denoting the position at time t of the train on a linear track, and another variable $a(t)$ denoting the acceleration at time t. As time passes, the values of these functions change because their domain, time, is continually extended by new increasing values. Reality variables are related by explicit equations that capture the dynamic physical properties of the problem world. For example, the function value of the variable $p(t)$ is related to the function value of $a(t)$. As time passes, the executing program is imagined to update these fictional reality variables in accordance with the requirement, the problem world properties and the machine's interactions with the environment through sensors and actuators.

Development of the control program is regarded as a series of refinement steps, proceeding from an initial statement of the requirement to an executable program. Initially the text refers only to the fictional reality variables mentioned in the requirement: the refinement eventually eliminates all reference to these variables (other than those that interact directly with the machine) by appealing to the properties of the problem world and of the sensors and actuators at the machine interface. Essentially, it refines the requirement $\mathcal{R}$ to the machine properties $\mathcal{M}$ by appealing to $\mathcal{W}$ in the refinement steps. The refinement style used is the weakest precondition calculus [8], extended to allow function-valued variables. The place of the postcondition is now taken by an expression over values of function-valued reality variables, denoting the conditions to be achieved or maintained in the problem world by the control program – that is, the requirement $\mathcal{R}$. The approach is particularly notable for the way in which reasoning about the problem world is fully assimilated to, and integrated with, reasoning about programs. A successful application of the approach might perhaps claim to have achieved system correctness by construction in exactly the manner demonstrated by Dijkstra for programs. The machine specification $\mathcal{S}$ in the form shown in Fig. 2.1 does not appear explicitly, but it could properly be said that the goal of the approach is wide verification, proving that $\mathcal{M}, \mathcal{W} \models \mathcal{R}$.

The machine and the problem world are again clearly separated in the work of Parnas and Madey [36], where they are fitted into the generalised structure shown in Fig. 2.2. The machine and problem world are called the *system* and the *environment*, respectively, and the interactions between them are mediated by *sensors* and *actuators*. Four sets of variables are identified

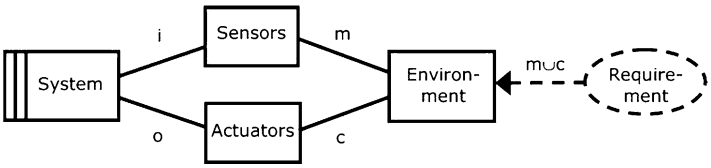


Fig. 2.2. Parnas' Four-Variable Model

(to which the approach owes its soubriquet "Four-Variable Model"). The sets of monitored and controlled variables of the environment are m and c, while the sets i and o carry the values that the machine reads from the sensors and writes to the actuators. Five relations over vectors of time functions of these variable sets are defined:

$SOF : \underline{i} \leftrightarrow \underline{o}$	captures the machine behaviour;
$IN : \underline{m} \leftrightarrow \underline{i}$ and $OUT : \underline{o} \leftrightarrow \underline{c}$	capture the properties of the sensors and actuators respectively;
$NAT : \underline{m} \leftrightarrow \underline{c}$	captures the given environment properties;
$REQ : \underline{m} \leftrightarrow \underline{c}$	captures the requirement.

The formula $NAT \cap (IN.SOF.OUT) \subseteq REQ$ characterises the *acceptability* of the system. That is: the requirement REQ is satisfied by the behaviour of the whole system, in which the environment (whose given behaviour is NAT) is placed in parallel with the machine (whose behaviour is $IN.SOF.OUT$). If we choose to regard the sensors and actuators of Fig. 2.2 as parts of the machine of Fig. 2.1, then NAT plays the role of $\mathcal{W}$, $IN.SOF.OUT$ plays the role of $\mathcal{S}$, and REQ is $\mathcal{R}$. The formula then corresponds to the entailment $\mathcal{S},\mathcal{W} \models \mathcal{R}$, both $\mathcal{S}$ and $\mathcal{R}$ being expressed in terms of the same set of phenomena $m \cup c$. If we choose to regard the sensors and actuators as parts of the environment, a rearrangement of the acceptability formula gives a different correspondence to the entailment $\mathcal{S},\mathcal{W} \models \mathcal{R}$. In either case, a proof of acceptability, combined with a proof that SOF is satisfied by the program text, constitutes a wide verification.

The KAOS approach to requirements engineering [41] treats the overall system requirement $\mathcal{R}$ as a *goal*, positioned at the root of a refinement tree of subgoals. Goals are formalised in a real-time temporal logic, and commonly occurring patterns may be identified and given such names as Achieve, Maintain, Avoid, Increase and so on. As the goal tree is refined, responsibilities for goals and subgoals are assigned to agents, each of which is either the machine or an identifiable part of the problem world. Each subgoal is considered in the context defined by its place in the goal tree and inherits assertions and assumptions from its ancestors. The problem world properties $\mathcal{W}$ emerge during this refinement as subgoals are assigned to agents, the agents' given properties being explored and captured in the process of choosing and justifying each assignment. The specification $\mathcal{S}$ is the conjunction of all the responsibilities that have been assigned to the machine during the refinement process.

Praxis Critical Systems Ltd. have developed, and use and advocate, an approach [2, 14] to requirements engineering that is firmly based on the entailment $\mathcal{S}, \mathcal{W} \models \mathcal{R}$. Their approach, named REVEAL, lays particular stress on *rich traceability*: that is, the ability not only to trace links between development artifacts but also to provide arguments associated with the links that explain how and why particular requirements are satisfied.

Finally, in [16], an approach is described that draws directly on ideas from a number of sources and specifically from the view depicted in Fig. 2.1. The problem is decomposed into a number of subproblems, each with its own machine, problem world and requirement. Explicit attention is paid to the recombination of the subproblems (and of their solutions) to give a structuring and solution of the original undecomposed problem. The specification of a machine is cast in the form of a *rely-guarantee* pair [24]: the machine specification is that it *relies* on the problem world properties and *guarantees* satisfaction of the requirement. That is: $\mathcal{W}$ and $\mathcal{R}$ are given, and the machine specification $\mathcal{S}$ is *defined* by the entailment $\mathcal{S}, \mathcal{W} \models \mathcal{R}$.

2.2.3 Enhancing Reasoning for Dependability

Each of the approaches mentioned in the preceding section applies formal reasoning quite directly to the problem worlds and requirements of software-intensive systems, with a view to deriving or validating a proposed machine behaviour. But this apparently straightforward application of formal reasoning to wide verification, powerful as it can be, is not strong enough or rich enough to bear the full burden of achieving dependability in systems of realistic size and complexity. It is necessary to add a more powerful structuring discipline, in which a system under development is viewed as a structure of *subsystems*, embodying the solutions to *subproblems*. The subsystems are to be members of known subsystem classes, each with its repertoire of *concerns* that must be addressed if a good solution to the subproblem is to be obtained. Combining the subsystems to embody a solution to the original development problem then becomes an explicit development task in its own right, with its own characteristic concerns.

Such a structuring discipline is the central theme of the *problem frames* approach [21–23], which is presented in the following sections. Its ultimate purpose is to give rise, over time, to a corpus of *normal designs* and *normal design practices* in the development of software-intensive systems. The concept of normal design is explained and discussed in the course of presenting the structuring discipline. The presentation is given in terms of the problem frames approach, but this choice is not motivated by an insistence that the particular nomenclature and detail of this approach is essential to the task. It is motivated rather by a claim that a focus on *problems* is fundamental: the concerns and challenges that it exposes are independent of the techniques that developers may choose to exploit and must be addressed by any effective approach to dependability.

2.3 Problem Decomposition

Figure 2.1 showed a generalised representation of a system. We may think of it also as a representation of a *problem* posed to the system's developers. In Polya's terminology [37], this is a *problem to find.* The *principal parts* of the problem are the problem world and the requirement, which are *givens*, and a machine, which is *unknown.* The *solution task* is to find – that is, to develop – a machine that will ensure satisfaction of the requirement.

As for any complex problem, a fundamental solution technique is to decompose the problem into subproblems that have known solutions or are, at any rate, easier to solve. This section explains the nature of the problem frames approach to decomposition, using a highly simplified example.

2.3.1 Complexity in Software-Intensive Systems

Most software-intensive systems are functionally very complex. There are many reasons for this complexity. The presence of a digital computer, monitoring, controlling and coordinating the behaviour of the problem world, inevitably permits and invites a degree of discrete complexity that was not possible in earlier physical systems. Computer-based central locking systems for cars, for example, are hugely more complex than the electro-mechanical systems that preceded them. Market forces motivate feature proliferation of the kind seen in a mobile telephone that is also a video camera, a web browser, an email client, a TV and a personal organiser. The many features are expected to work together – for example, taking and immediately transmitting a photograph while talking on the telephone – the features co-operating with little or no intervention by the user. Automatic interoperation is prized, not only between different features of the same system but also between different systems – for example, between a telephone and a banking system that allows micro-payments to be made by telephone with the minimum of user effort. As more systems become mobile, it becomes desirable to adapt system function automatically to the environment's changing properties as the system moves. These characteristics give rise to the difficulties of *feature interaction*, a kind of complexity that was first explicitly recognised in telephone-switching systems but has come to be seen as an almost ubiquitous source of difficulty, apparent in software-intensive systems of many kinds. Well-justified demands for fault-tolerance and safety, and for security in a potentially hostile world, contribute further increments of complexity.

This complexity of interaction can be mastered by a decomposition structure in which the distinct functional contributions to complexity are identified and separated as *subproblems.* The treatment of the identified subproblems is also separated from the treatment of their interactions: subproblem composition is a distinct task, deferred until some understanding has been achieved of the subproblems to be composed.

2.3.2 A Simplified Example

The simplified example problem that we will discuss is the development of a controller for a lift in a building. The requirement is for a safe, efficient lift service. The basic lift service is that the lift comes to a floor when it is summoned and takes users to the floors they request. Safety implies that in the event of any equipment failure – the most dramatic failure being the breakage of a hoist cable – the lift car will not be allowed to fall freely but will be held firmly in the shaft by its emergency brake. For efficiency, the building manager can set varying service priorities to reflect changing demand in the rush hours or at lunch time, or to give preference to particular floors. A panel of lights in the building lobby indicates the current position of the lift car and its direction of travel, and the floors for which requests are currently outstanding.

Because the purpose of the example is not to propose a serious solution to a realistic problem but only to illustrate some concepts of problem decomposition and wide verification, it has been highly simplified in every respect. Neither its structure nor its detailed supposed functionality can claim to be realistic, and most details of interface phenomena and problem world properties are left to be imagined or inferred from comments in accompanying discussions. The problem diagram is shown in Fig. 2.3.

The problem world has been decomposed into a number of *problem domains*: identifiable parts whose individual properties and interactions combine to give the overall problem world properties. The solid lines connecting the problem domains to each other and to the machine represent interfaces of shared phenomena, as in Fig. 2.1. The machine interacts directly with the Lobby Display, whose states it controls, and with the Buttons, whose states it monitors. It interacts with the Lift Equipment, monitoring the states of the floor sensors that detect the presence of the lift car and controlling the polarity and power supply of the winding motor. It interacts with the Building Manager through a keyboard at which the manager can enter details of a desired priority regime. The Users do not interact directly with the Machine,

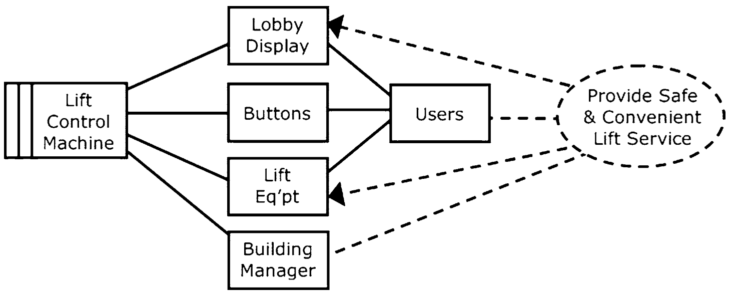


Fig. 2.3. A highly simplified example problem

but only with the Lobby Display, by looking at it, with the Buttons, by pressing them, and with the Lift Equipment, by entering and leaving the lift car at the different floors.

The dashed arrows linking the requirement to the problem domains show that the requirement constrains the Lobby Display and the Lift Equipment; the plain dashed lines to the Users and the Building Manager show that the requirement refers to those domains but does not constrain them. An observer asked to certify satisfaction of the requirement might refuse a certificate on the grounds that the Lobby Display does not have the appropriate lights on in some situation, or that the lift car has not arrived at the required floor; but nothing that the Users or the Building Manager do could in itself be grounds for refusal.

In this problem we may recognise three large subproblems suggested by the requirement. First, the *service* subproblem satisfies the requirement for efficient basic service according to the current priority regime set by the building manager. Second, the *display* subproblem monitors lift activity and drives the display. Third, the *safety* subproblem continually checks the equipment for faults and applies the emergency brake if a fault is detected.

2.3.3 Subproblems and Further Decompositions

These subproblems are further decomposed into smaller subproblems by applying standard decompositions. Figure 2.4 shows the decomposed *service* subproblem. The standard decomposition introduces a new problem domain: Priority Regime. This domain is a reified *lexical* domain: it is an encoding on a machine-readable medium of the rules to be observed in scheduling the lift service. Its explicit recognition as a problem domain in the decomposition allows the Building Manager's actions in editing the rules to be separated from the rule application in the provision of lift service. One advantage of this separation is the well-understood benefit of separating writers of a shared variable from its readers. Another advantage is that the subproblem of editing the rules falls squarely into a standard problem class.

Problem classes [22] are similar in concept to solution classes expressed in object-oriented patterns [12]. A problem class is characterised by the topology of its problem diagram, the nature of its problem domains, the relationship

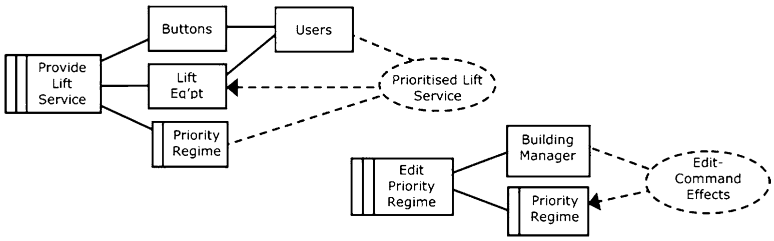


Fig. 2.4. The service subproblem, further decomposed

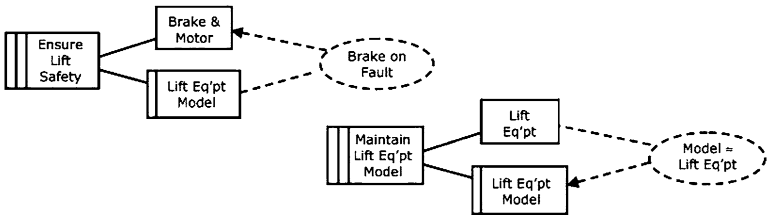


Fig. 2.5. The safety subproblem, further decomposed

among them stipulated by the requirement, and the locus of control of the phenomena in its interfaces. Edit Priority Regime is a *WorkPiece problem*, in which a human *User* issues commands to the machine, and the machine applies them to the *WorkPiece*, which is a reified lexical domain.

The decomposed *safety* subproblem is shown in Fig. 2.5. Again, the standard decomposition of the subproblem introduces a new problem domain, the Lift Equipment Model. This domain is a *model* – a reified representation – of the current and recent behaviour and states of the Lift Equipment, maintained by monitoring the floor sensors and the lift motor on/off and polarity states. The model is to be used as a surrogate for the real Lift Equipment in checking for faults to satisfy the *brake on fault* requirement. For example, if the model shows that the motor has been *on* with direction *up* for more than *maximum_interfloor_rise_time*, but the next floor sensor in the upwards direction has not yet closed, then ipso facto it shows that some equipment fault is present, and the emergency brake must then be applied by the Ensure Lift Safety machine.

The decomposition offers advantages similar to those of the lift service decomposition. The separation of writers from readers applies both to the Lift Equipment Model and to the real Lift Equipment itself: the Maintain Lift Equipment Model machine monitors the Lift Equipment but does not affect its state, while the Ensure Lift Safety machine applies the brake and switches the motor off, but does not monitor the Lift Equipment state. The subproblem of maintaining the Lift Equipment Model is an instance of a standard problem class, in which a dynamic model is built and maintained to act as a surrogate for a dynamic causal domain. The subproblem requirement *Model* $\approx$ *Lift Equipment* stipulates that a certain correspondence must be maintained between the states of the Lift Equipment and the states of the Lift Equipment Model: essentially, the value of the boolean model variable *faulty* must be *true* if and only if the Lift Equipment has an equipment fault.

Finally, the decomposed *display* subproblem is shown in Fig. 2.6. As before, the standard decomposition introduces a new problem domain, the LB (Lift&Buttons) Model. To ensure that the lobby display correctly shows the current position of the lift car, its direction of travel and the currently outstanding service requests, the Maintain LBModel machine monitors the floor sensor states and the button press events which indicate requests; on the basis

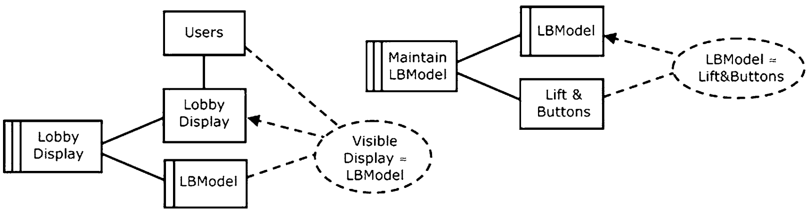


Fig. 2.6. The display subproblem, further decomposed

of this information and the problem domain properties of the Lift&Buttons domain, it maintains the LBModel, whose model variables are *current_floor*, *current_direction* and an *outstanding_request* variable for each floor. The Lobby Display machine uses the model to drive the display, ensuring that changes in the display are appropriately timed for convenient recognition by the users. For example, the current floor display must change smoothly, avoiding any hiatus in the display as the lift car goes from one floor to another or reverses its direction of travel.

2.3.4 Subproblem Independence and Subproblem Composition

In decomposing the development problem into these subproblems, we have wilfully ignored their interactions, considering each subproblem in isolation. For example, it is immediately obvious that in the event of an equipment fault, the Ensure Lift Safety subproblem will come into conflict with the Provide Lift Service subproblem: one of them will be required to lock the lift car in the shaft and the other to continue to move it up and down in response to service requests. Similarly, by separating the reader from the writer of the Priority Regime domain, we have ignored, or postponed, the question of their interleaving: after editing, how and when will the provision of lift service switch over to the newly edited rules?

We may regard this aspect of the decomposition as a separation of the *composition concerns* (how to put the subproblems together) from the *subproblem concerns* (how to develop a good solution to each subproblem). There are compelling reasons for this separation. One reason is that before considering how to put two things together it is wise to reach a clear idea of what the two things are. If the priority regime is trivial – for example, merely naming a single floor that is to be given service priority – it might be reasonable to switch over to a new priority regime as soon as it has been entered by the Building Manager; but if the rules are expressed in an elaborate structure specifying relative floor priorities for different days of the week and different hours of the day, then a more carefully designed interleaving of the subproblems would be desirable. Reported system faults, for example in the risks forum [1], often seem to have resulted from inadequate attention to the composition of parts that may be individually satisfactory when considered in isolation.

A second reason for the separation of composition from subproblem concerns is a desire to identify subproblems that fall into well-known, relatively simple classes, for whose solution a sound basis is known. For example, Edit Priority Regime is a *WorkPiece* problem, in which a human *User* (here, the Building Manager) issues editing commands to be performed by the machine, resulting in the editing of the lexical *WorkPiece* domain (here, the Priority Regime). Ensure Lift Safety is a *Required Behaviour* subproblem, in which the machine must impose a behaviour constraint on a *Controlled Domain* (here consisting of the Lift Equipment Model and the Brake & Motor). Maintain Lift Equipment Model is an instance of a *Build Dynamic Model* subproblem, in which the machine must monitor an autonomously active *Real World* domain (here, the Lift Equipment) and construct and update a dynamic analogic model whose states reflect those of the *Real World* in the required ways. Premature consideration of the eventual subproblem composition would complicate the subproblems, making them harder to recognise as instances of the well-known classes, or at least harder to handle by initial application of an available well-known solution.

In this paper we reflect the separation of composition from subproblem concerns by deferring any discussion of composition until after our discussion of the nature, treatment and verification of subproblems.

2.3.5 Normal Design

Recognising and treating subproblems as instances of well-known classes is, of course, an application of the standard mathematical approach of *reduction to a previously known problem.* But it is also a classic engineering approach attempting to bring as much as possible the development into the ambit of what Vincenti [42] calls *normal design.* Vincenti's examples are drawn from aeronautical engineering in the period from 1910 to the advent of the jet engine in 1945. Normal design is not a single generalised engineering discipline: there are multiple normal design disciplines, each specific to its particular subproblem class, that is, to what Vincenti calls each type of device. In normal design

> "... the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task".

In a *radical design*, by contrast

> "...how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development".

The foundation of a normal design discipline is an *operational principle* and a *normal configuration* that has been found to embody the operational principle very effectively. An example of an operational principle is Sir George Cayley's famous enunciation in 1809 [7] of the principle of the fixed-wing aircraft: “to make a surface support a given weight by the application of power to the resistance of air”. The normal configuration is the shape and arrangement of the device’s parts. For example, it was found that Cayley’s principle was best embodied in a monoplane aircraft with wings to provide the main supporting surface, a tail and rudder placed behind the wings and propulsion provided by an engine that pulled the aircraft from the front rather than pushed it from behind.

An obvious engineering example of successfully evolved normal design is the motor car. Karl Benz’s first car, built in 1886, had three wheels, the driver sitting in the middle and steering by a tiller linked to the single front wheel. The single front wheel was necessary because it was the only available alternative to the hopelessly cumbersome drawbar steering used on horse-drawn vehicles. This car was a radical design. It was not a dependable and practical vehicle, but was successful precisely in the sense that it functioned “well enough to warrant further development”. Benz’s next car was a four-wheeler, steered by two separately pivoted front wheels linked by the novel Ackerman steering geometry: this geometry avoided drawbar steering while solving the problem of keeping the plane of each wheel tangential to its path on the road. The problem of positioning the driver remained open for a few more years. Some early cars positioned the driver at the back, steering by a tiller; in some, the driver sat sideways, and in one design, the driver stood on a step behind the rear seats. By 1910, driver position had become a “customary feature”, the driver sitting in front on the offside and steering by a raked steering wheel. Over the following half century, the design of motor cars evolved to embody what were once novelties but are now customary features: fully enclosed body (1910), four-wheel brakes (1920), independent front suspension (1933), integral body and chassis (1938) and automatic gearbox (1940).

The canons of normal design, established by long experience for each device type, contribute to dependability and other desirable qualities in several ways. First, they fix, and make available to the whole community of specialised designers of the devices in question, inventions and discoveries that have been found advantageous. Second, they embody choices that avoid critical failures that have occurred in earlier designs. For example, if the fuel tank of a car is placed low down behind the rear axle under the luggage compartment, it can be easily ruptured in a rear-end collision, leading often to a fatal fire. Incidents of this kind eventually led to an explicit recognition that the fuel tank must be positioned in a safe place, for example, in “the area cradled by the rear wheels, above the rear axle and below the rear window [38]”, and such positioning became an integral part of normal design. Third, the normal design practice enjoins careful attention to *concerns* that must be addressed not by adopting a standard design choice but by explicitly investigating the properties of the particular design in hand. For example, since the notorious failure of the Tacoma Narrows Bridge in 1940, every civil engineering student is shown the filmed record of the collapse and is taught a very specific lesson: mechanical resonance and aerodynamic effects can lead to wind-induced oscillation of very large magnitude, and avoiding this possibility is a vital design concern.

2.3.6 Verification and Subproblem Classes

Subproblem decomposition aims to identify subproblems of known classes and accordingly insists on the exclusion of certain forms of complexity as a governing principle. When a complexity of one of these forms is unavoidable, it is, wherever possible, to be regarded as a composition concern in combining subproblems, and dealt with explicitly in that way. For example, since common problem domains having distinct writers and readers are always written and read in separate subproblems, they cause no interference concern within a single subproblem. More generally, the properties of a problem domain in a subproblem are always those seen from the point of that subproblem only. This principle excludes from each separately considered subproblem the kind of complexity that arises in programming from using certain old-fashioned programming language features such as Fortran *equivalence* or COBOL *redefines.* These features allowed the programmer, knowing how the compiler represents the language's built-in data types, to treat the same storage locations as having simultaneously the properties of two or even more types: for example, an array of four bytes might also be regarded as an integer. Such features are now recognised to be dangerous. They are available only in obsolete languages and language versions, and even when available are avoided in sensible programming practice. They would also present additional difficulties to a verification tool for two reasons. First, to check the correctness of a superimposition of distinct data types, it is necessary to appeal to the representation function by which they are mapped to the underlying hardware store. Second, a program that uses a superimposition of two data types is in effect a combination of two simpler concurrent programs, each using one of the data types, and presents all the potential difficulties of interference. Avoiding such complexity in the treatment of problem domains for each individual subproblem would simplify verification of the entailment $\mathcal{S},\mathcal{W} \models \mathcal{R}$ or $\mathcal{M},\mathcal{W} \models \mathcal{R}$.

Explicit assignment of each identified subproblem to a known class can help verification in other ways also. Tractable program verification of all but the smallest programs inevitably depends on a degree of normal design practice and on its support by explicit statements in the program text or in accompanying design documentation. By announcing "this is an integer function" or "this is a monitor" or "this is a text file", the programmer or designer exploits the normal design of the type in question and willingly submits to the associated restrictions. The developer of a monitor need not be concerned with the intricacies and risks of low-level operations on semaphores. The programmer who declares a text file need not be concerned with detecting line breaks at the character level. Declaring the type and accepting the restrictions allows programming at a higher level of abstraction, and permits verification tools to exploit the known properties of the type in checking correctness.

In these respects, subproblems of known classes may be regarded as instances of types. The types themselves must be adequately studied, and appropriate linguistic support must be devised to characterise their parts, properties and concerns. Consider, for example, the (highly simplified) *Work-Piece* problem class of which Edit Priority Regime in Fig. 2.4 is a member. A simplified WorkPiece problem has two problem domains: a User and a WorkPiece. The requirement is that the WorkPiece must be edited – including creation – in response to editing commands entered by the User. The WorkPiece problem domain must be *lexical*: that is, it is a reified symbolic object whose state changes only in response to externally controlled *create*, *access* and *update* events. The User problem domain must be *biddable*: that is, a human operator who at any time may cause any one of a repertoire of available command events. These domain characteristics are integral to the definition of a WorkPiece problem, and must be checkable in the description $\mathcal{W}$ of problem world properties that forms a part of the subproblem documentation to be verified. They give the developers both assurances and responsibilities: an assurance that the WorkPiece domain will not cause any events and will never change state spontaneously on its own initiative; and an obligation to specify the machine's behaviour in response to any sequence whatsoever of command events caused by the User.

A substantial benefit, analogous to the benefits of types in programming, can be obtained in this way. Subproblem types differ from programming types in one important way that diminishes, but does not destroy, their value in verification. A program describes, somewhat abstractly, a computer whose behaviour will have been specialised by the compiled machine-code text. This machine-code text is created by the compiler in accordance with the programming language definition. It will be interpreted by the computer hardware, which is highly reliable. So for many practical purposes, there is no significant possibility, apart from errors in the compiler and operating system software, that the computer's specialised behaviour will fail to correspond with – that is, will not be correctly described by – the program. However, for a subproblem in a software-intensive system many of the crucial descriptions in $\mathcal{W}$ describe parts that belong to the physical and human problem world. (This is not strictly true of lexical problem domains such as Priority Regime and Lift Equipment Model. They are problem domains for the subproblems in which they appear, but may be regarded as local variables of the undecomposed *service* and *safety* subproblems, and in that sense share some of the properties of programs.) These problem world parts are not created by a compiler from their descriptions, but are given in the world outside the computer. The truth or falsity of their descriptions cannot be checked by software tools, but only by direct comparison of the description with the problem world itself. Nonetheless, even in this matter, tools can give developers some useful help. We will return to this topic in a later section.

2.3.7 Subproblem Concerns

Each subproblem class presents a number of *concerns* – matters that must be addressed by the developers if a satisfactory solution to the problem is to be obtained. While some concerns arise in several subproblem classes, some arise only in particular classes or only in classes having problem domains of particular characteristics. Some examples of subproblem concerns are:

- Initialisation: The subproblem machine will begin execution in its initial program state at some point in time, and at that moment, the problem world will be in some particular state. The initialisation concern is to ensure that the machine and problem world states are compatible according to the subproblem solution design. For example, the *safety* subproblem solution may assume that initially the lift car is at the ground floor with the lift and lobby doors *open*, the motor *off* and the motor polarity set to *up*. If this assumption is false, the system is likely to fail. A different design of the subproblem solution might make no assumption about the initial state of the problem domain, the machine detecting the current state and initialising its own variables accordingly.
- Breakage: A causal problem domain sharing events controlled by the machine may be broken by certain sequences of those events. For example, the lift equipment may be broken if the motor polarity is reversed while the motor is running. The breakage concern is to ensure that no such sequences are caused.
- Identities: A problem domain containing multiple entities of the same kind presents a potential for failure in which one entity is mistaken for another. For example, in a centralised system to monitor patients in a hospital's intensive-care unit, the patient is connected to the central computer by a chain of several links: a thermometer or other sensing device is attached to the patient, and the device is plugged into a machine port, perhaps through some kind of multiplexer; the monitoring criteria may be specified by patient name, or patient identifier, or bed identifier. It would be a serious failure if the monitoring criteria for one patient were in fact being applied to another. The identities concern is to ensure that this does not happen.
- Completeness: A subproblem machine connected to a biddable problem domain and sharing events controlled by that domain must be designed to provide an appropriate specified response to any sequence whatsoever of events caused by the biddable domain. A machine connected to a causal domain and sharing events controlled by it must be designed to provide a specified response to any sequence permitted by the description of the domain's properties. Buffer overrun on input of an overlength string is an example of a completeness failure.

It is clear both from these examples and from the preceding section that control of shared phenomena – which of the sharing domains can cause the shared event or can change the shared state – is of central importance.

The machine specification and problem properties descriptions must make control explicit and therefore checkable by the verification tools.

2.3.8 Verifying Subproblem Concerns

In some simple cases, failure to address a subproblem concern can be identified by failure to prove the entailment $\mathcal{S}$, $\mathcal{W} \models \mathcal{R}$ or $\mathcal{M}$, $\mathcal{W} \models \mathcal{R}$. Suppose, for example, that the description $\mathcal{W}$ of the Lift Equipment properties in the service subproblem of Fig. 2.4 shows that the motor *on/off* state and *up/down* polarity are changed only by externally controlled events, and also that the behaviour of the Lift Equipment is specified by $\mathcal{W}$ only for sequences in which each polarity change is preceded by a motor=*off* period of length at least *motor_rest*. Then if the machine behaviour specification $\mathcal{S}$ allows the machine to cause event sequences that do not conform to this restriction, there is certainly an unaddressed breakage concern. A different approach to breakage concerns would require each description of the properties of a causal domain having externally controlled shared phenomena to specify the domain's response to all sequences of those phenomena. Some of these sequences would be described as leading to a conventionally identified *broken* state, from which the domain might exhibit any subsequent behaviour whatsoever. To prove that the breakage concern has been adequately addressed is then simply to prove that the broken state is not reachable.

In other cases, it will be necessary to use a very different approach, drawing the developers' attention to concerns that may or may not have been addressed. The output of the tools would have the general form of identifying a proof obligation to be discharged by the developers. This approach is similar in spirit to the technique used in the SPARK Ada Examiner [5]: that tool, for example, marks simple integer assignments as requiring manual proof of the absence of arithmetic overflow when it cannot provide a formal proof automatically. The following sections illustrate this approach for two concerns: initialisation and completeness.

2.3.9 Initialisation Concern

A potential initialisation concern arises whenever the behaviour of the subproblem machine interacts with the behaviour of a problem domain: the design of the machine and the satisfaction of the requirement may rely on assumptions about the initial state of the problem domain – that is, its state when the machine execution begins. There are several different ways of addressing this concern. For example:

- The machine design makes no assumptions about the problem domain state.
- The machine checks the initial problem domain state and does not execute its main function if the check fails.

- The machine forces the problem domain into a desired initial state before executing its main function.
- The machine execution is always preceded by execution of another subproblem machine, which always leaves the problem domain in the assumed initial state.
- A human procedure is stipulated for setting or checking the assumed initial problem world state before starting the machine.
- A human procedure is stipulated for informing the machine of the current problem domain state before starting the machine.

Whether the initial problem domain state is compatible with any assumptions made in the machine design can be checked from the descriptions of the subproblem alone. Relying on preceding execution of another subproblem machine is a matter of subproblem combination: correct handover from the preceding machine to the machine being considered would be a *switching* composition concern (which is discussed in the following section).

Where a human procedure is involved, there is an obligation on the developers to demonstrate that the procedure is reliable, the required degree of reliability depending on the criticality of the system in general and the subproblem in particular. The existence of this obligation can be determined by analysis of $\mathcal{W}$. If $\mathcal{W}$ asserts an initial domain state, there is an obligation on the developers to demonstrate that state holds when machine execution begins. Naturally a demonstration that relies, for example, on adherence to correct procedure by maintenance or installation engineers cannot be in any sense formal, although the system could follow such rules of good practice as insisting on confirmation before the subproblem machine can be started.

2.3.10 Completeness

In a completeness concern, the machine must be designed to deal with all possible sequences of shared events or state changes controlled by a problem domain to which it is interfaced. All possible sequences being shown in the domain property description $\mathcal{W}$, the verification tools must check, as a part of the basic verification of the subproblem development, that the machine does have specified responses to all of those sequences.

However, more can be done to check whether the specified machine behaviour is the result of sufficiently careful consideration. In a WorkPiece problem, the machine behaviour may be specified for some user command sequences explicitly but for others only by implication. Suppose, for example, that machine behaviour specification is structured in terms of the effects of individual user-controlled commands, as it might be in Z. If every operation has the precondition *true*, then no sequence is excluded, and machine behaviour is specified for all command sequences. The central difficulty then is that formal analysis of the requirement, domain properties and machine behaviour cannot in general distinguish intended from unintended machine behaviour in the specification.

The point is clearly illustrated in the failure of a small weapons system [32]. A “plugger” (a handheld GPS device) was used to target a missile. After setting the desired target location, the user saw that the battery was low and changed the battery. Having inserted the new battery, the user, on sending the *fire* command, was directly targeted and killed by the missile. The device had been designed to set its own current location in the location register when restarted by insertion of a fresh battery.

This design error would not be reliably avoided by the practice of inserting redundant assertions into the machine specification or requirement text, precisely because the designers had, evidently, not envisaged the possibility of such a failure and so would probably not have taken steps to exclude it by appropriate assertions. The error might, however, be detected by a technique specifically aimed at addressing the completeness concern. The developer could be invited – or required – to identify one or more phenomena that are crucially related. In the plugger case, this might be the *fire* command and the *target location* at which the missile would be fired; in a WorkPieces problem, it might be the delete command and the *text selected* to be deleted. The verification tool would then display all possible instances of the association at some level of abstraction high enough to make this possible. In the plugger case, this would show that the target location associated with a *fire* command has not necessarily been entered on the device’s keyboard but might have been set by the internal *restart* procedure that is automatically invoked when a new battery is inserted. The developer, invited to review the presentation of possible instances, may then recognise a previously neglected possible failure.

2.4 Combining Subproblems

Because analysis of each subproblem is conducted, at least initially, in isolation from the other subproblems of the system, combining the subproblems is not a simple implementation task of linking components in a uniform architectural framework. Combining the subproblems is an explicit development task that raises its own composition concerns. In general, combining subproblems means not only combining their machine executions, but also combining their requirements and reconciling their views of the problem domains. Addressing these composition concerns may demand “invasive composition” – that is, modifying previously designed subproblem solutions, and may demand the creation of further machines for which the only, or most significant, problem domains are the machines of the already analysed subproblems.

Subproblem machines interact within the combined machine, and also through any problem domains that are common to their problem worlds. So, for example, the Lobby Display machine is affected by the Lift Service machine through the Lift and Buttons domains, the Maintain LBModel machine and the LBModel domain. This particular interaction is entirely beneficial, allowing the lobby display to reflect the behaviour of the lift as it moves in satisfaction of the service requirement; but many subproblem interactions, like many feature interactions in telecommunications services, are potentially damaging. Considering, and if necessary managing, modifying or mitigating these interactions means addressing composition concerns.

2.4.1 Composition Concerns

Composition concerns arise, and must be explicitly addressed, wherever subproblems interact through common problem domains. Perhaps the most obvious example is the possibility of *interference* when writers and readers share state. In Fig. 2.6, the LBModel domain has the Maintain LBModel and Lobby Display machines as writer and reader. Some form of mutual exclusion is necessary to ensure that the Lobby Display machine is never accessing a lexical LBModel that fails to conform to its domain description because it is simultaneously being updated by the other subproblem. There are more ways than one of ensuring mutual exclusion; verifying that mutual exclusion is guaranteed is a task for conventional program verification of the composition.

2.4.2 Switching

Subproblems must be composed in respect of their machine execution. There is a natural inclination on the part of a conscientious developer to enlarge the span in time of each subproblem requirement – and hence of the machine execution that ensures its satisfaction – to the maximum possible. This is both natural and salutary because too small a span can easily lead to errors of the kind made in the development of the plugger system [32], while too large a span is unlikely to do significant harm. But there can be no *a priori* assumption that execution of every subproblem machine should be coterminous with execution of the complete system and thus coterminous with the execution of all other subproblem machines. Many systems must operate in different modes at different times according to the progress of the execution, and the span of one subproblem machine execution may be confined to a single occurrence of a single mode.

This kind of *mode-based* operation leads to *switching concerns*. A common problem domain must be – so to speak – handed over from a machine that is terminating to one that is starting. There is, therefore, an *initialisation concern* for the starting machine and a *terminating concern* for the terminating machine to ensure that the domain is handed over in a permissible state that is compatible with the starting machine. A simple version of a switching concern occurs when a subproblem initialisation concern is handled by introducing a separate initialisation subproblem. For example, the Lift Service subproblem might be further decomposed into a Lift Initialisation subproblem, which from any starting state of the Lift Equipment domain brings the lift car to the ground floor and opens the doors, and a Lift Operation subproblem which assumes that initial state and goes on to provide lift service. There would then be a switching concern for the transition from Lift Initialisation to Lift Operation.

2.4.3 Requirement Conflict

Another important composition concern is *requirement conflict*: the requirements of two subproblems may be directly contradictory. The Service and Safety subproblem requirements exhibit such a conflict. In certain possible circumstances, where there is an outstanding request for lift service and a malfunction has been detected, the Service machine may be required to move the lift car while, at the same time, the Safety machine is required to lock it in the shaft. In principle, verification can detect such conflicts automatically, but their resolution must be a matter for the developers. The resolution depends on the relative importance of the conflicting requirements. In this simple case, the resolution is obvious: safety is far more important than service, and must, therefore, take precedence: when a malfunction has been detected, lift service will be abandoned. (In fact, this analysis is implausibly oversimplified. If the hoist cable has broken, it is necessary to lock the lift car in the shaft immediately; but some lesser malfunctions would be better dealt with by less drastic action – for example, by bringing the car to the ground floor and only then abandoning lift service.)

2.4.4 Mode-Based Operation and Domain Properties

When different subproblems assume different views of a common problem domain, a composition concern arises: the different domain descriptions must be reconciled in a way appropriate to the composition of the subproblem machines. Consider, for example, the appearance of Lift Equipment as a common domain in the service and safety subproblems in Figs. 2.4 and 2.5. For the service subproblem, the Lift Equipment is assumed to satisfy a description that we may call *workable*: essentially, the equipment is working correctly within close enough tolerances to enable reliable lift service. For the safety subproblem, the assumed Lift Equipment properties of interest are those that capture the disjunction of what we may call *healthy* (no malfunction) and *unhealthy* (some malfunction) behaviours of the equipment, and relate them to the monitored behaviour at the interface with the Maintain Lift Equipment Model machine. The Ensure Lift Safety subproblem assumes a property of the Brake and Motor domain that we may call *stoppable*: switching off the motor and applying the emergency brake will lock the lift car in the shaft.

The system will operate, essentially, in two phases. In phase 1, the Provide Lift Service machine operates concurrently with the Maintain Lift Equipment Model machine and the Ensure Lift Safety machine; the former is updating, and the latter is monitoring, the Lift Equipment Model, but no malfunction has yet occurred. In phase 2, when an equipment malfunction has been detected, the Ensure Lift Safety machine and the Maintain Lift Equipment Model machine will execute, but not the Provide Lift Service machine. It is necessary to verify that the following relationship always holds among the domain descriptions:

$$(\textit{healthy} \Rightarrow \textit{workable}) \wedge (\textit{unhealthy} \Rightarrow \textit{stoppable}) \wedge (\textit{healthy} \vee \textit{unhealthy}).$$

The first conjunct applies in phase 1, the second in phase 2 and the third in both phases. Falsity of any of these conjuncts gives good reason to believe that the system is potentially dangerous: it shows that circumstances can arise in which some subproblem machine is executing at a time at which some of its assumptions about its problem world are false.

2.4.5 Relative Criticality

As a final example of a composition concern, we may consider *relative criticality.* A dependable system is not necessarily one in which every function is perfectly dependable (a goal that is in any case unattainable). A practical and more useful criterion of dependability is that every system function is sufficiently dependable for its degree of criticality: the most critical functions must be the most dependable. This principle provides both a heuristic for problem decomposition and a composition concern. In decomposition, it forbids the combination of more critical and less critical functionality in a single subproblem. As a composition concern, it requires the developer to ensure that execution of a more critical subproblem machine is not dependent on a less critical subproblem machine. A striking illustration is given by a version (now fortunately superseded after analysis had already revealed the fault) of the software design for a proton-beam therapy system. Two functions of the system are *command logging* and *emergency shutdown.* The command logging requirement stipulates that every command sent to the equipment must be logged to disk for audit purposes; the emergency shutdown requirement stipulates that when the emergency button is pressed, the beam must be blocked and shut off. Since shutdown is a command, it is routed through the logging module. Unfortunately, when the disk is full, the logging module does not operate correctly: it hangs, and neither returns to the caller nor passes on the command to the relevant actuator. In effect, a failure in execution of the command logging function can disable the emergency shutdown.

The point of this example here is not, of course, that the logging module should have been error-free. It is rather that the shutdown function should have been maximally independent of the logging module and, indeed, of every other part of the system. If system functions – that is, the subproblems – are associated with stated levels of criticality, then an automatic tool can trace dependency paths and show where and how more critical functions had been made dependent on less critical functions.

2.5 Non-Formal Problem Worlds

Brian Cantwell Smith in [39] discusses the relationship among three parts or aspects of a system: the designed part or *computer*, the problem world, or *real world*, and the formalised *model* of the real world on which the development is based (the *formalised model* is, of course, the formal description $\mathcal{W}$ of the problem world properties). His representation of the relationship is shown in Fig. 2.7.

He writes: "One way to understand the model is as the glasses through which the program or computer looks at the world: it is the world, that is, as the system sees it (though not, of course, as it necessarily is)". He observes that model theory is applicable to the left side relationship between the computer and the model and adds that "at this point in intellectual history", we have no theory of the right side relationship between the model and the real world. In the terminology we are using here, he is pointing out the lack of a theory of the formalisation of non-formal worlds. Model theory is a purely formal discipline and we should not expect a theory of formalisation to be equally formal. But it is equally – or even more – important, and we should at least seek techniques of constructing more reliable formalisations, and of finding errors in our use of them in development.

2.5.1 Alphabets and Designations

Any formalisation of a physical problem world is based on an *alphabet*: an enumerated set of formal terms that are understood to denote phenomena of the world. The denotation of each term in the world must be explicitly described by a rule stating, in natural language, how instances of the denoted phenomena are to be recognised. Each formal term, with its accompanying recognition rule, constitutes a *designation*. Almost always designation involves abstraction. To designate *MotorOn* as an *event* controlled by the machine and shared by the Lift Equipment domain is to abstract from the causal chain that runs from the machine port, to the relay that switches the motor power, to the flow of current through the motor windings, to the consequent rotary motion. Whether this particular abstraction is appropriate depends on the purpose of the formalisation and on the reliability and speed of the causal chain in the problem world.

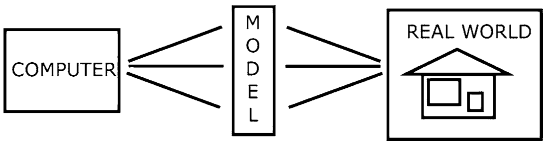


Fig. 2.7. The computer, the model and the real world

Designations must be distinguished from *definitions*. A designation picks out a new class of observable phenomena in the world and gives it a newly created formal name. A definition, by contrast, picks out no new phenomena, but merely creates a new formal name and states its meaning in terms of previously created formal names. So having designated $\mathit{person}(p)$ and $\mathit{childOf}(p, q)$, we can use them to define $\mathit{siblingOf}(p, q)$. In an abstract mathematical universe, this distinction between designation and definition may sometimes be of only minor importance; but it is always of vital importance in creating and interpreting formal descriptions of a physical problem world. A carelessly expressed definition can cause confusion and error by masquerading as an assertion of a problem world property.

Because the recognition rule in a designation is itself informal, there always is scope for uncertainty in interpreting the formal terms and hence in validating formal descriptions that rest on them. This uncertainty is one contributor to the necessarily approximate nature of any formalisation. The degree of uncertainty depends partly on the clarity with which the recognition rule is expressed, but it depends far more on the boundaries of the problem world and on the extent of the properties and requirements to be formalised. The denotation of a formal term $\mathit{mother}(p, q)$ may be perfectly clear in a problem world consisting solely of the patients of the labour ward in a maternity hospital. In the wider context of social legislation, it may invite queries about adoptive mothers, surrogate mothers, genetic mothers, stepmothers and other putative mothers. In constructing a formalisation, it is, therefore, an important concern to designate only phenomena that can be designated with a very high degree of clarity in the context of the problem at hand.

2.5.2 Formal Reasoning in Subproblem Composition

A non-formal world presents serious obstacles to formal verification. Formal reasoning about formalised descriptions of the problem world can show the presence of errors, but it can never show their absence. This is a truth well-known to the established branches of engineering. If analysis of the formal model of a product or of its environment shows that the product will fail, then it almost certainly will. But if analysis shows that the product in its environment will be satisfactory, there are still many grounds for doubt. This is one reason for the use of safety factors, which are often referred to as "factors of ignorance". In software-intensive systems, where problem domains are often discrete rather than continuous, safety factors are harder to devise and apply. The analytical model provided by the problem world properties description $\mathcal{W}$ governs the system behaviour more tightly than we may always recognise.

One source of ignorance is the approximate nature of formalisations based on designated phenomena. The fuzziness of each designation introduces an error term into the calculation because it admits some probability of an awkward instance that will be wrongly classified or will otherwise furnish a counterexample to the formalisation, with unforeseen consequences. These error terms, like the errors due to floating-point representation of reals in numerical calculations, may accumulate to give an erroneous result. The experience embodied in normal design practice serves as a guard against this kind of error: for some particular kind of problem domain in some subproblem class, a particular formalisation based on a particular set of designations is known to work well.

Another source of ignorance is potentially important in subproblem composition. Combining two subproblems combines their views of common domains, formalised in their separate descriptions of problem world properties. In the words of Bill Addis, an experienced structural engineer [4]:

> "It must never be forgotten, however, that the primary models of loads, materials and structure are all idealisations and simplifications of the real world, and the behavioural output of the composite model is merely an infallible consequence of the information contained in the primary models, not of their real-world counterparts. Any predictions made from the output of the composite model about the likely behaviour of the completed structure must be treated with intelligence and care".

In effect, the $\wedge$-introduction rule of logic may not hold. That is: two problem world descriptions *w1* and *w2* may be separately true, in the sense that for its particular purpose each one is a close enough approximation to the same reality; but their conjunction *w1* $\wedge$ *w2* may yet be too far from the truth to exclude failure. The approximation here may lie not in fuzzy designations, but in what has been ignored. The physical and human problem world, at the scale that concerns software-intensive systems, offers an unlimited repertoire of phenomena. To choose the alphabet for describing a problem domain is to exclude an unbounded set of phenomena that have been judged, explicitly or implicitly, to have negligible relevance to the properties of interest for each description. When the separate descriptions are put together, some of the neglected phenomena may be no longer negligible in their effects.

Can software tools offer useful assistance in the exercise of the intelligence and care of which Addis writes? Certainly they cannot examine the real world directly to check its conformity with the developers' formalisations. But perhaps formal techniques can be devised for recognising likely locations of failure in compositions of problem domain descriptions, and for drawing the developers' attention to what are, in effect, their implicit assumptions about compositionality of problem world properties. To suggest specific techniques here would go beyond the limits of plausible speculation; however, it may prove useful to investigate sets of descriptions that are associated with known failures in problem worlds and to seek patterns in their formalisations. These patterns may be found in the set-theoretic relationships between the alphabets of two descriptions and those of other descriptions. They may be patterns in graphs representing the causal chains asserted by the descriptions. And there are, no doubt, other possibilities.

2.6 Concluding Reflections

In many areas, surprisingly casual and uninformed approaches have characterised much software development, even in such critical systems as the notorious Therac-25 [30]. It is clear that major improvements are urgently needed. Sixty years of impressive work on program correctness, both in correctness-by-construction and in *post hoc* verification, have naturally engendered optimism and a high degree of confidence among the verification community. It does not seem unreasonable to hope that application of the same or very similar approaches will achieve dependability in the software-intensive systems that are becoming ubiquitous, and at the same time, increasingly complex and increasingly critical.

This hope should not rest on the belief that a substantial reduction in pure programming errors, achieved by a more widespread and determined application of narrow verification, will in itself produce a comparably substantial improvement in system reliability. That belief is not well supported by the evidence. Estimates of the huge cost of system failures (for example in the report by the US Department of Commerce [3]) do not analyse the sources of the failures whose effects they claim to measure: in particular, they do not distinguish those failures that could be eliminated by narrow verification. A paper by MacKenzie [33] pointed out the difficulty of making this distinction: "Typically, neither electromagnetic interference nor software error leave physical traces ... . Their role can often only be inferred from experiments seeking to reproduce the conditions leading to an accident. While this can on occasion be done convincingly, it is sometimes far from easy, and the suspicion therefore remains that these causes are underreported". Given this caveat, MacKenzie estimated that for a large sample of computer-related accidental deaths the proportion that could be confidently ascribed to programming error was about 3%.

Realistic assessment of the possibility of extending program verification techniques to improve system dependability must rest on a serviceable understanding of the ways in which a verified program and a dependable system differ, and the ways in which they are the same – or, at least, can be striven for with the help of similar intellectual and other tools. One important point of similarity has been almost irretrievably obscured by the heated controversy aroused by Fetzer's famous paper [10] "Program Verification: The Very Idea". Verification, in the sense of an irrefutable mathematical proof of correctness, is clearly a notion that cannot be applied to the physical world at the scale of interest. Both program executions and problem world properties and behaviours are intrinsically grounded in the physical world, where, at the level of human, mechanical and electrical phenomena, there are always unbounded possibilities of counterexamples to any universally quantified assertion. Verification, whether narrow verification of a program against a specification or wide verification of a system against a requirement, can prove correctness only in the mathematical domain of the chosen abstractions.

Whether the chosen abstractions correspond well enough to the physical reality is a very different question. For computer programs, the hardware engineers have done their work so well that only in the most critical systems it is necessary to consider the possibility of hardware failure at the instruction execution level. But in the problem worlds of software-intensive systems, equipment malfunctions and similar failures are common occurrences. Such incidents are the central topic in the construction of safety cases for critical systems [6, 27]. Even when a system has been carefully designed for a chosen level of fault tolerance, there is inevitably a residue of possible failures that have not been accommodated. In considering the given problem world properties, it is an advantage of the rely-guarantee perspective that this residue of possible failures is made explicit. The description $\mathcal{W}$ of the Lift Equipment domain shown in Fig. 2.5 makes explicit the assumed relationship of equipment failures to their detectable symptoms on which the design of the safety subproblem relies. The residue of failures falling outside these assumptions lies in the unbounded complement of the description $\mathcal{W}$.

Another point of similarity is the importance of normal design. Normal design has been emphasised here as a potential contributor to wide verification – that is, to system dependability; it also plays a major (though largely unrecognised) role in specification, including the specification of programs that are the object of narrow verification. For an archetypical small programming example, such as the integer function $GCD(x, y{:}integer){:}integer$, it seems easy to give a complete specification: "for positive integers x and y, $GCD(x, y)$ is the largest integer that divides both x and y". In the abstract world of integer functions, this specification is complete: it states both necessary and sufficient conditions for acceptability of the function value. In the world of real program executions, even in a perfectly reliable computer, it is very far from complete. It is at best a statement of one necessary property of the program execution: a crucial property, indeed, but certainly not sufficient. Even assuming correct behaviour of the hardware and system software, a perverse programmer could still deliver an implementation that calculates the specified value but is in other respects unacceptable. It might use too much storage, or use a novel algorithm that computes an intermediate result greater than either argument and causes arithmetic overflow; or it might gratuitously start a new thread or access the disk or even the internet. Essentially, there is no bound to the set of implicit conditions for acceptability. We would castigate as perverse the programmer who fails to satisfy these implicit conditions, arguably because there is an unspoken normal design of "small integer functions", which implicitly allows none of the unacceptable behaviours. In the presence of such a normal design discipline, a desired instance of the device class can be adequately specified by a bounded alphabet – here, the range and type of the integer arguments. The device class name implies the standard configuration and implementation of the class and thus, carries the main weight of the unbounded implicit part of the specification.

Narrow verification assumes the existence or availability of a formal machine specification. For a subproblem of a normal class, it may be reasonable to assume that a machine specification $\mathcal{S}$ can be obtained at some stage of the development process, and that program correctness, $\mathcal{M} \models \mathcal{S}$, can be usefully separated from system correctness, $\mathcal{S},\mathcal{W} \models \mathcal{R}$. The unbounded part of the specification $\mathcal{S}$ is carried by the normal design discipline, which governs the machine implementation as well as the properties and behaviour of the problem world. For a system that is a radical combination of subsystems, it is not clear that an adequate machine specification $\mathcal{S}$ can be obtained, even if each subsystem is a normal device. Certainly, as each composition concern is considered in turn, an increment can be added to a growing partial machine specification. But this is very different from the unbounded specification implicit in the standard implementation of a normal design, because it has been created by the relatively small number of increments made in the course of the current project, rather than by the much larger number made in a long history of many developments during which a normal design evolves. In the normal design of a complex product, the composition concerns have already been addressed, including many whose established impact on the design is no longer present in the consciousness of the designers. Normal design of a car mandates not only the form and design of engines and gearboxes, but also the design and even implementation of their composition. By governing just how they should be interfaced and combined to make a dependable car, it both ensures satisfaction of the explicit, positive requirements and avoids the many undesirable properties and behaviours that have been identified in the diagnosis of past failures.

It is neither surprising that the established branches of engineering are so highly specialised, nor that normal design and standard products play such a large role at every level in their practice. This is not standardisation in the sense of interchangeable or interoperable parts – although that is often an important goal in its own right – but the standardisation of problem and solution classes and of the design procedures by which dependable solution instances can be obtained. One symptom of the extent of this standardisation within each problem or product class is the depth and detail of further structuring and classification, and the richness of the vocabularies used. The vocabulary of bridge design, for example, includes such terms as *cantilever*, *truss*, *arch*, *girder*, *beam*, *rigid frame*, *suspension*, *lifting*, *swing*. Each one has many variants: *fixed arch*, *one-hinged arch*, *ribbed arch*, *trussed arch*, *tied arch*, *arch truss*, *lattice truss*, *Smith truss*, *Pratt truss*, *bowstring truss*. These terms denote normal design choices at different levels. Each design choice is associated with rules of thumb, calculation techniques and data tables for guiding the designer to appropriate choices in each situation and for determining the loads in particular designed instances.

Too few areas of software development exhibit anything approaching this richness. Validating a design should depend crucially on the standard properties and structures of the specialised class to which the product belongs, and on applying its normal design disciplines and practices. The level of specialisation and structuring implicit in the suggestions made earlier for a discipline of decomposition and for the use of verification tools is at most an early step towards the goal of dependability in software-intensive systems.

Acknowledgements

I am very grateful to Daniel Jackson and Cliff Jones, who by reading and commenting on earlier versions of this paper helped me to reach a more coherent perspective on its argument.

References

1. `http://catless.ncl.ac.uk/risks`.
2. `http://www.praxis-his.com/reveal`.
3. The economic impacts of inadequate infrastructure for software testing. *US National Institute for Standards and Testing*, May 2002.
4. B. Addis. *Creativity and Innovation: The Structural Engineer's Contribution to Design.* Architectural Press, Oxford, 2001.
5. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, Reading, MA, 2003.
6. P. Bishop and R. Bloomfield. A methodology for safety case development. In F. Redmill and T. Anderson, editors, *Industrial Perspectives of Safety-Critical Systems: Proceedings of the Sixth Safety-Critical Systems Symposium.* Springer, New York, 1998.
7. G. Cayley. On aerial navigation. *Nicholson's Journal*, November 1809, February 1810 and March 1810; available at `http://invention.psychology.msstate.edu/i/Cayley/CayleyP1.html`, `http://invention.psychology.msstate.edu/i/Cayley/CayleyP2.html`, and `http://invention.psychology.msstate.edu/i/Cayley/CayleyP3.html`.
8. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.
9. E. W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, 1989.
10. J. H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.
11. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Symposium in Applied Mathematics*, vol. 19, pp. 19–32. American Mathematical Society, Providence, 1967.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.
13. H. H. Goldstine and J. von Neuman. Planning and coding of problems for an electronic computing instrument. Report prepared for U.S. Army Ordnance Department (Republished in A. H. Traub, editor, *John von Neuman: Collected Works*, vol. V: Design of Computers, Theory of Automata and Numerical Analysis, Pergamon Press, 1963, pp. 80–151), 1947.

14. J. Hammond, R. Rawlings, and A. Hall. Will it work? In *5th IEEE International Symposium on Requirements Engineering*, pp. 102–109. IEEE Computer Society Press, Los Alamitos, CA, 2001.
15. I. J. Hayes. *Specification Case Studies.* Prentice-Hall, Englewood Cliffs, NJ, 1986.
16. I. J. Hayes, M. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods: Proceedings of FME03*, Lecture Notes in Computer Science, vol. 2805, pp. 154–169. Springer, 2003.
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
18. C. A. R. Hoare, C. B. Jones, and B. Randell. Extending the Horizons of DSE (GC6). Technical Report CS-TR-853, University of Newcastle upon Tyne, 2004.
19. C. A. R. Hoare and J. Misra. Verified Software: Proposal for a Grand Challenge Project (draft). Technical Report, July 2006.
20. T. Hoare and R. Milner, editors. *Grand Challenges in Computing Research.* British Computer Society, Swindon, 2004.
21. M. Jackson. *Software, Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, Reading, MA, 1995.
22. M. Jackson. Problem analysis and structure. In C. A. R. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction, Proceedings of NATO Summer School*, Marktoberdorf, pp. 3–20. IOS Press, Amsterdam, 2000.
23. M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley, Reading, MA, 2001.
24. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of the IFIP Congress*, pp. 321–332. North-Holland, 1983.
25. C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 1986.
26. C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.
27. T. P. Kelly and J. A. McDermid. A systematic approach to safety case maintenance. *Reliability Engineering & System Safety*, 71(3):271–284, 2001.
28. J. C. King. *A Program Verifier.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.
29. G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for Enhanced Languages and Methods to Aid Verification. Technical Report 06-21, Iowa State University, 2006.
30. N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.
31. Z. Li, J. G. Hall, and L. Rapanotti. Reasoning about decomposing and recomposing problem frame developments: A case study. In *1st IEEE International Workshop on Applications and Advances of Problem Frames*, 2004.
32. V. Loeb. *"Friendly Fire" Deaths Traced To Dead Battery: Taliban Targetted, But US Forces Killed.* Washington Post, p. 21, 24 March 2002.
33. D. MacKenzie. Computer-related accidental death: An empirical exploration. *Science and Public Policy*, 21(4):233–248, 1994.

34. K. Marzullo, F. B. Schneider, and N. Budhiraja. Derivation of sequential, real-time process-control programs. In A. M. van Tilborg and G. Koob, editors, *Foundations of Real-Time Computing: Formal Specifications and Methods*, pp. 39–54. Kluwer Academic Publishers, Boston, MA, 1991.
35. J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *Information Processing '62*, pp. 21–28. North-Holland, 1963.
36. D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.
37. G. Polya. *How to Solve It.* Princeton University Press, Princeton, NJ, 1957.
38. D. M. Severy, H. J. Brink, and J. Baird. Vehicle Design for Passenger Protection From High Speed Rear-End Collisions. Technical Report 689774, Society of Automobile Engineers, 1968.
39. B. C. Smith. The limits of correctness. In *Prepared for the Symposium on Unintentional Nuclear War, Fifth Congress of the International Physicians for the Prevention of Nuclear War*, Budapest, Hungary, June 28–July 1, 1985.
40. A. M. Turing. Checking a large routine. In *Report of a conference on High Speed Automatic Calculating Machines*, University Mathematical Laboratory, Cambridge, pp. 67–69, 1949.
41. A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE01, the Fifth IEEE International Symposium on Requirements Engineering*, pp. 249–263. IEEE CS Press, August 2001.
42. W. G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History.* The John Hopkins University Press, Baltimore, 1993.
43. J. C. P. Woodcock. Dependable systems evolution. In C. A. R. Hoare and R. Milner, editors, *Grand Challenges in Computing Research*, pp. 25–28. British Computer Society, Swindon, 2004.
44. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice-Hall, Englewood Cliffs, NJ, 1996.

3

The Abstract State Machines Method for High-Level System Design and Analysis

Egon Börger

Dipartimento di Informatica, Universita di Pisa, Pisa, Italy
boerger@di.unipi.it

Abstract We explain the main ingredients of the Abstract State Machines (ASM) method for high-level system design and analysis and survey some of its application highlights in industrial software-based system engineering. We illustrate the method by defining models for three simple control systems (sluice gate, traffic light, package router) and by characterising Event-B machines as a specific class of ASMs.We point to directions for future research and applications of the method in other areas than software engineering.

3.1 Introduction

In this paper, we give an answer to the often asked question *What characterises the Abstract State Machine (ASM) method among the practical and scientifically well-founded systems engineering methods?* The question is justified since the ASM method, which has been developed during the 1990s (see [24] for a historical account), is a latecomer among other well-known rigorous system design and analysis methods, including what misleadingly is called "formal" methods. For answering the question, we assume the reader to have some basic knowledge of what formal methods are and what they intend to achieve, but we sketch the major ingredients of the ASM method.[1]

We do not speak about special-purpose techniques, like static analysis, model checking etc., which draw their success from being tailored to particular types of problems. The discussion is focussed on *wide-spectrum methods*, which assist system engineers in every aspect of an effectively controllable

[1] However, this paper is neither an introduction to the ASM method nor a survey of its achievements. For the former see [29], for the latter [27], or Chaps. 2 and 9 of the AsmBook [46].

construction of reliable computer-based systems. In particular, such methods have to bridge the gap between the two ends of system development:

- The human understanding and formulation of real-world problems.
- The deployment of their algorithmic solutions by code-executing machines on changing platforms.

The activities these development disciplines have to support cover the wide range from requirements capture and analysis to writing executable code, including verification, validation (testing), documentation and maintenance (change management). As a consequence, a scientifically rigorous approach, which enhances best engineering practice by adding mathematical rigor to it, calls for a smooth integration into traditional hw/sw-system engineering procedures and notations of multiple ways to achieve various degrees of certifiable system trustworthiness and quality assurance.

Among such approaches, the ASM method is characterised by providing a *simple practical framework, where in a coherent and uniform way the system engineer can* adopt a divide-and-conquer approach, i.e.,:

- *Systematically separate multiple concerns, concepts and techniques*, which are inherent in the large variety of system development activities.
- *Freely choose for each task an appropriate combination of concepts and techniques* from the stock of engineering and mathematical methods, at the given level of abstraction and precision where the task occurs.

As will become clear in the following sections, not a single ingredient of the ASM method is original. What is unique is the simplicity of the method and the freedom it offers the practitioner to choose for each problem an appropriate combination of concepts, notations and techniques, which are integrated by the framework in a coherent way as elements of a uniform mathematical background. Among the examples we will discuss are the following:

- Abstract states, which can be richly structured, possibly unbounded or even infinite, as known from the theory of abstract data types and algebraic specifications [9, 10, 62, 80, 87], VDM [67], Z [104], COLD [66].
- Abstract instructions for changing states (high-level operational definition of state changes by guarded assignments), as familiar from pseudo-code notation, Virtual Machines (VMs)[2] and later RAISE [74].
- Synchronous parallel execution model, including conditional multiple assignments as present also in UNITY [88] and COLD [65].
- Locality principle as known from programming languages.
- Functional definitions, as in mathematics and functional programming [14].
- Declarative (axiomatic) definitions, as known from logic and declarative programming and specification languages.

[2] For example in Cremers' and Hibbard's *data spaces* [53], the operational transformations of abstract states are described by means of static functions, which form what is called there an *information structure*.

- Refinement concept, generalising the method which has been introduced by Wirth [113] and Dijkstra [59] and adapted to numerous formal specification methods [11, 12, 54, 91], including Z [58, 114] and B [1].
- Decomposition and hierarchy concepts, as familiar from automata theory, layered architectures and the problem frames approach [84].
- Function classification into monitored, controlled, shared etc., as known from programming and Parnas' Software Cost Reduction (SCR) method [82, 96].
- Mathematical verification of model properties by proofs at the required level of precision: sketched, detailed, machine assisted (interactive or fully automated).
- Experimental validation by simulation (model execution), e.g. for model checking invariants, run-time verification of properties, testing of runs (scenarios).

The *combined separation and integration capabilities of the ASM framework*, which allow the engineer to tailor his methods to the problem under investigation, are responsible for the successful applications of the ASM method in a variety of academic and industrial projects. They range from the design and analysis of programming languages, computer architectures, protocols and web services to the design, reengineering and validation of industrial control systems and the definition of industrial standards. Some examples are highlighted in Sect. 3.7.

Among the different development and analysis tasks, which can be coherently linked together in the ASM framework, we discuss the following ones:

- *Design.* The design activities split into three major groups:
  - *Ground model construction*, i.e., definition of a system blueprint that can be justified to correctly capture the requirements. This is supported by the ASM ground model technique explained in Sect. 3.4.
  - *Model refinement*, reflecting one by one the various design decisions, which lead from the ground model to code. A rigorously controllable discipline of stepwise adding implementation details is supported by the ASM refinement method explained in Sect. 3.5.
  - *Model change*, a combination of the ground model construction and refinement task, which uses the hierarchy of models constructed during the transformation of the ground model into code. When change requirements occur, this hierarchy is analysed to determine the level starting from where new refinements are needed to traceably incorporate the requested changes. Changing the models may trigger also new analysis tasks.
- *Analysis.* The goal of the analysis activities is to provide documentation and justification for the steps that lead from the requirements to the ground model and its implementation. This is required for two purposes: (a) an evaluation (read: explanation, verification, validation) of each

design decision by repeatable procedures and (b) change management and reuse of models. The analysis activities split into two major groups:

 – *Mathematical verification* of system properties by a variety of reasoning techniques applicable to system models at different levels of precision and under various assumptions, e.g.,
   · Outline of a proof idea or proof sketch.
   · Mathematical proof in the traditional meaning of the term.
   · Formalised proof within a particular logic calculus.
   · Computer-checked (automated or interactive) proof.
 – *Experimental validation* of system behaviour through simulation and testing of rigorous models at various levels of abstraction, like system test, module test, unit test, simulation of ground model scenarios, etc.

The core of the ASM method is based upon the following three concepts we are going to explain in the following sections, starting with an illustration by three simple examples in Sect. 3.2:

- *Notion of ASM*, a mathematically precise substitute for the intuitive notion of high-level algorithmic processes (including what software engineers call pseudo-code) and for the nowadays omnipresent concept of VMs. Technically, ASMs can be defined as a natural generalisation of Finite State Machines (FSMs) by extending FSM-states to Tarski structures. Tarski structures, also called first-order or simply mathematical structures, represent truly abstract data types. Therefore, extending the special domains of FSM-computations to these structures turns FSMs into ASMs, which work over possibly richly structured yet abstract states, as is explained in Sect. 3.3.
- *ASM ground models* as accurate high-level descriptions of given system requirements. They are expressed at a level of abstraction that is determined by the application domain and provide requirements documentation that is to be used as an authoritative reference for an objective evaluation of the requirements and the following further system development activities. The ground model must be kept synchronised with those activities, namely:[3]
  – *Detailed design.*
  – *Design evaluation* and *quality assurance* via analysis, including testing and an inspection and review process, focussed on certifying the

[3] The definition of the ground model may change during the design phase, namely if it is recognised during the implementation process that some important feature is missing in the ground model or has to be changed there. The *process* of building a ground model is iterative; it ends only with the completion of the design and may be re-opened during maintenance for change management. But at each moment of the development process, there is one ground model, documenting the current understanding of the problem the system has to solve.

consistency, correctness and *completeness* properties of the system that are needed to guarantee the desired degree of reliability.[4]
  – *System maintenance*, including requirements change management.

  In Sect. 3.4, we discuss the ASM ground model method further.
- *ASM refinements*, linking the more and more detailed descriptions at the successive stages of the system development cycle in an organic and effectively maintainable chain of rigorous and coherent system models. The refinement links serve the purpose to guarantee that the system properties of interest are preserved in going from the ground model via a series of design decisions to its implementation by the code – and to document this fact for possible reuse during maintenance and in particular for change management. We discuss the concept of ASM refinement further in Sect. 3.5.

The simple mathematical foundation of ASMs as FSMs working over arbitrary data types makes it easy for practitioners to understand and work with the concept. It also allows one to exploit for the ASM method the *uniform conceptual and methodological framework* of traditional mathematics, where one can consistently relate standard notions, techniques and notations to express any system features or views.[5] Having as background for the ASM method not just one *a priori* chosen formal language and associated proof calculus, but the full body of usual mathematical notations and techniques "supports a *rigorous integration of common design, analysis and documentation techniques* for model reuse (by instantiating or modifying the abstractions), validation (by simulation and high-level testing), verification (by human or machine-supported reasoning), implementation and maintenance (by structured documentation)" [46, p. 1]. We discuss this in Sect. 3.6 and illustrate it there by a characterisation of Event-B Machines as a family of specialised ASMs. In Sect. 3.7, we point to some application highlights of the ASM method.

3.2 Illustration by Examples

We illustrate here ASMs by three simple examples for (a) the *construction of ASM ground models*, which can be shown to capture the requirements in application problem terms and (b) their *refinements*, which can be proven to correctly reflect both (b1) the implementation details and (b2) the changes in the models when changes in the requirements come along. The examples are taken from [84], a book which explains very well the various descriptions

[4] Note that the evaluation of the design against the ground model also provides an objective, rational ground for settling disputes on the code after its completion.

[5] Thus the ASM method satisfies Parnas' request [95] to base the foundation for a reliable software engineering discipline on standard mathematics, avoiding the introduction of complicated specification languages and theories of language semantics.

one has to make and to fit together into a correctness argument, in order to show that under certain assumptions on the environment – typically reflecting the relevant domain knowledge – the behaviour of the specification satisfies the requirements. What we call a ground model is a *closed* model. It includes both the specification and the statement of the environmental assumptions and domain knowledge that are required in a correctness argument.

3.2.1 Sluice Gate Control

The following problem description is taken from [84, p. 49], the italics are ours:

> A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is required to control the sluice gate: the *requirement* is that the gate should be held in the fully open position for 10 min in every 3 h and otherwise kept in the fully closed position.
> The gate is opened and closed by rotating vertical screws. The screws are driven by a small *motor*, which can be controlled by clockwise, anticlockwise, on and off pulses.
> There are *sensors* at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut.
> The *connection to the computer* consists of four pulse lines for motor control and two status lines for the gate sensors.

Ground model. To simplify the correctness argument to be provided for the ground model, we stick to first modelling only the *user requirements* for the equipment, abstracting from the details about the screws, the motor, the sensors and the pulses. This reduces the system to an abstract device, which switches from a *fullyClosed* phase to a *fullyOpen* phase whenever the time *closedPeriod* has elapsed, and back when *openPeriod* has elapsed. To separate the issues related to (an implementation of) the timing model from the analysis of the user requirements, we use two so-called *monitored locations* (read: array or instance variables), $Passed(openPeriod)$ and $Passed(closedPeriod)$. Their truth values are assumed to be controlled correctly by the environment and to indicate when the intended time periods have passed, here $openPeriod = 10\,\text{min}$ for *fullyOpen* and $closedPeriod = 3\,\text{h} - 10\,\text{min}$ for *fullyClosed*. We interpret the term "for 10 min in every 3 h" as "at the end of the closure period" and being included in the total $period = 3\,\text{h}$. This leads to the model in Fig. 3.1, which is displayed in the usual FSM-style graphical notation using circles for phases (also called control states or internal states), rhombi for test predicates (also called guards) and rectangles for actions of submachines (for a definition of these control state ASMs see Sect. 3.3). Due to the abstraction from the motor and the sensors, the submachines to Open respectively Shut the gate do nothing and are included only

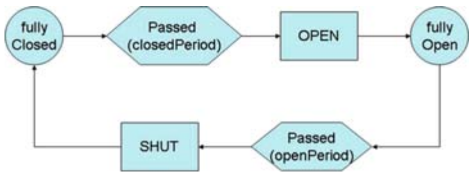


Fig. 3.1. SluiceGateGround model

to hold the place for the refinement by motor actions. Assuming appropriate conditions on initial states, this abstract machine SluiceGateGround can clearly be justified in terms of the gate being kept open (read: the device being in state *fullyOpen*) for *openPeriod* and closed (read: the device being in state *fullyClosed*) for *closedPeriod* to rigorously reflect the (apparently intended meaning of the) above stated requirement.

A refinement step. The first refinement step reflects the domain knowledge about a screw's driving *motor* and the sensors. We know that the motor can be set *on* and *off* and has two move *dir*ection values, *clockwise* (say to raise the gate) and *anticlockwise* (say to lower the gate). It is in these terms, namely the two so-called *controlled locations motor* and *dir*, which can be updated to any of their values in $\{on, off\}$, respectively $\{clockwise, anticlockwise\}$, that the submachines Open and Shut are refined by using three motor action submachines StartToRaise, StartToLower and StopMotor. The control of these actions uses the environmental gate status information that is obtained from the two sensors. The indication by the sensors that the gate travel has reached its top (fully open) and the respective bottom (fully closed) position is formalised by two monitored locations $Event(Top)$ and $Event(Bottom)$, respectively, taking boolean values. The time assumed for the execution of the new submachines is taken into account by refining the definition of *closedPeriod* and *openPeriod*. These two locations are examples of what we call *derived locations*, since their value is defined in a fixed manner (here by an equation) in terms of the values of other locations. This leads to the refinement SluiceGateMotorCtl of SluiceGateGround as defined in Fig. 3.2, together with the following definition of abstract motor actions:[6]

$$\begin{aligned}
\textsc{StartToRaise} &= dir := clockwise \qquad\qquad \textsc{StopMotor} = (motor := off)\\
&\ motor := on\\
\textsc{StartToLower} &= dir := anticlockwise\\
&\ motor := on
\end{aligned}$$

$$\begin{aligned}
&closedPeriod = period\\
&\quad -(StartToRaiseTime + OpeningTime + StopMotorTime)\\
&\quad -(StartToLowerTime + ClosingTime + StopMotorTime)
\end{aligned}$$

[6] In general, in an ASM all updates are executed in parallel, though in this example also a sequential reading will do.

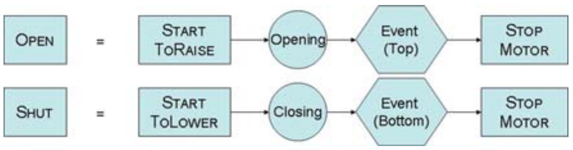


Fig. 3.2. SLUICEGATEMOTORCTL refinements of motor actions

The correctness proof for this refinement uses an *Input Locations Assumption*. It relates what happens at the environmental sensors to the model events:

- When the *top* respectively *bottom* of the gate travel is detected by the corresponding sensor, *Event*(*Top*) respectively *Event*(*Bottom*) becomes true in SLUICEGATEMOTORCTL.

Another refinement step. Here we introduce the four status lines connecting the controller and the physical equipment. The single SLUICEGATEMOTORCTL machine is replaced by a so-called multi-agent ASM SLUICEGATE. The latter consists of two abstract machines, an environmental machine PULSES describing the equipment actions when pulses appear and a software machine SLUICEGATECTL. In the definition of PULSES, we use the notation **upon** *Event* **do** *Action* for **if** *Event* **then** *Action*.

```
PULSES = upon Event(Clockwise) do dir := clockwise
         upon Event(AntiClockwise) do dir := anticlockwise
         upon Event(MotorOn) do motor := on
         upon Event(MotorOff) do motor := off
```

SLUICEGATECTL is the same as SLUICEGATEMOTORCTL except for a refined submachine STARTTORAISE. The refined submachine will EMIT(*Pulse*(*Clockwise*))
and EMIT(*Pulse*(*MotorOn*)). Similarly for START-TOLOWER, STOPMOTOR.

```
SLUICEGATECTL = SLUICEGATEMOTORCTL where
  STARTTORAISE = EMIT(Pulse(Clockwise))
                 EMIT(Pulse(MotorOn))
  STARTTOLOWER = EMIT(Pulse(AntiClockwise))
                 EMIT(Pulse(MotorOn))
  STOPMOTOR = EMIT(Pulse(MotorOff))
```

The correctness proof for this refinement step relates runs of the abstract and the refined machine. It relies upon the following assumptions:

- *Pulse Output Assumption*: each EMIT(*Pulse*(*p*)) yields *Event*(*p*) to happen in the environment.
- *Input Locations Assumption*: reinterpreted to reflect that the information detected by the sensors arrives at SLUICEGATECTL via two status lines as input *Event*(*Top*), *Event*(*Bottom*).

We also use the usual convention that events are consumed when they trigger a rule to be fired. Note that the refinement type is $(1, 2)$, meaning that every segment consisting of one step of SluiceGateMotorCtl is refined by a segment of two corresponding steps of SluiceGate, namely of one step of the software machine followed by one step of the environment machine.

Admittedly this is an elementary example. The very same technique has been applied in [41, 90] to successively refine an ASM ground model for a robot controller to a validated C^{++} control program.

3.2.2 One-Way Traffic Light Control

This example is about one-way traffic control:

> ...the traffic is controlled by a pair of simple portable traffic light units...one unit at each end of the one-way section...connect(ed)...to a small computer that controls the sequence of lights.
> Each unit has a Stop light and a Go light.
> The computer controls the lights by emitting RPulses and GPulses, to which the units respond by turning the light on and off.
> The regime for the lights repeats a fixed cycle of four phases. First, for 50 s, both units show Stop; then, for 120 s, one unit shows Stop and the other Go; then for 50 s both show Stop again; then for 120 s the unit that previously showed Go shows Stop, and the other shows Go. Then the cycle is repeated.

Ground model. From the user perspective, the problem is about two light units, each equipped with a $StopLight(i)$ and a $GoLight(i)$ $(i = 1, 2)$, which can be set *on* and *off*. In the ground model, we treat the latter as controlled locations to which a value *on* or *off* can be assigned directly, abstracting from the computer emitting pulses. We also abstract from an explicit time computation and treat the passing of time by monitored locations $Passed(timer(phase))$, where the function *timer* defines the requested light regime. The monitored locations are assumed to become true in the model when $timer(phase)$ has elapsed in the environment since the current *phase* was entered.

For definiteness, let us assume that the sequence of lights starts with both $StopLight(i) = on$ and both $GoLight(i) = off$. Let us call this phase $Stop1Stop2$. After $timer(Stop1Stop2)$ has passed, the control executes a submachine SwitchToGo2 and then enters phase $Go2Stop1$ (say), followed upon $Passed(timer(Go2Stop1))$ becoming true by a SwitchToStop2 to enter phase $Stop2Stop1$, then a SwitchToGo1 to enter phase $Go1Stop2$, finally a SwitchToStop1 to return to phase $Stop1Stop2$. This behaviour of the equipment is rigorously expressed by the sequence of four phase changing "ASM rules" in Fig. 3.3, defined again in FSM-like notation and using the submachine macros defined below.

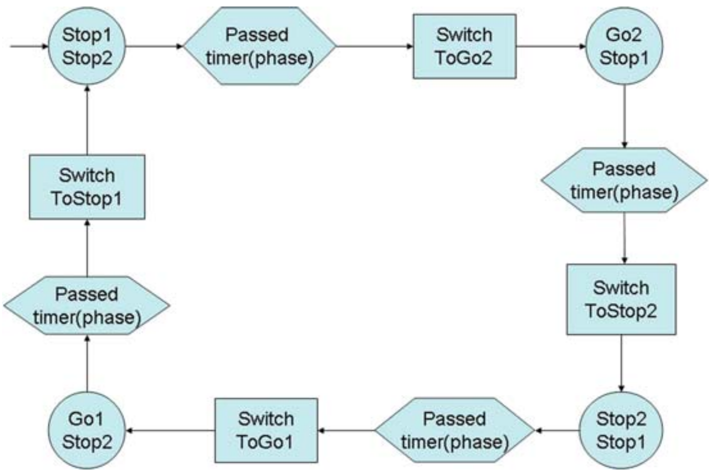


Fig. 3.3. 1WayTrafLightGround model

The four control states correspond to the required combinations of "showing" *Goi* and *Stopj*, reflecting that in the above requirements description, the values of $\mathit{StopLight}(i)$ and $\mathit{GoLight}(i)$ appear to be complementary:

$\mathit{Stop}\ i$ means $\mathit{StopLight}(i) = \mathit{on} \wedge \mathit{GoLight}(i) = \mathit{off}$
$\mathit{Go}\ i$ means $\mathit{GoLight}(i) = \mathit{on} \wedge \mathit{StopLight}(i) = \mathit{off}$

The complementarity of $\mathit{StopLight}(i), \mathit{GoLight}(i)$ values implies that switching them can be done in parallel. Thus, the two submachines SwitchToGo and SwitchToStop are copies of one machine (which only later will be refined to different instantiations introducing a sequentialisation, see below):

$$\textsc{SwitchToGo}i = \textsc{SwitchToStop}i = \begin{array}{l}\textsc{Switch}(\mathit{GoLight}(i))\\ \textsc{Switch}(\mathit{StopLight}(i))\end{array}$$

where $\textsc{Switch}(\mathit{Light}) = (\mathit{Light} := \mathit{Light}')$
($'$ for complement)

The light regime (50,120,50,120) associates to each *phase* its length (in terms of some time measurement), represented by function values $\mathit{timer}(\mathit{phase})$. Following the requirements, for this ground model, the function is assumed to be static (set before running the machine). A change request to include the possibility to configure the time intervals associated to the phases would make it dynamic and controlled by the configuration machine.

$$\mathit{timer}(\mathit{phase}) = \textbf{case}\ \mathit{phase}\ \textbf{of}\ \begin{array}{l}\mathit{Stop}1\mathit{Stop}2 : 50\,\mathrm{s}\\ \mathit{Go}2\mathit{Stop}1 : 120\,\mathrm{s}\\ \mathit{Stop}2\mathit{Stop}1 : 50\,\mathrm{s}\\ \mathit{Go}1\mathit{Stop}2 : 120\,\mathrm{s}\end{array}$$

With these definitions and assumptions, 1WayTrafLightGround can be justified in application domain terms to realise the desired cyclic light sequence.

A refinement step. We used for the ground model a mixed behavioural and declarative, instead of a purely declarative observational description to ease the correctness proofs for adding further details, which eventually should transform the abstract ground model into executable code. In a first refinement step, we introduce the software interface feature that relates R/G Pulses of the computer to turning the light units on/off. As in the sluice gate control example in Sect. 3.2.1, this refinement step replaces the single abstract machine 1WayTrafLightGround by a multi-agent ASM 1WayTrafLight consisting of an environmental pulse-triggered machine Pulses and a software machine 1WayTrafLightCtl. The latter is obtained from 1WayTrafLightGround by refining the submachines SwitchTo...i to emitting pulses:

```
1WayTrafLightCtl = 1WayTrafLightGround  where
  forall $i \in \{1,2\}$  SwitchTo...$i$ = Emit($RPulse(i)$)
                                          Emit($GPulse(i)$)
```

```
Pulses = forall $i \in \{1,2\}$
  upon $Event(RPulse(i))$ do  Switch($StopLight(i)$)
  upon $Event(GPulse(i))$ do  Switch($GoLight(i)$)
```

The link between 1WayTrafLight and 1WayTrafLightGround is provided by the following *Pulse Output Assumption*, which relates the software actions to what happens in the environment:

- Emit($RPulse(i)$) yields $Event(RPulse(i))$ to happen in the environment
- Emit($GPulse(i)$) yields $Event(GPulse(i))$ to happen in the environment

Using this assumption, it is easy to prove the refinement to be correct, observing that each software control step of the refined SwitchTo...i triggers an environment step of Pulses, which switches the corresponding lights. Thus, one ground model step is correctly refined to two steps in the refined multi-agent machine ($(1, 2)$-refinement).

Change requirement. We illustrate here two simple change requests, which lead to reusing the abstract machines defined above. The first one is:

> use simultaneous *Stop* and *Go* lights to indicate "Stop, but be prepared to Go" [84, p. 111]

To adapt the models to this request for change, it suffices to give up the view that the $StopLight(i)$ and $GoLight(i)$ values are complementary to each other and to *refine* the SwitchToGoi submachines by a sequentialised version. Everything else is kept unchanged in both the above ground model and its refinement. With this instantiation of SwitchToGoi, it should be clear how

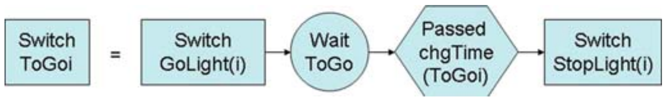


Fig. 3.4. Refining SwitchToGo by change time

to show that the new ground model correctly reflects the changed requirements and how to prove that it is correctly refined by the new refined model.[7]

$$\text{SwitchToGo}i_{grd} = \\ \quad GoLight(i) := GoLight(i)' \ \mathbf{seq}\ StopLight(i) := StopLight(i)'$$

$$\text{SwitchToGo}i_{ref} = \ \text{Emit}(GPulse(i))\ \mathbf{seq}\ \text{Emit}(RPulse(i))$$

Another typical change request could be to add a simultaneous *Stop* and *Go* lights period (change time) of say 10 s. This comes up to refine SwitchToGoi from an atomic to a durative action, leaving everything else unchanged. A natural way to do this is to introduce an intermediate control state *WaitToGo* between the executions of Switch(*GoLight*(i)) and Switch(*StopLight*(i)), as indicated in Fig. 3.4. The new function *chgTime* indicates the length of the *WaitToGo* phase, in the example 10 s.

Admittedly, these reuse examples are rather elementary, but the method is general. The reader who is interested in a more involved example may look at [38,69–71] for a reuse for C# and .NET CLR of the ASM models built and analysed in [107] for Java and the JVM.

3.2.3 Package Router Control

In this example, we illustrate the use of (a) the synchronous parallelism underlying the semantics of ASMs and (b) an abstract event handling scheme. The problem is about the control of a package router to sort packages into bins according to their destinations and appeared in [83]. The formulation below is copied from [84, p. 147].[8]

> The packages carry bar-coded labels. They move along a conveyor to a reading station where their package-ids and destinations are read. They then slide by gravity down pipes fitted with sensors at top and bottom. The pipes are connected by two-position switches that the computer can flip (when no package is present between the incoming and outgoing pipes). At the leaves of the tree of pipes are destination bins, corresponding to the bar-coded destinations.

[7] The definition of the **seq** constructor in the context of the parallel ASM execution model is defined in [45].

[8] For reasons of space we leave out the operator commands to stop or start the conveyor. Adding them would involve adding an operator machine.

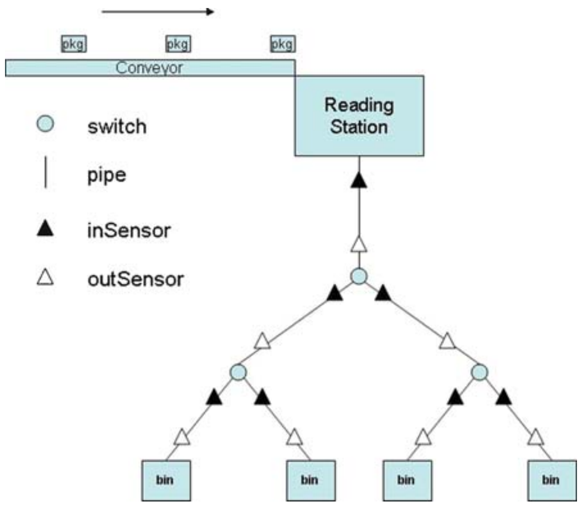


Fig. 3.5. Package router layout

> A package cannot overtake another either in a pipe or in a switch. Also, the pipes are bent near the sensors so that the sensors are guaranteed to detect each package separately. However, packages slide at unpredictable speeds, and may get too close together to allow a switch to be set correctly. A misrouted package may be routed to any bin, an appropriate message being displayed ...
> The problem is to build the controlling computer ... to route packages to their destination bins by setting the switches appropriately, and to report misrouted packages.

From the layout illustration in Fig. 3.5, one can recognise the elements of the problem signature: a static tree structure whose nodes are decorated with elements from various sets, namely *PkgLabel*, *Pipe*, *Switch*, *Bin*. Each of these domains is equipped with various functions. Assuming a unique association of bar-coded labels to packages, the reader decodes elements $l \in \mathit{PkgLabel}$ and stores the obtained information into locations $\mathit{pkgId}(l)$, $\mathit{dest}(l)$. To separate the decoding program from the control program, we consider these locations as static for the ground model.

Each $p \in \mathit{Pipe}$ comes with locations $\mathit{inSensor}(p)$ $\mathit{outSensor}(p)$ to signal package passing events. Elements $sw \in \mathit{Switch}$ come with a controlled location $\mathit{pos}(sw)$ indicating the current value of the switch position, *right* or *left*. To be able to correctly set $\mathit{pos}(sw)$ at runtime to reach a given bin b, a static function *dir* with values $\mathit{dir}(sw, b) \in \{\mathit{left}, \mathit{right}, \mathit{none}\}$ is required indicating whether there is a path from sw to b and in the positive case where to direct sw for taking this path.

Bin comes with a static function associating to every $d \in \mathit{Destination}$ the corresponding $\mathit{bin}(d)$ where packages for that destination are requested to be routed. It is required to define the derived predicate *MisroutedPkgInBin*, which is defined to be true for a pair (l, b) if and only if the package identified by $pkgId(l)$ arrived in bin b, but $bin(dest(l)) \neq b$.

The static tree structure is defined by the root *reader* and *succ*essor locations that satisfy the following conditions:

$$\begin{array}{l} succ(reader) \in Pipe \\ \textbf{forall } p \in Pipe : \; succ(p) \in Switch \cup Bin \\ \textbf{forall } sw \in Switch : \; succ(left, sw), succ(right, sw) \in Pipe \end{array}$$

Using these locations, we can define the following derived successor location for switches, which indicates its current, dynamically computed successor:

$$succ(sw) = succ(pos(sw), sw)$$

The information on no overtaking of packages can be formalised as first-in first-out behaviour of packages, using three types of queues:

- $Queue(reader)$ of labels of packages whose label has been read, but which did not yet enter the top pipe $succ(reader)$
- $Queue(pipe)$ of labels of packages that entered the pipe at $inSensor(pipe)$, but did not yet exit it at $outSensor(pipe)$
- $Queue(switch)$ of labels of packages that
  - Entered into *switch* at its entry point $outSensor(pipe)$, where *pipe* is the predecessor of *switch*
  - Did not yet exit at its end point $inSensor(succ(switch))$[9]

We define the control program ground model as a parallel composition of five submachines, each concerned with transferring a package (label) from one queue to the next one. The synchronous parallelism allows us to separate the routing functionality from the decision about the concrete scheduling mechanism required for an implementation.

```
PackageRouter = IntoReader
                FromReader2Pipe
                FromPipe2Switch
                FromSwitch2Pipe
                FromPipe2Bin
```

Each of these submachines is triggered by an event, namely a package arriving at a sensor. It is represented by a predicate of the corresponding sensor location. For a succinct notation, we adopt the convention that events are consumed by firing a rule guarded by this event, which saves us to repeat in

[9] This is well-defined since by assumption, *switch* can be flipped only when there is no package in it, i.e., $succ(switch)$ and thereby the exit point of *switch* can change only when $queue(switch)$ is empty.

each rule body the update through which the event predicate is reset to false. There are three types of events:

- $Event(ArrPkgLab(l))$ becomes true when the reader has decoded the barcode of a package into its associated label l.
- For any $sensor$, $EventOn(sensor)$ becomes true when the leading edge of a package arrives at $sensor$.
- For any $sensor$, $EventOff(sensor)$ becomes true when the trailing edge of a package leaves the $sensor$ behind.

The machine IntoReader simply enqueues a newly arrived package label:

```
IntoReader = if $Event(ArrPkgLab(l))$ then Enqueue($l, queue(reader)$)
```

Labels of packages arriving in a pipe, from the reader or from a switch, simply advance from the reader or switch queue to the queue of the pipe:

```
FromReader2Pipe =
  if $EventOn(inSensor(succ(reader)))$ then AdvanceFrom($reader$)
FromSwitch2Pipe
  forall $sw \in Switch$
    if $EventOn(inSensor(succ(sw)))$ then AdvanceFrom($sw$)
where
  AdvanceFrom($a$) = Dequeue($queue(a)$)
                     Enqueue($fstout(queue(a)), queue(succ(a))$)
```

Before advancing labels of packages that leave a pipe to enter a switch, one has to determine the correct position of that switch for that package to be routed correctly (if still possible). The correctness of the positioning submachine comes from the definition of *dir* and the assumption in the requirements that a switch can be flipped only when no package is present between its incoming and outgoing pipes.

```
FromPipe2Switch = forall $p \in Pipe$
  if $succ(p) \in Switch$ and $EventOff(outSensor(p))$ then
    Position($succ(p), fstout(queue(p))$) seq AdvanceFrom($p$)
      where Position($sw, e$) =
        if $Empty(queue(sw))$ and $dir(sw, bin(dest(e))) \neq none$
        then $pos(sw) := dir(sw, bin(dest(e)))$
```

Upon moving a package from a pipe into a bin, a submachine checks whether this package has been misrouted:

```
FromPipe2Bin = forall $p \in Pipe$
  if $succ(p) \in Bin$ and $EventOff(outSensor(p))$ then
    MoveToBin($p$)
    ReportMisrouting($fstout(queue(p)), succ(p)$)
where
  MoveToBin($p$) = Dequeue($queue(p)$)
                   Insert($fstout(queue(p)), succ(p)$)
```

```
ReportMisrouting$(l, b)$ =
    if $bin(dest(l)) \neq b$ then Display$(pkgId(l), b)$
```

The reader would have noticed that we left various submachines and auxiliary functions unspecified, for example, for queue operations or insertion of elements into bins, which we consider as generally understood or not critical for the investigated problem. The ASM method allows the specifier to build models with holes, that is, to leave parts of the specification either as completely abstract or as accompanied by informal explanations. For proving properties of such models, appropriate assumptions have to be made for these not furthermore specified parts and have to be proved once the holes are filled by definitions. This pragmatic separation of definition and proof concerns is an efficient technique to piecemeal solve complex problems, which is deeply rooted in the tradition of mathematics.

Refining Position. Assume that the control program cannot access the location $pos(sw)$ directly, but only send a pulse to Switch($pos(sw)$). As in Sects. 3.2.1 and 3.2.2, this can be reflected by letting the switching be done by an abstract machine Pulses triggered by a pulse, which is emitted by a refined ASM Position$_{ref}$. But one has to pay attention: if $pos(sw) := dir(sw, bin(dest(e)))$ is executed when $pos(sw) = dir(sw, bin(dest(e)))$ already holds, by the semantics of ASMs this update will leave $pos(sw)$ unchanged. But an execution of Switch($pos(sw)$) changes the value of $pos(sw)$ anyway, so that for a correct refinement its trigger must be restricted to the case that the direction required for the newly arrived package is different from the one required by the preceding package. Had we included this condition already into the guard of the abstract machine Position, the refinement here would have become a pure data refinement. As alternative one can include the additional guard into the refinement step:

```
Position$_{ref}(sw, e)$ =
    if $Empty(queue(sw))$ and $dir(sw, bin(dest(e))) \neq none$
        and $pos(sw) = dir(sw, bin(dest(e)))'$ then Emit$(Pulse(sw))$.
```

Verification. The requirement for the control program, namely "to route packages to their destination bins by setting the switches appropriately, and to report misrouted packages", does not state anything about the relation between correct routing and misrouting and when misrouting is to be expected. Given that

- Reader and sensors are guaranteed to detect packages separately.
- Packages cannot overtake in either conveyor or pipes or switches.

the following can be proved for (an appropriately initialised) PackageRouter:

Lemma 1. *Every package from the conveyor belt eventually arrives at some bin b (if* PackageRouter *is not stopped before). If the package never meets and succeeds another package with a different destination bin in a* *switch, then b is its associated destination bin. If the package is misrouted,* $\textsc{Display}(\mathit{pkgId}(l), b)$ *will be activated, where l is the label associated to the package and sent by the reader to* $\textsc{PackageRouter}$.

This can be proved by an induction on the $\mathit{level}(sw)$ of the switches the package goes through in the router tree. For the induction step, a stronger hypothesis is required for packages that never meet and succeed another package with a different destination bin in a switch, so that for the proof the correctness statement has to be strengthened to guarantee also the following property:

- Assume a package labeled l, which never meets and succeeds another package with a different destination bin in a switch, enters a switch sw in state S. Then in this state $\mathit{dir}(sw, \mathit{bin}(\mathit{dest}(l))) \neq \mathit{none}$ is true. Additionally in state S, either $\mathit{pos}(sw) = \mathit{dir}(sw, \mathit{bin}(\mathit{dest}(l)))$ holds or the guard of $\textsc{Position}_{ref}$ is true; in the second case, $\mathit{pos}(sw)$ is correctly set to $\mathit{dir}(sw, \mathit{bin}(\mathit{dest}(l)))$.

3.3 Enriching FSMs to ASMs

Following [23, 31] we start by analysing FSMs, which we consider as the archetype of ASMs.[10] In fact, from the practitioner's point of view, it seems obvious that to accurately characterise VMs, it suffices to extend FSM instructions from symbol-reading/writing to reading and updating of arbitrarily structured abstract data.

3.3.1 Generalising FSM States

The well-known interpretation of FSM instructions

in state i reading input a, go to state $\delta(i, a)$ and print output $\lambda(i, a)$

of Mealy automata can be formalised as follows by simultaneous updates of a control state location (read: a variable) *ctl_state* and an output location *out* when an input event is present (read: when the input location *in* is defined).[11]

```
$\textsc{MealyFsm}(in, out, \delta, \lambda) =$ **if** $\mathit{Defined}(in)$ **then**
  $\mathit{ctl\_state} := \delta(\mathit{ctl\_state}, in)$
  $\mathit{out} := \lambda(\mathit{ctl\_state}, in)$
```

[10] The original definition of ASMs in [79] was motivated by a different goal: an epistemological desire to generalise Turing's thesis.

[11] We consider here deterministic FSMs, since non-deterministic FSMs can be modelled using the ASM **choose** construct described below. For the sake of simplicity of exposition, we suppose each input to be consumed by executing a transition, technically speaking *in* to be a monitored function as explained below.

Two restrictions one sees here are characteristic for the FSM computation model, besides the strict separation of input and output:

- *Only three locations* are read resp. updated (per step): *in*, *ctl_state*, *out*
- *Only three special data types* are used: finite sets of (a) symbols representing input/output (letters of an alphabet) and (b) abstract control states representing a bounded memory (typically written as labels or integers)

ASMs result from withdrawing these restrictions and permitting a machine in each step to read and update *arbitrarily many, possibly parameterised, locations whose values can be of arbitrary type.* Consequently, an arbitrary condition may be used as a rule guard.

Stated differently, the *notion of state* is generalised from the three FSM-locations holding FSM-specific values to an arbitrary set of updatable locations where values of whatever type reside, whether atomic or structured: application objects, mathematical entities, etc. This flat view of state (read: abstract machine memory) lends itself to standard methods for grouping of data into a modular memory structure. A common method is to group subsets of data into tables, via an association of a value to each table entry $(l, (a_1, \ldots, a_n))$, also called *location* (think of it as an array variable). Here l plays the role of the name of the table, the sequence $(a_1, \ldots, a_n)$ the role of an entry and $l(a_1, \ldots, a_n)$ denotes the value currently contained in the table entry $(l, (a_1, \ldots, a_n))$. In logic, such a table is called the interpretation of a function or a predicate. The common mathematical notion of structure, as explicitly defined by Tarski [109] and since then in the centre of the model theory branch of mathematical logic, is defined as a set of tables. It represents the most general notion of structure that has come up in occidental science and thus is what we need for a sufficiently general notion of VM or ASM state.[12] Via the view of a Tarski structure as given by domains of objects coming with predicates (attributes) and functions defined on them, there is also a close relation to the object-oriented understanding of classes and their instances.

Having as state a set of parameterised locations, it comes natural for state changes that in one step an ASM can update simultaneously the value not only of two or three, but of arbitrarily many locations. This generalises the above FSM-transition rules to *guarded update rules* (called ASM rules) of the following form:

if *Condition* **then** *Updates*

where *Updates* is a finite set of assignments of the form $f(t_1, \ldots, t_n) := t$.

Such a view is also taken in [3, p. 52] "to completely separate, during the design, ... individual assignments from their scheduling". Sets of guarded update rules as above in fact constitute a normal form for a class of ASMs,

[12] This view of Tarski structures supports a generalisation of Parnas' table technique [95, 96] as a convenient notation for ASMs, as detailed in [22, 23].

which suffice to define every Event-B model (see the Event-B-model normal form ASMs defined in Sect. 3.6.1) and to compute every synchronous UML activity diagram [33].

A basic ASM is, therefore, defined by a finite set of ASM rules, which play the role the instructions play for an FSM. They constitute a mathematical substitute for the intuitive concept of UML activity diagram transitions **upon** *Event* **do** *Action*, defining actions as value changes of some locations.

The concept of computation (run) of an ASM is the same as for FSMs, except that the possibility of having multiple locations updated in one step is further enhanced by the following stipulation: an ASM, instead of executing per step one rule as do FSMs, fires in each step all its rules whose guard is true in the given state.[13] This synchronous parallelism in an ASM step helps the designer to make the independence of multiple actions explicit and to avoid introducing irrelevant sequentialisations of orthogonal features. For *asynchronous multi-agent ASMs*, it suffices to generalise the notion of run from sequences of moves (execution of rules) of just one basic ASM to *partial orders* of moves of multiple agents, each executing a basic ASM, subject to a natural *coherence condition*, see [46, Definition 6.1.1].

The general form of ASM rules no longer shows the particular structure determined by the control states $i, j, \ldots$ of an FSM, which, however, is particularly useful for modelling control systems, protocols, business processes and the like.[14] We therefore proposed in [23] the name *control state ASMs* for ASMs which keep at their top level the characteristic FSM control states, as a means to model some overall status or mode guiding the execution of guarded synchronous parallel updates of the underlying rich state. Formally, control state ASMs are ASMs where all the rules have the form $\text{Fsm}(i, \textbf{if } \textit{cond} \textbf{ then } \textit{rule}, j)$, standing for the following ASM rule (where *rule* is supposed to also be an ASM rule):

```
if $ctl\_state = i$ and $cond$ then
    $rule$
    $ctl\_state := j$
```

To display such rules often the standard graphical notation for FSMs is used, where circles represent the control states, rhombi the guard *cond*ition and rectangles the *rule* body. See Figs. 3.1–3.4.

For pragmatic reasons – ease of modelling real-life VMs – we are going to indicate in the next subsection two further constructs to form basic ASM rules, one which makes it possible to explicitly name forms of non-determinism and

[13] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *Condition* is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates. This yields the successor state S' of S.

[14] See the use of modes in [95] as a means to structure the set of states.

one which enhances the parallelism of finitely many simultaneous updates. Similarly, we freely use other standard notations, where a rigorous definition of their meaning can be given.

3.3.2 Classification of Locations, Non-Determinism, Parallelism

The roles played by the locations and functions appearing in an FSM determine the ASM classification of locations and functions. Some locations or functions are *static*, meaning that their values do not depend on the (dynamics of) states, e.g. the two FSM-functions δ, λ that are defined by the FSM program. Static ASM locations can be given purely functional or axiomatic definitions, as done for the locations $timer(phase)$ in Sect. 3.2.2. Thus, ASMs provide a framework for a theoretically well-founded, coherent and uniform *practical combination of abstract operational descriptions with functional and axiomatic definitions.*[15]

Locations whose values may depend not only on the values of their parameters, but also on the states where they are evaluated, are called *dynamic*. Examples are the FSM-locations *in*, *ctl_state*, *out*. These locations can have four different roles:[16]

- *in* is only read by the FSM and updated only by the environment. Such locations of an ASM M, which are only readable by the machine and writable only by other machines or the environment, are termed *monitored* for M. It is often convenient to describe their meaning for M axiomatically, by a list of assumptions, thereby relegating the proof for these assumptions to the model of the other machine(s) where the computation of the monitored locations takes place (*divide-and-conquer* technique).
- *out* is only written by the FSM and read only by the environment. Such locations of an ASM M, which are only writable by M and readable only by other machines or the environment, are termed *output* locations of M.
- *ctl_state* is read and updated by an FSM. ASM locations that are readable and writable only by M are called *controlled* locations of M.
- ASM locations that are readable and writable by M and some other machine or the environment are called *shared*. Typically protocols are used to guarantee the consistency of updates of such locations.

This classification distinguishes between the roles different machines (e.g. the system and its environment) play in using dynamic locations for providing or updating their values. Monitored and shared locations represent two general mechanisms to specify *communication* types between different ASMs. For *modularisation* purposes, we also distinguish between basic and derived

[15] This avoids the alleged, though unjustified and in fact destructive, dichotomy between declarative and operational design elements, which unfortunately has been strongly advocated in the literature over the last 30 years. See the discussion at the end of Sect. 3.6.2.

[16] The naming is influenced by its pendant in Parnas' Four-Variable-Model [95].

ASM locations. Derived locations are those whose definition in terms of basic locations is fixed and may be given separately, e.g. in some other part ("module" or "class") of the machine or by axioms, equations, etc. For an example, see the function $succ(sw)$ in Sect. 3.2.3.

A function or predicate f is called of a type if every location $f(x)$ is of that type. Selection functions constitute a particularly important class of monitored functions, for which also the following special notation is provided to make the inherent non-determinism explicit:

choose x **with** ϕ **in** *rule*

standing for the rule to execute *rule* for one element x, which is arbitrarily chosen among those satisfying the selection criterion ϕ. Similarly, also the synchronous parallelism, which is already present in the execution of ASM rules, is extended by the following standard notation:

forall x **with** ϕ **do** *rule*

standing for the simultaneous execution of *rule* for every element x satisfying the property ϕ.

3.4 ASM Ground Model Technique

The concept of ASM ground model goes back to [16–21], where it has been used to produce a faithful ASM model for the, at the time to-be-defined, ISO standard of Prolog [36, 43]. We describe here the foundational and technical characteristics of ASM ground models and refer the reader to [25, 30] for a more detailed dicussion of the concept.

The goal of building a ground model is to turn given informal requirements into a clear, unambiguous, accurate, complete and authoritative reference document for their intended content. This document is to be used for the evaluation, the implementation and possible changes of the requirements. Ground models have to be formulated and analysed at the level of abstraction of the given application domain, prior to coding in any programming language. In other words, ground models are specifications that precisely and authoritatively define, in rigorous application domain terms and at the level of detailing that is determined by the application, what the to-be-constructed software-controlled system is supposed to do. Ground models constitute a "blueprint" of the to be implemented piece of "real world". In the semiconductor industry, they are named "golden models" [103, p. 26]. They represent what Brooks [49] calls "the conceptual construct" or the "essence" of a software system, whose definition precedes the development of its machine-managed representation by code. It must be possible to justify such a definition as

- *Consistent* internally
- *Correct* and *complete* with respect to the intuitions underlying the informal requirements

These three properties characterise specifications we call ground models. To establish these properties, every available scientific or engineering technique must be usable. This includes inspection and review of the ground models to get the correctness and completeness properties checked by application domain experts (or potential users) and system designers. They must be enabled to use a combination of tool-supported simulation techniques – for systematic experiments (model checking and testing) with the models – and of mathematical verification techniques – to show the model to possess the properties of interest, e.g. as part of a certification procedure.

In [30] we explain the meaning of these three basic properties in more detail and show that to establish them, a ground model language is needed that satisfies the following two properties:

- The language is understood by all the parties involved. It mediates between the application world, where user and domain experts live, and the world of mathematical models and software-intensive systems, where software architects, programmers, testers, reviewers and maintenance experts live.
- The language provides a means for a combined use of both model validation by simulation (testing or model checking) and property verification by mathematical proof.

The language of ASMs satisfies these conditions, clearly separating models and their properties. It also permits to construct ground models with the following three constituent attributes:

- *Precision* to satisfy the required accuracy exactly.
- *Minimality*, abstracting from details that belong only to the further design and not to the application problem.[17]
- *Simplicity* to be understandable, rigorously analysable and acceptable as contract by domain experts, system architects, reviewers and testers.[18]

Thus, ground models share all the properties Parnas advocates convincingly for the software documentation provided by a good engineering discipline [95]. Obviously most of these properties – precision, accuracy, consistency, correctness, completeness, authoritative character – must be preserved for the further documentation produced on the way from ground models to code. This documentation of the detailed design process is provided by the ASM refinement method we are going to characterise in the next section.

[17] The minimality avoids to define ground models that restrict the problem solution unnecessarily. Thus, it helps to leave the design space open as much as possible.

[18] Experience in the following domains has confirmed that the ASM language is understandable for domain experts without computer science education: railway, control and telephony systems [32,42,50], business and aviation security processes [7,13,35,77,78], linguistics [55,85,92,93], biology [89], social sciences [47,48].

One final word on the often heard claim that such a ground model construction and analysis effort is an add-on one should avoid in an efficient software engineering process. This claim reflects only the fact that numerous current system development approaches consider the code as the true definition of the system, excluding any other authoritative description of the system-to-be-built before it has been encoded. Among the typical, rather expensive effects of this view, one finds the following: (a) the final system may not really do what it was required by the customer to do, (b) the final system is not well-understood (first of all not by the application domain experts who have to work with it, but often also not by the software specialists who face serious difficulties in analysing, understanding and repairing unexpected system breakdowns), (c) the testing effort becomes overwhelming (without the possibility of guaranteeing a certifiable standard of reliability), (d) system changes can become rather difficult to program and hard to control (e.g. with respect to their compatibility with some previously guaranteed behaviour), (e) maintenance becomes a nightmare once those who have led the development are not available any more, etc. Careful construction and analysis of ground models, combined with their stepwise detailing (refinement) to code, is a means to prevent such undesired effects of missing conceptual application-centric control over the system. Thus, the effort spent on ground models is highly compensated by what is saved in later development stages like testing, inspection and maintenance. See also the remark at the end of Sect. 3.5 and [30] for further discussion.

3.5 ASM Refinement Concept for Detailed Design

Parnas [95] explains why good software documentation must record the key design decisions in a transparent and easily accessible way. Typically one has to take numerous and often orthogonal design decisions when implementing a ground model by code. Refined models document those design decisions that do not belong to the application problem and therefore, by the minimality property, do not appear in the ground model but pertain to the implementation of the algorithmic problem solution.[19] In fact some authors distinguish

[19] As already observed in Sect. 3.1 for ground models, this distinction does not pertain to the process of model building. The classification of what belongs to the "essence" of the system and what only to its implementation may change during the design process, typically when "implementation decisions are made before specification is complete and the decisions can have a major effect on the further specification of the system" [108, p. 440]. The final definition, yielding a document with the hierarchy of stepwise refined models, can be given only at the end of the design process. This document too is typically re-opened during maintenance for change management, where it has to be synchronised with the decisions taken for the system change.

between requirements specifications, documented by ground models, and design (also called technical) specifications with the frequent “explosion of ‘derived requirements’ (the requirements for a particular design solution), caused by the complexity of the solution process” [75, Fact 26].

We have generalised the classical refinement method [59,113] to the mathematically precise notion of structure-transforming pseudo-code defined by ASMs (see [26] for a recent survey with further details). It allows the designer to fine-tune any given abstract construct, which can be viewed as “the already-fixed portion of a multi-step system development”, to the new details required to realise a design decision by an implementation, “the yet-to-be-done portion of a multi-step system development” [108, p. 438]. In this way the ASM refinement method directly supports the practitioner’s view of an implementation as “a multi-step process. Each stage of this process is a specification for what follows” [108, p. 440]. For this reason and differently from other refinement notions in the literature, an ASM refinement step can simultaneously involve both the signature (the data structure) and the control structure (the flow of step-by-step operations). In fact, in choosing how to refine an ASM M to an ASM M^*, one has the freedom to define the following five concepts (see Fig. 3.6):

- A notion (signature and intended meaning) of *refined state.*
- A notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest. Since certain intermediate M-states or M^*-states may be irrelevant (not of interest) for the refinement relation, they are hidden by defining what are the corresponding states of interest, namely those pairs of states in the runs one wants to relate through the refinement. Usually initial states in M and M^* correspond to each other, similarly for pairs of final states (if there are any).

$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \cdots \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \cdots \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

With an equivalence notion $\equiv$ between data in locations of interest in corresponding states.

Fig. 3.6. The ASM refinement scheme

- A notion of abstract *computation segments* $\tau_1, \dots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \dots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m, n)-diagrams and the refinements (m, n)-refinements).
- A notion of *locations of interest* and of *corresponding locations*, i.e., pairs of (possibly sets of) locations one wants to relate in corresponding states.
- A notion of *equivalence* $\equiv$ of the data in the locations of interest; these local data equivalences usually accumulate to a notion of equivalence of corresponding states of interest.

Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M if and only if every (infinite) refined run simulates an (infinite) abstract run with equivalent corresponding states. This ASM refinement concept generalises other more restricted refinement notions in the literature, as analysed in [97, 98], and scales to the controlled and well documented stepwise development of large systems.

In particular, the ASM refinement method supports modular system descriptions, including the modularisation of ASM refinement correctness proofs aimed at mechanisable proof support, for examples see [37,44,97,107].

If the authoritative character of both ground and intermediate models is taken seriously, then refined models support change management, namely by documenting natural points for possible design changes, e.g. to cover new cases or to optimise a solution. Obviously this implies that to maintain the series of models as documentation of a changed system, the models affected by the change have to be updated and to be kept synchronised with the ground model or other intermediate models, as already pointed out in the preceeding footnote. If this implies additional modelling work, it saves work for the coding during maintenance steps. The report [42] on the successful use of the ASM method for an industrial reengineering project shows that in certain situations, using a compiler from the underlying class of ASMs to executable code can even make re-coding completely superfluous and keep maintenance at the application-centric modelling level, where the changed requirements are formulated. More generally the report illustrates that using the ASM method produces no useless overhead: it typically shifts the most error-prone part of the development effort from the late coding and testing phases to the early modelling phase. This phase is dictated by high-level architectural concerns, major design decisions, the need to achieve a correct overall system understanding by experts and to lay the ground for a certification of the required trustworthiness. But the amount of the overall system development effort is essentially kept unchanged. See also the remark at the end of Sect. 3.4.

Good intermediate models, refining an otherwise unaltered model to capture a to-be-realised design feature, provide descriptions which are easier to understand than the code. This is helpful in particular when the maintenance team has to take its information on the system behaviour from the documentation, after the development team has left. The examples in the literature (e.g. [37,42,44,107]) and my own industrial experience show that, contrary to Parnas' claim [95] on the classical refinement method, good ASM refinements do not lead to long or repetitive programs, but allow the designer to succinctly document design decisions in a focussed and piecemeal manner, one by one, avoiding any irrelevant detail or repetition.

3.6 Integration of Multiple Design and Analysis Methods

Due to the mathematical character and the simplicity of the conceptual framework of the ASM method, the method can be combined in a semantically coherent and natural way with any other accurate (scientifically well-defined) design and analysis technique. We illustrate this in Sect. 3.6.1 by a characterisation of Event-B models as a particular class of ASMs, coming with a specialised refinement definition. In Sect. 3.6.2, we highlight the embedding of further (including so-called semi-formal) approaches to system design and analysis into the ASM method.

3.6.1 ASM-Characterisation of Event-B Machines

Event-B models are presented in [2,3,6] as an extension of B-machines [1], which have been used with success for the development of reliable industrial control software in a variety of safety-relevant large-scale industrial applications [5]. A discussion of the foundational relations between B-machines and ASMs can be found in [23, Sect. 3.2]. We limit ourselves here to characterise Event-B models as a class of ASMs, following [31, Sect. 4]. The three basic constituents to analyse are the notions of state, event and refinement.

States of Event-B models can be viewed as Tarski structures with a static and a dynamic part. The static part, called *context*, consists of three items:

- Sets s which represent the universes of discourse (domains)
- Constants c which are supposed to have a fixed interpretation
- Properties (c, s) used as axioms characterising the intended model class

The dynamic state part consists of variables v and an environment, which is viewed as another Event-B model. Inputting is captured by non-determinacy. The states are supposed to be initialised technically via a special event with true guard.

Events come in the following form:

if *guard* **then** *action*

where the *guard* is a closed first-order set theory formula with equality and *action* has one of the following three forms (we use the ASM notation which is slightly different from Abrial's notation):

- Updates. The syntactical Event-B-model reading of Updates is that of a simultaneous substitution $v_1, \ldots, v_n := e_1, \ldots, e_n(v)$ of v_i by e_i, whereas its equivalent semantic ASM reading is that of a finite set of simultaneous updates $v_i := e_i$ of the values of variables (parameter-free locations) by expression values
- **skip**
- **choose** x **with** $P(x, v)$ **in** *Updates* where *Updates* is a simultaneous substitution $v_1, \ldots, v_n := e_1, \ldots, e_n(x, v)$

Abrial views the operational interpretation of events, for example, as defined above by the semantics of ASM rules, only as an informal intuitive account, whereas the official semantic definition comes in the form of logical descriptions using pre/post conditions and invariants. For the present comparison with ASMs it suffices to consider invariants as descriptions of properties the designer wants and claims to hold in every state that is reachable from an initial state. Of course, eventually such invariants will have to be justified, by whatever available means.

The preceding definition of events represents an Event-B normal form for ASM rules. The outstanding characteristic feature, namely the underlying interleaving semantics ("at each moment only one event can occur"), is reflected by a top-level choice among the finitely many events which may be applicable. For this non-deterministic choice among ASM rules $R(i)$ we use the following notation:

$$R(0) \textbf{ or } \ldots \textbf{ or } R(n-1) = \textbf{choose } i \textbf{ with } i < n \textbf{ in } R(i)$$

There is a technical consequence of this interleaving interpretation of which the designer should be aware. The form in which simultaneous updates are collected under a guard into one event has an impact on the semantics of the model, differently from the basic parallelism of ASMs where in each state every applicable rule (read: event) is applied. The splitting of updates into different events implies some non-deterministic scheduling of events with overlapping guards. This is reflected by the following Event-B normal form (the part in brackets [ ] is optional):

$$\begin{array}{l} \mathit{Rule}_1 \textbf{ or } \ldots \textbf{ or } \mathit{Rule}_n \textbf{ where } \textbf{ forall } 1 \le i \le n \\ \quad \mathit{Rule}_i = \textbf{if } \mathit{cond}_i \textbf{ then } \ [\textbf{choose } x \textbf{ with } P_i(x) \textbf{ in}] \ \ \mathit{Updates}_i \end{array}$$

In this normal form, the limit cases are that cond_i is always true (unconditional updates) or that the set $\mathit{Updates}_i$ is empty (**skip**). We have disregarded the constraint imposed in [4] that in an Event-B model, no parallel update is allowed for the same variable. This constraint is only of technical nature and prevents the case with inconsistent update sets to happen, whereas the semantics of ASMs prescribes in this case only that the computation is aborted.

In comparison to the general form of ASM rules, there are no rules of the form **forall** x **with** $P(x)$ **do** *Rule* and the only external **choose** (i.e., one that is not applied directly to a set of updates) that is permitted is the top-level one on rules defining the interleaving model so that there is no further nesting of choices. This prevents the designer from using complex quantifier-change structures in his models, so that they have to be circumscribed by different means in case they are part of a "natural" description of intended system behaviour.

Refinement of Event-B models plays a crucial role for the B-method, as it does for the ASM method. However, the goal to provide definitions of Event-B models together with mechanically checked (interactive or automated) proofs of the desired invariants dictates a restriction to certain forms of the general ASM refinement concept. Using the analysis of the ASM refinement concept in [97,98], one can say that for Event-B model refinements, only $(1, n)$-refinements with $n > 0$ are permitted. No $(1, 0)$-refinement is allowed, reflecting the condition that each abstract event must be refined by at least one refined event, and no (n, m)-refinement with $n > 1$ is allowed. In addition, Event-B model refinements must satisfy the following constraints:

- In a $(1, n)$-refinement $F_1, \ldots, F_n, F$ of E, each F_i is supposed to be a new event refining **skip**.
- The new events F_i do not diverge.
- If the refinement deadlocks, then the abstraction deadlocks.

As to the *observables* in terms of which Event-B refinements are formally defined, they correspond to what we have called the *locations of interest* of an ASM [26]. Formally they can be viewed as projections of state variables. Technically speaking, in Event-B refinements the observables are variables which are required to satisfy the following conditions:

- They are fresh variables with respect to state variables and to invariants.
- They are modifiable only by observer events of form $a := A(v)$.
- They depend only on state variables v.
- The abstract observables $A(v)$ can be "reconstructed" from the refined ones by an equation $A(v) = L(B(w))$, which represents an "invariant gluing the abstract observables to the refined ones".

The only pragmatically relevant one of these technical conditions on observables is the gluing invariant. In ASM refinements, any mathematically accurate scheme to relate refined and abstract observables is allowed, and it need not be describable equationally.

3.6.2 Integrating Special Purpose Methods

In a similar way to Event-B models and to UML activity diagrams and Parnas' table technique, mentioned in Sect. 3.3.1, all the major computation and system design models turned out to be natural instances of specific classes of

ASMs (for details see [23, 28]). This confirms the unifying-framework character of the ASM approach to systems engineering. Many specification approaches are geared for some particular type of application and equipped with specially tailored definition or proof techniques. Such features can be naturally reflected by imposing appropriate restrictions on the class of ASMs, the refinement scheme and the related proof methods.

For the general, but loose UML-based approach to system engineering, the ASM method offers a rigorous, semantically well-defined version. The ASM ground model and refinement concepts replace the loose character of human-centric UML models and of the links the UML framework offers between descriptions at different system design levels. Starting from the accurate ASM-based semantic definition of the various UML diagrams and related notations (see [33, 34, 51, 52]), this equips UML-based practice with the degree of mathematical precision that distinguishes a scientifically rooted engineering discipline worth its name.

Another way to seamlessly include so-called semi-formal design techniques into an ASM-based development process is discussed below. The less rigorous a specification is, e.g. when there are reasons to momentarily leave parts of the specification as only informally explained, the more "holes" the ASM model shows that one has to fill by providing assumptions on the intended meaning. These assumptions have to be discharged for the refined models, where the detailed design introduces the missing elements. Within the framework of mathematics, much of which is what in computer science is called semi-formal, this is a legitimate way to proceed. A similar freedom of formality concerns the notations. As is characteristic for mathematical disciplines, the ASM method is not bound by the straitjacket of a particular formal language, but allows one to freely use any standard algorithmic and mathematical notation. The only condition for adopting any useful description technique, whether textual or tabular or graphical or whatever, is a mathematically rigorous definition of its meaning.

The ASM method also incorporates within one common modelling framework two pairs of approaches that in the literature are frequently, but erroneously, viewed as antagonistic instead of complementary, namely using so-called declarative (denotational) vs. operational and state-based vs. event-based system descriptions. For the definition of an ASM, one can use as much of declarative or denotational characterisations as desired, using functional definitions or logico-axiomatic descriptions. But this does not exclude the use of abstract operational descriptions where the latter turn out to be simpler, more intuitive and easier to refine to code. Declarative definitions are often used to define the background signature of an ASM (via static, monitored, derived functions, see Sect. 3.3.2). It is also often used to define what among the class of all possible runs of an ASM is considered as a legal run, describing axiomatically a certain number of features one wants to abstract away at the investigated level of abstraction. Similarly, whatever one wants to classify as an event can be included into the declaration of the state signature and be treated correspondingly in rule guards; see for example Event-B, where events are considered as rule firings, or [15] where process-algebraic event-based structuring techniques and concurrency patterns are combined with the state-based abstraction mechanism and synchronous parallelism of ASMs.

In a similar way, the general mathematical framework of the ASM method allows one the coherent separation and integration of defining a model and proving model properties. The ASM method does not force you to *simultaneously* define your models and prove properties for them, still less to do this in an *a priori* determined deductive system, but it allows you to add proofs to your definitions, where appropriate, and to do this in various ways, depending on what is required. Obviously a price has to be paid for this generality: if one wants a machine-assisted mechanical verification of your system, one will have to formalise it in the language of the theorem prover. In this case, it will be an advantage if one succeeds to define a model right from the beginning as a set of particular logical or set-theoretical formulae, as is the case for example in the B method [1]. On the other side, since ASMs are not formulae but represent executable models, the ASM method allows one to adopt for abstract models simulation, run-time verification and testing techniques, where proofs for whatever reason are not viable.

3.7 ASM Method Applications in a Nutshell

The proposal to use ASMs (a) as a precise mathematical form of ground models and (b) for a generalisation of Wirth's and Dijkstra's classical refinement method [59,113] to a practical systems engineering framework supporting a systematic separation, structuring and documentation of orthogonal design decisions goes back to [16, 17, 21]. It was used there to define what became the ISO standard of Prolog [36]. Since then, numerous case studies provided ground models for various industrial standards, e.g. for the standard of BPEL4WS [64], for the ITU-T standard for SDL-2000 [76], for the de facto standard for Java and the Java Virtual Machine [107], the ECMA standard for C# and the .NET CLR [38,71,105], the IEEE-VHDL93 standard [39]. The ASM refinement method [26] has been used in numerous ASM-based design and verification projects surveyed in [24].

The ASM method, due to the mathematical nature of its constituent concepts, could be linked to a multitude of tool-supported analysis methods, in terms of both experimental *validation* of models and mathematical *verification* of their properties. The validation (testing) of ASM models is supported by various tools to mechanically execute ASMs, including *ASM Workbench* [56], *AsmGofer* [100], an Asm2C^{++} compiler [101], C-based *XASM* [8], .NET-executable *AsmL* engine [68], *CoreASM* Execution Engine [63]. The verification of ASM properties has been performed using justification techniques ranging from proof sketches [41] over traditional [37,40] or formalised mathematical proofs [94, 106] to tool supported proof checking or interactive or automatic theorem proving, e.g. by model checkers [57, 73, 112], KIV [99] or PVS [60, 72]. As required for a comprehensive development and analysis environment, various combinations of such verification and validation methods have been supported and used for the correctness analysis of compilers [61, 86] and hardware [81, 102, 110, 111].

For more applications, including industrial system development and re-engineering case studies that demonstrate the scalability of the method to large systems, see the website of the ASM Research Center at `www.asmcenter.org` and the AsmBook [46].

3.8 Concluding Remarks

The ASM method is not a silver bullet, but shares the intrinsic limitations of every engineering discipline rooted in mathematics. Whereas ASMs are easily grasped and correctly understood by application domain experts and system engineers, namely as rule sets describing event triggerred actions, or as pseudo-code or FSMs over arbitrary data types, it is not an easy task to teach or to learn a judicious use of the inherent abstraction potential for constructing appropriate ground models and refinement hierarchies.

There are also pragmatical limitations, which the method shares with other rigorous practical methods, for example, the B-method [5]. They have to do with the proposed shift in the current software system development process. The proposal is not to start coding right away and not to relegate the correctness and reliability issues to an ever growing testing phase. Instead it is suggested to first construct and reason about accurate ground models for the requirements and exact interfaces for the various design decisions, leading to more and more detailed models. From that and only from that basis should executable code be generated, which then comes with objectively verifiable and validatable correctness properties one can trace through the refinement hierarchy to their ground model pendant. It is by no means easy to influence, let alone to change an established industrial practice.

Besides the engineering of complex software-based systems, we see other exciting directions where the ASM method can be (and partly has started to be) applied, mainly by exploiting the modelling potential of ASMs in areas where dynamic features are in the focus. Examples are business and similar technical processes [7, 77, 78], social processes [47] and biological systems [89].

Acknowledgements

I thank Paul Boca, Uwe Glässer, Antje Nowack, Bernhard Thalheim, Mona Vajihollahi and the anonymous referees for their critical comments on a preliminary version of this paper.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, MA, 1996.
2. J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *Proceedings of the 1st Conference on the B Method*, pp. 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.
3. J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *Proceedings of the FME 2003*, pp. 51–74. Springer, Berlin, 2003.
4. J.-R. Abrial. Discrete system models. Version 2, September 2004.
5. J.-R. Abrial. Formal methods in industry: Achievements, problems, future. In *Proceedings of the ICSE'06*, Shanghai, ACM, May 2006.
6. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, MA, 2009.
7. M. Altenhofen, E. Börger, A. Friesen, and J. Lemcke. A high-level specification for virtual providers. *International Journal of Business Process Integration and Management*, 1(4):267–278, 2006.
8. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at http://www.xasm.org/, 2001.
9. E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In G. Levi, H. Ehrig, R. Kowalski and U. Montanari, editors, *Proceedings of the TAPSOFT'87 Vol. 1*, Lecture Notes in Computer Science, vol. 249, pp. 169–201. Springer, Berlin, 1987.
10. E. Astesiano and G. Reggio. Labelled transition logic: An outline. *Acta Informatica*, 37(11–12), 2001.
11. R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.
12. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, Berlin, 1998.
13. A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering. Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, Lecture Notes in Computer Science, vol. 3785, pp. 5–35. Springer, Berlin, 2005.
14. R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 1998.
15. T. Bolognesi and E. Börger. Abstract state processes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, Lecture Notes in Computer Science, vol. 2589, pp. 22–32. Springer, Berlin, 2003.
16. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. K. Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, Lecture Notes in Computer Science, vol. 440, pp. 36–64. Springer, Berlin, 1990.
17. E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 452, pp. 1–14. Springer, Berlin, 1990.

18. E. Börger. Dynamische Algebren und Semantik von Prolog. In E. Börger, editor, *Berechenbarkeit, Komplexität, Logik*, pp. 476–499. Vieweg, Wiesbaden, 3rd edition, 1992.
19. E. Börger. A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In Y. N. Moschovakis, editor, *Logic From Computer Science*, Mathematical Sciences Research Institute Publications, Berkeley, CA, vol. 21, pp. 17–50. Springer, New York, 1992.
20. E. Börger. A natural formalization of full Prolog. *Newsletter of the Association for Logic Programming*, 5(1):8–9, 1992.
21. E. Börger. Logic programming: The evolving algebra approach. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress, vol. 1: Technology/ Foundations*, pp. 391–395. Elsevier, Amsterdam, 1994.
22. E. Börger. Evolving Algebras and Parnas tables. In H. Ehrig, F. von Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics.* Dagstuhl Seminar, Schloss Dagstuhl, Germany, July 1996.
23. E. Börger. High-level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, Lecture Notes in Computer Science, vol. 1641, pp. 1–43. Springer, Berlin, 1999.
24. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.
25. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N. Dershowitz, editor, *Verification: Theory and Practice*, Lecture Notes in Computer Science, vol. 2772, pp. 145–160. Springer, Berlin, 2003.
26. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
27. E. Börger. Modeling with abstract state machines: A support for accurate system design and analysis. In B. Rumpe and W. Hesse, editors, *Modellierung 2004*, GI-Edition, Lecture Notes in Informatics, vol. P-45, pp. 235–239. Springer, Berlin, 2004.
28. E. Börger. Abstract state machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 133:149–171, 2005.
29. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems*, Lecture Notes in Artificial Intelligence, vol. 3717, pp. 264–283. Springer, Berlin, 2005.
30. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 18:495–517, 2006.
31. E. Börger. From finite state machines to virtual machines (Illustrating design patterns and event-B models). In E. Cohors-Fresenborg and I. Schwank, editors, *Präzisionswerkzeug Logik–Gedenkschrift zu Ehren von Dieter Rödding.* Forschungsinstitut für Mathematikdidaktik Osnabrück, 2006. ISBN 3-925386-56-4, Proceedings of the of 2005 Colloquium.
32. E. Börger, H. Busch, J. Cuellar, P. Päppinghaus, E. Tiden, and I. Wildgruber. Konzept einer hierarchischen Erweiterung von EURIS. Siemens ZFE T SE 1 Internal Report BBCPTW91-1, pp. 1–43, Summer 1996.
33. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology,*

8th International Conference on AMAST 2000, Iowa, May 20–27, 2000 Proceedings, Lecture Notes in Computer Science, vol. 1816, pp. 293–308. Springer, Berlin, 2000.

34. E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, vol. 1912, pp. 223–241. Springer, Berlin, 2000.
35. E. Börger, M. Cesaroni, M. Falqui, and T. L. Murgi. Caso di Studio: Mail from form system. Internal Report FST-2-1-RE-02, Fabbrica Servizi Telematici FST (Gruppo Atlantis), Uta (Cagliari), 1999.
36. E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England, 1990.
37. E. Börger and I. Durdanović. Correctness of compiling occam to transputer code. *Computer Journal*, 39(1):52–92, 1996.
38. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.
39. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pp. 500–505, Los Alamitos, CA. IEEE Computer Society, 1994.
40. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, Lecture Notes in Computer Science, vol. 1212, pp. 151–187. Springer, Berlin, 1997.
41. E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.
42. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, vol. 1912, pp. 361–366. Springer, Berlin, 2000.
43. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.
44. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, *Studies in Computer Science and Artificial Intelligence*, vol. 11, chap. 2, pp. 20–90. Elsevier science, North-Holland, 1995.
45. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, Lecture Notes in Computer Science, vol. 1862, pp. 41–60. Springer, Berlin, 2000.
46. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, Berlin, 2003.
47. P. L. Brantingham, B. Kinney, U. Glässer, P. Jackson, and M. Vajihollahi. A computational model for simulating spatial and temporal aspects of crime in urban environments. In L. Liu and J. Eck, editors, *Artificial Crime Analysis Systems: Using Computer Simulations and Geographic Information Systems.* Idea Publishing, 2006.

48. P. L. Brantingham, B. Kinney, U. Glässer, K. Singh, and M. Vajihollahi. A computational model for simulating spatial aspects of crime in urban environments. In M. Jamshidi, editor, *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, pp. 3667–3674. IEEE, October 2005.
49. F. P. Brooks, Jr. No silver bullet. *Computer*, 20(4):10–19, 1987.
50. G. D. Castillo and P. Päppinghaus. Designing software for internet telephony: Experiences in an industrial development process. In A. Blass, E. Börger, and Y. Gurevich, editors, *Theory and Applications of Abstract State Machines*, Schloss Dagstuhl, International Conference and Research Center for Computer Science, 2002.
51. A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method*. PhD thesis, University of Catania, Sicily, 2000.
52. A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML static and dynamic views and formalizing the interaction mechanism of UML state machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, Lecture Notes in Computer Science, vol. 2589, pp. 229–243. Springer, Berlin, 2003.
53. A. B. Cremers and T. N. Hibbard. Formal modeling of virtual machines. *IEEE Transactions on Software Engineering*, SE-4(5):426–436, 1978.
54. W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, Cambridge, MA, 1998.
55. S. Degeilh and A. Preller. ASMs specify natural language processing via pregroups. Technical Report Local Proceedings ASM'04, Department of Mathematics, University Halle-Wittenberg, 2004.
56. G. Del Castillo. *The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. PhD thesis, Universität Paderborn, Germany, 2001.
57. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on TACAS 2000*, Lecture Notes in Computer Science, vol. 1785, pp. 331–346. Springer, Berlin, 2000.
58. J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Formal Approaches to Computing and Information Technology. Springer, Berlin, 2001.
59. E. W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pp. 1–82. Academic Press, New York, 1972.
60. A. Dold. A formal representation of abstract state machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
61. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proceedings of the 5th International Workshop on ASMs*, pp. 50–67. Magdeburg University, Germany, 1998.
62. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1,2*. Springer, Berlin, 1985.
63. R. Farahbod et al. *The CoreASM Project*, 2005. http://www.coreasm.org.
64. R. Farahbod, U. Glässer, and M. Vajihollahi. An abstract machine architecture for web service based business process management. *International Journal on Business Process Integration and Management*, 1(4):279–291, 2006.

65. L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design.* Cambridge University Press, Cambridge, MA, 1992.
66. L. M. G. Feijs, H. B. M. Jonkers, C. P. J. Koymans, and G. R. Renardel de Lavalette. Formal definition of the design language COLD-K. Technical Report No. 234/87, Philips Research Laboratories, 1987.
67. J. Fitzgerald and P. G. Larsen. *Modeling Systems. Practical Tool and Techniques in Software Development.* Cambridge University Press, Cambridge, MA, 1998.
68. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.
69. N. G. Fruja. The correctness of the definite assignment analysis in C#. *Journal of Object Technology*, 3(9):29–52, 2004.
70. N. G. Fruja. *Type Safety of C# and .NET CLR.* PhD thesis, ETH Zürich, 2006.
71. N. G. Fruja and E. Börger. Modeling the .NET CLR exception handling mechanism for a mathematical Analysis. *Journal of Object Technology*, 5(3): 5–34, 2006.
72. A. Gargantini and E. Riccobene. Encoding abstract state machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, vol. 1912, pp. 303–322. Springer, Berlin, 2000.
73. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, Lecture Notes in Computer Science, vol. 2589, pp. 278–292. Springer, Berlin, 2003.
74. C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE specification language.* Prentice-Hall, Upper Saddle River, NJ, 1992.
75. R. L. Glass. *Facts and Fallacies of Software Engineering.* Addison-Wesley, Reading, MA, 2003.
76. U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of SDL-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, 2003.
77. U. Glässer, S. Rastkar, and M. Vajihollahi. Computational modeling and experimental validation of aviation security procedures. In S. Mehrotra, D. D. Zeng, H. Chen, B. M. Thuraisingham, and F.-Y. Wang, editors, *Intelligence and Security Informatics, IEEE International Conference on Intelligence and Security Informatics, ISI 2006, San Diego, CA, May 23–24, 2006, Proceedings*, Lecture Notes in Computer Science, vol. 3975, pp. 420–431. Springer, Berlin, 2006.
78. U. Glässer, S. Rastkar, and M. Vajihollahi. Modeling and validation of aviation security. In H. Chen and C. C. Yang, editors, *Intelligence and Security Informatics: Techniques and Applications.* Springer, Berlin, 2006.
79. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pp. 9–36. Oxford University Press, 1995.
80. J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978.
81. A. Habibi. *Framework for System Level Verification: The SystemC Case.* PhD thesis, Concordia University, Montreal, July 2005.
82. C. Heitmeyer. Software cost reduction. In John J. Marciniak, editor, *Encyclopedia of Software Engineering.* 2nd edition, 2002.

83. G. Hommel. Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage. Technical report, Kernforschungszentrum Karlsruhe, August 1980.
84. M. Jackson. *Problem Frames*. Addison-Wesley, Reading, MA, 2001.
85. D. E. Johnson and L. S. Moss. Grammar formalisms viewed as evolving algebras. *Linguistics and Philosophy*, 17:537–560, 1994.
86. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, Lecture Notes in Computer Science, vol. 2589, page 415. Springer, Berlin, 2003.
87. B. H. Lisko and S. N. Zilles. Specification techniques for data abstraction. *IEEE Transactons on Software Engineering*, SE-1, 1975.
88. K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison-Wesley, Boston, MA, 1988.
89. D. Mazzei. Ottimizzazione di un bioreattore high throughput con strategia di controllo autonoma. Master's thesis, University of Pisa, October 2006.
90. L. Mearelli. Refining an ASM specification of the production cell to C++ code. *Journal of Universal Computer Science*, 3(5):666–688, 1997.
91. J. M. Morris. A theoretical basis for stepwise refinement. *Science of Computer Programming*, 9(3):287–306, 1987.
92. L. S. Moss and D. E. Johnson. Dynamic interpretations of constraint-based grammar formalisms. *Journal of Logic, Language, and Information*, 4(1):61–79, 1995.
93. L. S. Moss and D. E. Johnson. Evolving algebras and mathematical models of language. In L. Polos and M. Masuch, editors, *Applied Logic: How, What, and Why*, Synthese Library, vol. 626, pp. 143–175. Kluwer Academic Publishers, Dordecht, 1995.
94. S. Nanchen and R. F. Stärk. A security logic for Abstract State Machines. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, Lecture Notes in Computer Science, vol. 3052, pp. 169–185. Springer, Berlin, 2004.
95. D. L. Parnas. The use of precise documentation in software development. Tutorial at FM 2006, see `http://fm06.mcmaster.ca/t8.htm`, August 2006.
96. D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–62, 1995.
97. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *Journal of Universal Computer Science*, 7(11):952–979, 2001.
98. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2–3):403–436, 2005.
99. G. Schellhorn and W. Ahrendt. Reasoning about abstract state machines: The WAM case study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.
100. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.
101. J. Schmid. Compiling abstract state machines to C++. *Journal of Universal Computer Science*, 7(11):1069–1088, 2001.
102. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.

103. Semiconductor Industry Assoc. Internat. technology roadmap for semiconductors. Design. http://www.itrs.net/Common/2005ITRS/Design2005.pdf, 2005.
104. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Upper Saddle River, NJ, 1989.
105. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, Lecture Notes in Computer Science, vol. 3052, pp. 38–60. Springer, Berlin, 2004.
106. R. F. Stärk and S. Nanchen. A logic for abstract state machines. *Journal of Universal Computer Science*, 7(11):981–1006, 2001.
107. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, Berlin, 2001.
108. W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, 1982.
109. A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936.
110. J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, Lecture Notes in Computer Science, vol. 1912, pp. 266–286. Springer, Berlin, 2000.
111. J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for ASIPs. In *Proceeding of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pp. 26–33, San Jose, CA, ACM, November 2000.
112. K. Winter. Model checking for abstract state machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.
113. N. Wirth. Program development by stepwise refinement. *Communication of the ACM*, 14(4):221–227, 1971.
114. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Upper Saddle River, NJ, 1996.

4

Applications and Methodology of νZ

Martin C. Henson

Department of Computer Science, University of Essex, UK
hensm@essex.ac.uk

Abstract In this chapter, we describe a specification logic called νZ. This is a Z-like formal method in which specifications are theories, not simply definitions. We examine simple applications and discuss some methodological issues that these illustrate. The chapter is introductory, and should be comprehensible to a reader with a knowledge of predicate logic and some familiarity with the ideas of computer system specification.

4.1 Introduction

In [3] Anthony Hall argues forcefully, and with compelling supporting evidence, that the major practical benefits of formal methods are those which follow from our ability to *reason* about precise system specifications at the earliest stages of development. This is the philosophy of νZ, a "new Z"-like specification framework: specifications are not just definitions, they are theories; so, a specification's fitness for purpose (the extent to which it captures intended requirements) is an issue which is addressed as soon as it is created.

This chapter is introductory; it provides an overview of νZ and then covers simple applications and methodology. νZ is a *logic*, so we obviously provide rules and (some) proofs alongside the example specifications, but we avoid, as far as possible, the mathematical *foundations* of νZ; these are covered in detail in [5]. It is also part of the νZ philosophy that a specification framework should be as simple as possible: if a formal method is ever to be seriously and widely *used*, then it will have to be easily understood – an advanced degree in mathematics should not be required in order to pick up the tool, let alone wield it; and νZ is, in fact, barely larger or more complicated than predicate logic. A good yardstick might be to assume the competence of a reasonably bright second-year computer science undergraduate, and indeed this chapter has been written with such a level of experience in mind. Specifically, to follow this chapter, we assume the reader has an acquaintance (at the level of an introductory undergraduate course) with predicate logic, and some experience of the general principles and ideas underlying programming and specification. A little knowledge of Z might also be helpful, but beware: appearances can be deceptive. νZ, as we will see in detail later, is in many ways very different from Z.[1]

4.2 The Specification Logic νZ – Overview I

νZ, like Z, is a framework with an *algebra of schemas* at its heart. Unlike Z, it is very small; in this first overview section, we introduce simple specifications and all its specification-forming operations. Together with the material on operation specifications in Sect. 4.4, this covers νZ in its entirety.

4.2.1 Atomic State Specifications

Here is a simple example of a *specification* (also called a *schema*):

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Dial}} \\ \hline
\textit{setting} \in \mathbb{N} \\ \hline
\textit{setting} < 10 \\ \hline
\end{array}$$

Specifications of this kind are essentially sets of states. Above the line we have a *declaration* of an *observation* (in this case *setting*) and below the line is an *invariant* (a constraint that is invariably true of all states of the *Dial* system). This specification can also be written horizontally:

$$\textit{Dial} =_{df} [\textit{setting} \in \mathbb{N} \mid \textit{setting} < 10]$$

The states which inhabit such specifications are called *bindings*; in this case having the form $\langle\!| \textit{setting} \Rrightarrow n |\!\rangle$, and one can extract the information from this using *binding selection* (also called *dot selection*): $\langle\!| \textit{setting} \Rrightarrow n |\!\rangle.\textit{setting} = n$. An obvious question is this: for which values of n is a binding of the form $\langle\!| \textit{setting} \Rrightarrow n |\!\rangle$ a member of *Dial*? The answer is fairly obvious: when n is a natural number less than ten. Luckily, making such a schema definition in νZ automatically establishes a little theory of the *Dial*, within which such judgements can easily be established. For example, we have an (introduction) rule:

$$\frac{n \in \mathbb{N} \quad n < 10}{\langle\!| \textit{setting} \Rrightarrow n |\!\rangle \in \textit{Dial}}$$

[1] There are a few footnotes in this chapter which may be of interest to the Z sophisticate.

which establishes the conditions under which we may conclude that a binding belongs to the specification. We can already use this rule to construct an elementary proof:

$$\frac{\overline{9 \in \mathbb{N}} \quad \overline{9 < 10}}{\langle\!| \, \mathit{setting} \Rrightarrow 9 \, |\!\rangle \in \mathit{Dial}}$$

Conversely, if we have a state which we *knew* was in *Dial*, then there are (elimination) rules which allow us to conclude that it satisfies the conditions laid down in the definition:

$$\frac{\langle\!| \, \mathit{setting} \Rrightarrow n \, |\!\rangle \in \mathit{Dial}}{n \in \mathbb{N}} \qquad \frac{\langle\!| \, \mathit{setting} \Rrightarrow n \, |\!\rangle \in \mathit{Dial}}{n < 10}$$

These three rules completely describe the state space *Dial*. In particular, the elimination rules enable us to show certain negative properties. Consider the following:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\langle\!| \, \mathit{setting} \Rrightarrow 11 \, |\!\rangle \in \mathit{Dial}}\ (1)}{11 < 10} \qquad \cfrac{}{11 \not< 10}}{\mathit{false}}}{\langle\!| \, \mathit{setting} \Rrightarrow 11 \, |\!\rangle \notin \mathit{Dial}}\ (1)$$

This uses two rules from predicate logic. *Reductio ad absurdum*:

$$\frac{P \vdash \mathit{false}}{\neg P}$$

and *contradiction*:

$$\frac{P \quad \neg P}{\mathit{false}}$$

The numerical labels in the proof (here (*1*)) indicate the places where assumptions are used and discharged. Let's call this result *Lemma* 1 – we will use it again shortly.

We said that the rules that establish the *Dial* theory arise *automatically* from the definition. How? Simply because *Dial* is a special case of a general theory of specifications. These have the form:

$$[S \mid P]$$

where S is (another) specification (the declaration) and P is an assertion (the invariant). There are then general rules for this general case:

$$\frac{s \in S \quad s.P}{s \in \big[S \mid P\big]} \qquad \frac{s \in \big[S \mid P\big]}{s.P} \qquad \frac{s \in \big[S \mid P\big]}{s \in S}$$

and those for *Dial* are just the special cases of these which arise when S is the schema $[\mathit{setting} \in \mathbb{N}]$ and P is the assertion $\mathit{setting} < 10$. The schema

$[setting \in \mathbb{N}]$ is particularly simple: the proposition is missing, or rather more precisely, it is *true* (and hence omitted); such simple schemas are called *schema sets*. Note that two of the rules involve the notation $s.P$. This is a simple extension of binding selection to assertions, allowing us to distribute the binding s throughout the assertion P in search (so to speak) of all the observations it contains. For example: $s.(setting < 10)$ is $s.setting < 10$ (we will see other examples).

We can now give the full syntax for what we call *atomic states specifications* in νZ:

$$\begin{array}{lll} S ::= & [\cdots\ \mathtt{z}_i \in C_i\ \cdots] & \text{– schema sets} \\ & [S \mid P] & \text{– atomic state schemas} \end{array}$$

Note that *Dial* involved just one observation, *setting*; in general we may introduce as many as we wish; in the general case, the observations are the $\mathtt{z}_i$ (we show just the ith of the sequence) and these are declared as belonging to the sets C_i. When S in $[S \mid P]$ is a schema set, we lose its outer brackets (as in *Dial*, which we did not write as $[[setting \in \mathbb{N}] \mid setting < 10]$). Now, we have already given the rules for atomic state specifications; those for schema sets comprise one introduction rule:

$$\frac{\cdots \quad s.\mathtt{z}_i \in C_i \quad \cdots}{s \in [\cdots\ \mathtt{z}_i \in C_i\ \cdots]}$$

and one elimination rule for each observation (here we give the ith):

$$\frac{s \in [\cdots\ \mathtt{z}_i \in C_i\ \cdots]}{s.\mathtt{z}_i \in C_i}$$

It is often useful to know which observations are used by a specification S. We write αS for this *alphabet set*.

4.2.2 Specification-Forming Operations

In νZ, as in Z, a central idea is that we can combine small specifications to form larger and more complex system descriptions; for this, we have a number of combining forms. In νZ there are very few:

$$\begin{array}{lll} S ::= & X & \text{– variable} \\ & \neg S & \text{– negation} \\ & S \vee S & \text{– disjunction} \\ & \exists\, \mathtt{x} \bullet S & \text{– existential hiding} \end{array}$$

There is in fact one other operator which permits the construction of recursive schemas. Although these are enormously important, we have not included them in this introductory chapter; there are many details in [5]. Each of these operators is, of course, characterised by a number of rules; and there is something interesting to say about each of them. First, negation:

$$\frac{s \notin S}{s \in \neg S} \qquad \frac{s = \perp}{s \in \neg S}$$

$$\frac{s \in \neg S \quad s \notin S \vdash P \quad s = \perp \vdash P}{P}$$

The idea, of course, is that $\neg S$ should contain all the states which are ruled out (by the constraint) of S. For example, consider the following little proof, which uses the first of these rules and Lemma 1:

$$\frac{\overline{\langle\!| \mathit{setting} \Rrightarrow 11 |\!\rangle \notin \mathit{Dial}}\ \mathit{Lemma}1}{\langle\!| \mathit{setting} \Rrightarrow 11 |\!\rangle \in \neg \mathit{Dial}}$$

Just take note of the second introduction rule (which mentions an undefined state, written $\perp$) and the slightly complicated elimination rule. We do not need to discuss this in great detail, but simply note that this undefined state is a member of *all* specifications; this is captured by the (derived) rule (called $\perp_S$):

$$\frac{s = \perp}{s \in S}$$

Clearly, taking the complement of S to form $\neg S$ would remove this state from $\neg S$ – it is, therefore, necessary to add it explicitly by means of the second introduction rule. The interested reader will find further details in [5]. Luckily, this little complication does not do any damage and we can still prove some expected properties of negation (which are all we really need in this chapter):

$$\frac{s \in S}{s \in \neg\neg S} \qquad \frac{s \in \neg\neg S}{s \in S} \qquad \frac{}{s \in S \vee \neg S}$$

Note that all possible states belong to $S \vee \neg S$; we will also call this specification chaos_S.

Rules for disjunction schemas:

$$\frac{s \in S_0}{s \in S_0 \vee S_1} \qquad \frac{s \in S_1}{s \in S_0 \vee S_1} \qquad \frac{s \in S_0 \vee S_1 \quad s \in S_0 \vdash P \quad s \in S_1 \vdash P}{P}$$

This all looks very familiar, being very similar to disjunction in predicate logic. There is, nevertheless, something of interest here: the alphabets of S_0 and S_1 may not be the same. Thus, a state s which is a member of $S_0 \vee S_1$ will generally mention more observations than occur in S_0 or S_1 individually. In νZ, it is, however, reasonable to write $s \in S$ even when s mentions observations which are not present in S – the truth (or otherwise) of $s \in S$ depends only on those observations actually mentioned in S. For example:

$$\langle\!| \mathit{setting} \Rrightarrow 3, \mathit{level} \Rrightarrow 2 |\!\rangle \in \mathit{Dial}$$

is true, despite the fact that *Dial* does not mention the observation *level*. Rules for existential hiding schemas:

$$\frac{s_0 \star \langle\!|\, \mathtt{x} \Rrightarrow s_1 \,|\!\rangle \in S}{s_0 \in \exists\, \mathtt{x} \bullet S} \qquad \frac{s \in \exists\, \mathtt{x} \bullet S \quad s \star \langle\!|\, \mathtt{x} \Rrightarrow y \,|\!\rangle \in S \vdash P}{P}$$

where y is a fresh variable. Again, this is reminiscent of predicate logic. There is a new operation on bindings here, called *concatenation*, and indicated by the star. For example:

$$\langle\!|\, \mathtt{z}_0 \Rrightarrow v_0 \,|\!\rangle \star \langle\!|\, \mathtt{z}_1 \Rrightarrow v_1 \,|\!\rangle = \langle\!|\, \mathtt{z}_0 \Rrightarrow v_0, \mathtt{z}_1 \Rrightarrow v_1 \,|\!\rangle$$

Note that the alphabet operator can be extended to all expressions and, in particular, that $\alpha(\exists\, \mathtt{x} \bullet S)$ is $\alpha S - \{\mathtt{x}\}$.

4.3 Case Studies and Methodology I

νZ also encourages the definition of new operators, something which is almost impossible in Z (consult [8] to see the very real difficulties). This permits a second dimension of modularity in νZ: we will call the use of specification-forming operators to form larger systems from smaller ones (as in Z) *horizontal* modularity; we will call the use of layered schema operator definitions (in νZ) *vertical* modularity, which will be discussed in more detail in Sect. 4.5.3. Vertical modularity helps to keep definitions small, theories concise and (most importantly) proofs as simple as possible.

4.3.1 Conjunction

Using specification variables, we can define schema conjunction:

$$X_0 \wedge X_1 =_{df} \neg(\neg X_0 \vee \neg X_1)$$

This uses the standard De Morgan definition and so it may be thought to be obviously *correct*. Indeed, it is not difficult to make this precise by proving, of our new operator, the following:

$$\frac{s \in S_0 \quad s \in S_1}{s \in S_0 \wedge S_1} \qquad \frac{s \in S_0 \wedge S_1}{s \in S_0} \qquad \frac{s \in S_0 \wedge S_1}{s \in S_1}$$

In order to illustrate how a theory for a new operator is created, we provide the proof for the last of these:

$$\dfrac{s \in \neg(\neg S_0 \vee \neg S_1) \qquad \dfrac{\dfrac{\dfrac{}{s \notin \neg S_0 \vee \neg S_1}\,(1) \qquad \dfrac{\dfrac{}{s \in \neg S_1}\,(2)}{s \in \neg S_0 \vee \neg S_1}}{\mathit{false}}}{s \in S_1}\,(2) \qquad \dfrac{\dfrac{}{s = \bot}\,(1)}{s \in S_1}}{s \in S_1}\,(1)$$

Note how this uses the rules from the theories of the operators that are used to make the definition; in particular, the elimination rule for negation schemas and the second introduction rule for disjunction schemas; in addition, the rules $\bot_S$, *contradiction* and *reductio ad absurdum* make an appearance. This is our first example specification in νZ – it is interesting because what we are specifying is an additional *specification* operator. As we will see, we can then go on to use this to form other specifications.

4.3.2 Methodology I

There are three really important methodological observations to make at this point. First, that *specifications are theories not definitions*. This means that alongside *every* specification we make, we establish logical rules which characterise it. Second, the theory itself gives us confidence that the specification *makes sense*: captures our intentions. These two points are exemplified by our simple specification of conjunction. Third, once we have a theory for some specification we can go on to prove additional properties, establishing an auxiliary theory that often strengthens our confidence in that specification. This last point is exemplified by the additional properties of negation, which we provided in Sect. 4.2.2, especially so since the theory for negation comprises rules which are not quite what we might have expected.

4.3.3 Primed and Δ-Specifications

To each observation z we have its *co-observation* z' such that $\mathsf{z}'' = \mathsf{z}$ (this differs from Z). We can easily extend this to bindings, propositions and schemas. For example:

$$\begin{aligned}
\langle\!| \, \mathit{setting} \Rrightarrow 3 \, |\!\rangle' &= \langle\!| \, \mathit{setting}' \Rrightarrow 3 \, |\!\rangle \\
(\mathit{setting} < 10)' &= \mathit{setting}' < 10 \\
\big[\mathit{setting} \in \mathbb{N} \mid \mathit{setting} < 10\big]' &= \big[\mathit{setting}' \in \mathbb{N} \mid \mathit{setting}' < 10\big]
\end{aligned}$$

The rules are obvious:

$$\frac{s \in S}{s' \in S'} \qquad \frac{s' \in S'}{s \in S}$$

Combining this with conjunction in place, we can define Δ-specifications:

$$\Delta X =_{df} X \wedge X'$$

The rules are very similar, of course, to those for conjunction. The purpose of priming and Δ-specifications will be clear on reading the next section.

4.4 The Specification Logic νZ – Overview II

In this section, we show how operations over states are specified in νZ, and explain how these differ from Z.

4.4.1 Operation Specifications

Here is a first example:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{IncreaseDial}} \\ \hline
\Delta \mathit{Dial} \\ \hline
\mathit{setting} < 9 \\ \hline
\mathit{setting}' = \mathit{setting} + 1 \\ \hline
\end{array}$$

This differs from Z in separating the *precondition* (here: $\mathit{setting} < 9$) from the *postcondition* (here: $\mathit{setting}' = \mathit{setting} + 1$). Its rules automatically follow:

$$\frac{s_0 \in \mathit{Dial} \quad s_1 \in \mathit{Dial} \quad s_0.\mathit{setting} < 0 \;\vdash s_1.\mathit{setting} = s_0.\mathit{setting} + 1}{s_0 \star s_1' \in \mathit{IncreaseDial}}$$

$$\frac{s_0 \star s_1' \in \mathit{IncreaseDial}}{s_0 \in \mathit{Dial}} \qquad \frac{s_0 \star s_1' \in \mathit{IncreaseDial}}{s_1 \in \mathit{Dial}}$$

$$\frac{s_0 \star s_1' \in \mathit{IncreaseDial} \quad s_0.\mathit{setting} < 0}{s_1.\mathit{setting} = s_0.\mathit{setting} + 1}$$

The members of the operation are *state transitions*, written $s_0 \star s_1'$ where s_0 (the *before state*) is in *Dial* and s_1' (the *after state*) is in Dial'. These rules are derived from the general case for *operation specifications*:

$$U \;::=\; [\Delta S \mid P \mid Q] \text{ – atomic operation specifications}$$

with rules:

$$\frac{s_0 \in S \quad s_1 \in S \quad s_0.P \vdash s_0.s_1'.Q}{s_0 \star s_1' \in \big[\Delta S \mid P \mid Q\big]} \qquad \frac{s_0 \star s_1' \in \big[\Delta S \mid P \mid Q\big]}{s_0 \in S}$$

$$\frac{s_0 \star s_1' \in \big[\Delta S \mid P \mid Q\big]}{s_1 \in S} \qquad \frac{s_0 \star s_1' \in \big[\Delta S \mid P \mid Q\big] \quad s_0.P}{s_0.s_1'.Q}$$

These atomic operation specifications can be combined to form specification expressions using the same specification-forming operations we introduced earlier, and with the same rules.

4.4.2 Adding Inputs and Outputs

Sometimes we will wish to include additional inputs and outputs in the specification of a operation. We use standard decorations for these, for example $\mathtt{z}?$ is an input observation (and part of the before state) and $\mathtt{z}!$ is an output observation (and part of the after state).

To maintain symmetry, we *do* have co-observations corresponding to input observations z? and output observations z! . These are z¿ and z¡, respectively, but z¿ is *never used* as an output observation and z¡ is *never used* as an input observation. For example, consider:

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
val \in \mathbb{N} \\ \hline
\end{array}$$

and:

$$\begin{array}{|l}
\multicolumn{1}{l}{U} \\ \hline
\Delta S \\
in? \in \mathbb{N} \\
out! \in \mathbb{N} \\ \hline
in? < val \\ \hline
out! = in? \\ \hline
\end{array}$$

The declaration amounts to $S \wedge [in?, out¡ \in \mathbb{N}] \wedge S' \wedge [out!, in¿ \in \mathbb{N}]$, which is $\Delta[val, in?, out¡ \in \mathbb{N}]$ or equivalently, $\Delta[val', in¿, z! \in \mathbb{N}]$. The symmetry only serves to keep the declaration of *all* operation specifications of the form ΔS and means that we do not need to introduce additional (more complex) rules for operation specifications: those above suffice.

4.5 Case Studies and Methodology II

In this section, we turn to less Z-like examples, beginning with *guarded* and *conditional* specifications; these less familiar examples raise a number of questions that allow us to discuss further methodological issues.

4.5.1 Guarded Specifications

Let us now turn to an example where the specification is less immediately obviously correct: *guarded specifications.* The idea here is to provide an operator that permits us to introduce a firing condition G (containing only before observations) in addition to a precondition (that may be weakened). The specification we propose is:

$$G \longrightarrow X =_{df} \left[\, chaos_X \mid \neg G \mid G \,\right] \wedge X$$

Is this right though? Does it capture the intended behaviour? Indeed, what *is* the intended behaviour? For example, in [7] the guard must be weaker than the precondition, but one might be more permissive. What condition, if any, have we imposed here? Before we can address any of these questions, we can and must establish the theory of guarded schemas. These rules, as usual, are derived from the definition (in this case from the rules for, atomic schemas, chaos, negation and conjunction):

$$\frac{t.G \quad t \in U}{t \in G \longrightarrow U} \qquad \frac{t \in G \longrightarrow U}{t.G} \qquad \frac{t \in G \longrightarrow U}{t \in U}$$

Intuitively, the introduction rule tells us that a state transformation t is in the guarded specification $G \longrightarrow U$, if it satisfies the guard and it is in U too. Taken together with the elimination rules, the theory tells us that such t are the *only* state transformations which are in $G \longrightarrow U$. In particular, those t in U which fail to satisfy the guard are not in $G \longrightarrow U$. This is all quite satisfactory, but as definitions become more complex, the base theory, in itself, may not provide sufficient explicit convincing information. We will return to the guarded specification in the next section, extending its theory to capture additional properties.

4.5.2 Conditional Specifications

With guards in place, we can specify further interesting operators. Consider the following conditional specification:

$$\texttt{if } D \texttt{ then } X_0 \texttt{ else } X_1 =_{df} D \longrightarrow X_0 \vee \neg D \longrightarrow X_1$$

The theory for this operator is then easily derived (from the rules for disjunction and guarded specifications):

$$\frac{t.D \quad t \in U_0}{t \in \texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1}$$

$$\frac{\neg t.D \quad t \in U_1}{t \in \texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1}$$

$$\frac{t \in \texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1 \quad t.D, t \in U_0 \vdash P \quad \neg t.D, t \in U_1 \vdash P}{P}$$

The first introduction rule says: "*A state transition t in U_0 which satisfies D is also in* $\texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1$", which is reasonable and expected. Once again, these rules constitute a theory of the conditional, which aligns well with our intuitions and provides us with significant confidence that we have the "correct" specification: the one which captures the requirements.

There is, however, an alternative approach. To see this, we need to introduce two more specification constructs. First, a restricted form of chaos

$$chaos_P =_{df} \left[\neg P \mid false\right]$$

with the rules:

$$\frac{t_0.P}{t_0 \star t_1' \in chaos_P}\ (chaos_P^+) \qquad \frac{t_0 \star t_1' \in chaos_P \quad \neg t_0.P}{false}\ (chaos_P^-)$$

Second, an operator allowing us to strengthen the precondition of a specification

$$U \upharpoonright P =_{df} chaos_P \rightarrow U$$

with the rules:

$$\frac{t.P \vdash t \in U}{t \in U \upharpoonright P} \qquad \frac{t \in U \upharpoonright P \quad t.P}{t \in U}$$

This allows us to define the conditional specification as follows:

$$\texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1 =_{df} U_0 \upharpoonright D \wedge U_1 \upharpoonright \neg D$$

The rules for this version of the conditional are:

$$\frac{t.D \vdash t \in U_0 \quad \neg t.D \vdash t \in U_1}{t \in \texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1}\ (\texttt{if}^+)$$

$$\frac{t \in \texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1 \quad t.D}{t \in U_0}\ (\texttt{if}_0^-)$$

$$\frac{t \in \texttt{if } D \texttt{ then } U_0 \texttt{ else } U_1 \quad \neg t.D}{t \in U_1}\ (\texttt{if}_1^-)$$

Once again, these are very intuitive and they suggest that we have captured our intentions, but note that they are very different from the rules we had for our first definition. So, which is the *correct* definition? Well, of course, we should hope that they are *both* correct; and that can be verified by demonstrating that the two theories are equivalent.

There is, in fact, a rather nice and general methodology for this: we show that the membership conditions of one theory satisfy the elimination rule (or rules) of the other theory. Since the introduction and elimination rules of every theory enjoy a symmetry property, this is sufficient to show that all judgements of the first theory are also judgements of the second.

We can illustrate this. Let us write $\texttt{if}_{fst}$ for the first definition of the conditional and $\texttt{if}_{snd}$ for the second definition. Then we can prove that:

$$\frac{t \in \texttt{if}_{fst}\ D \texttt{ then } U_0 \texttt{ else } U_1 \quad t.D}{t \in U_0}$$

and:

$$\frac{t \in \texttt{if}_{fst}\ D \texttt{ then } U_0 \texttt{ else } U_1 \quad \neg t.D}{t \in U_1}$$

That is, the first definition satisfies the elimination rules of the second theory. Then, using the introduction rule of the second theory, we have:

$$\dfrac{\dfrac{t \in \texttt{if}_{fst}\ D\ \texttt{then}\ U_0\ \texttt{else}\ U_1 \quad \overline{t.D}^{\ (1)}}{t \in U_0} \qquad \dfrac{t \in \texttt{if}_{fst}\ D\ \texttt{then}\ U_0\ \texttt{else}\ U_1 \quad \overline{\neg t.D}^{\ (1)}}{t \in U_1}}{t \in \texttt{if}_{snd}\ D\ \texttt{then}\ U_0\ \texttt{else}\ U_1}\ (1)$$

This shows that the first theory of the conditional is contained in the second. Showing that the second theory satisfies the elimination rule of the first theory establishes the converse and completes the equivalence proof. This is an entirely general methodology when theories are specified in the symmetrical form of introduction and elimination rules.

4.5.3 Methodology II

Further important methodological points arise here. First, the ease with which it is possible to introduce new schema operators encourages a modular *layered* approach to specification construction and theory development. We are used, in Z for example, to *horizontal* modularity: the development of larger specifications from smaller components by assembling them as schema expressions that we form using a fixed set of schema operators. Here, in νZ, the ability to introduce new operators allows an additional *vertical* modularity that keeps theories small and well organised. To see this, consider the conditional specification we introduced above. It was first defined using disjunction and guarded specifications; the latter are defined using atomic schema and conjunction; and, as we have seen, conjunction is defined in terms of negation and disjunction. Without this vertical modularity, the definition in the base νZ logic would be:

$$\texttt{if}\ D\ \texttt{then}\ X_0\ \texttt{else}\ X_1 =_{df} \neg(\neg \left[\neg D \mid D \right] \vee \neg X_0) \vee \neg(\neg \left[D \mid \neg D \right] \vee \neg X_1)$$

which is rather opaque to say the least. However, the *real* benefit is in theory development. To see this very forcefully, it would be a useful exercise to compare the proof of the introduction rule for the conditional using each of the two definitions; the difference in the proofs is enormously more pronounced than the difference between the two definitions themselves.

A second methodological point concerns the *ease* with which informal requirements can be formalised: how easy would it be to come up with the *unpacked* specification of the conditional given above? There is a considerable conceptual distance between (just) the theories of disjunction and negation and the theory of the conditional that we hope to capture, and much less distance between the conditional and the theories of guards and disjunction (or, indeed, the strengthened preconditions and conjunction used in our second definition).

A third methodological point concerns our confidence that we have made the *correct* specification: that we have successfully captured our informal requirements precisely. We have already argued that the existence of rules for reasoning about specifications can greatly increase our confidence in what we have formalised, and having *two* (or more) definitions for a single concept, written from different perspectives and that are demonstrably equivalent, provides enormous evidence that we have captured our intuitions. This we illustrated with our two different, but equivalent, specifications of the conditional.

A fourth methodological point concerns the *purposes* or uses of schema operators. In Z, schema operators are very clearly intended to structure specifications; a program may not follow that structure at all. In νZ, we use schema operators to structure specifications as we do in Z, but there are additional possibilities: schema operators might also be used to introduce design or program structuring, in addition to specification structuring. The conditional is a case in point: this might very well be used as a device for program structuring.

4.5.4 Varieties of Hiding

There is a useful variation on the theme of the basic existential hiding schema provided in the core νZ framework, permitting the hiding of a schema. First, we have the obvious generalisation of the basic quantifier:

$$\exists\, \mathtt{z}_0 \cdots \mathtt{z}_n \bullet X =_{df} \exists\, \mathtt{z}_0 \bullet \cdots \exists\, \mathtt{z}_n \bullet X$$

with the obvious theory. Then we can introduce:

$$\exists\, X_0 \bullet X_1 =_{df} \exists \cdots \mathtt{x}_i \cdots \bullet X_0 \wedge X_1$$

where $\alpha X_0 = \{\cdots \mathtt{x}_i \cdots\}$. This leads to the following theory:

$$\frac{t \star \langle\!| \cdots \mathtt{x}_i \Rrightarrow t_i \cdots |\!\rangle \in U_0 \quad t \star \langle\!| \cdots \mathtt{x}_i \Rrightarrow t_i \cdots |\!\rangle \in U_1}{t \in \exists\, U_0 \bullet U_1}$$

$$\frac{t \in \exists\, U_0 \bullet U_1 \quad t \star \langle\!| \cdots \mathtt{x}_i \Rrightarrow y_i \cdots |\!\rangle \in U_0, t \star \langle\!| \cdots \mathtt{x}_i \Rrightarrow y_i \cdots |\!\rangle \in U_1 \vdash P}{P}$$

where the y_i are fresh variables. We will use this generalisation in Sect. 4.7.3 below.

4.5.5 θ-Terms and Ξ-Schemas

Another useful notion (also very familiar from Z) are θ-terms. These are badly described in much of the literature, which is understandable since their formal treatment is extremely awkward. Some progress was made with the introduction of *characteristic bindings* (see [9]), although this does not help immediately with such terms of primed type, nor with distinguishing between observations (constants) and variables (see [4] for an in-depth analysis). This leads to another difference between Z and νZ:

$$\theta S =_{df} \langle\!| \cdots\ \mathtt{z}_i \Rrightarrow \mathtt{z}_i\ \cdots |\!\rangle$$

where $\alpha S = \{\cdots \mathtt{z}_i\ \cdots\}$. Note that θ' is then $\langle\!| \cdots\ \mathtt{z}'_i \Rrightarrow \mathtt{z}'_i\ \cdots |\!\rangle$, but this is not what is intended (and exemplifies a difficulty in Z). In fact, we need a second definition:

$$\theta' S =_{df} \langle\!| \cdots\ \mathtt{z}_i \Rrightarrow \mathtt{z}'_i\ \cdots |\!\rangle$$

With these in place, it is possible to define the standard notion of a Ξ-schema:

$$\Xi X =_{df} [\Delta X \mid \theta X = \theta' X]$$

The theory is:

$$\frac{s_0 \in S \quad s_1 \in S \quad s_0 = s_1}{s_0 \star s'_1 \in \Xi S} \qquad \frac{s_0 \star s'_1 \in \Xi S}{s_0 \in S} \qquad \frac{s_0 \star s'_1 \in \Xi S}{s_1 \in S} \qquad \frac{s_0 \star s'_1 \in \Xi S}{s_0 = s_1}$$

We will need Ξ-schemas in Sect. 4.7.1.

4.6 Refinement

Before we move on to further examples, it is time to consider the fundamental relation that allows us to compare specifications: *refinement*. νZ does not specify any particular notion of refinement; this is another area in which the framework is entirely flexible. The simplest notion, *operation refinement*, is easily introduced:[2]

$$X_0 \sqsupseteq X_1 =_{df} X_0 \subseteq X_1$$

One might also introduce notions of data refinement, but we will not do so in this chapter. The rules for refinement are as follows: Let z be a fresh variable

$$\frac{z \in U_0 \vdash z \in U_1}{U_0 \sqsupseteq U_1} \qquad \frac{U_0 \sqsupseteq U_1 \quad t \in U_0}{t \in U_1}$$

We can now explore how our specifications interact with refinement.

[2] A reader familiar with Z may be surprised, as this is once again different from Z. The usual *lifted-totalisation* is not necessary here: there are technical details in [5].

4.6.1 Refining Atomic Operations

For atomic operation schema, we can prove, as we probably expected, that we may weaken the precondition, strengthen the postcondition and refine the declaration:

$$\frac{S_0 \sqsupseteq S_1}{[\, S_0 \mid P \mid Q \,] \sqsupseteq [\, S_1 \mid P \mid Q \,]}$$

and:

$$\frac{z.P_1 \vdash z.P_0}{[\, S \mid P_0 \mid Q \,] \sqsupseteq [\, S \mid P_1 \mid Q \,]} \qquad \frac{z.Q_0 \vdash z.Q_1}{[\, S \mid P \mid Q_0 \,] \sqsupseteq [\, S \mid P \mid Q_1 \,]}$$

Such results provide confidence on two fronts: in this case that our notion of atomic operation schema and our notion of refinement are both properly specified.

4.6.2 Refining Guarded Specifications

Let us return to guarded specifications. We introduced these but, as we discussed in Sect. 4.5.1, the introduction and elimination rules alone do not quite tell us enough to ensure that we had captured our requirements: it was not immediately clear whether or not we had a more general notion than that, for example, given in [7]. We are now in a position to prove the following properties, which clear this up immediately:

$$\frac{z.P_1 \vdash z.P_0}{G \longrightarrow [\, P_0 \mid Q \,] \sqsupseteq G \longrightarrow [\, P_1 \mid Q \,]} \qquad \frac{z.Q_0 \vdash z.Q_1}{G \longrightarrow [\, P \mid Q_0 \,] \sqsupseteq G \longrightarrow [\, P \mid Q_1 \,]}$$

$$\frac{z.G_0 \vdash z.G_1}{G_0 \longrightarrow [\, P \mid Q \,] \sqsupseteq G_1 \longrightarrow [\, P \mid Q \,]}$$

There is, evidently, no constraint on the guard to remain weaker than the precondition in this case.

4.6.3 Monotonicity in the Schema Calculus

How does refinement interact with our basic schema calculus? Let us consider disjunction, as an example. We can prove the following:

$$\frac{U_0 \sqsupseteq U_2 \quad U_1 \sqsupseteq U_3}{U_0 \vee U_1 \sqsupseteq U_2 \vee U_3}$$

This shows that disjunction is fully *monotonic* with respect to refinement. As we mentioned in the introduction, νZ is *not* Z: disjunction in Z is certainly not monotonic with respect to refinement; so this is a very different schema calculus. And that means that νZ will be differently *used*: a textually similar Z specification will have a different meaning if understood in νZ (we will see this forcefully in Sect. 4.7.1). Indeed, all the basic schema operators of νZ are fully monotonic (negation is, of course, anti-monotonic in its single argument). As a consequence, new operations which are constructed from these will either be monotonic or anti-monotonic in their arguments; we will call argument positions which are monotonic (anti-monotonic) positive (negative). For example, consider:

$$X_0 \to X_1 =_{df} \neg X_0 \vee X_1$$

The first argument of implication is negative and the second positive.

Monotonicity is rather important: it is one thing to have a modular *language*, but this does not mean much unless we also have modular *reasoning*: suppose we have a large system S_0 within which there is a small component C_0 in a positive position; monotonicity means that if we plug a component C_1 that refines C_0 into S_0 to form S_1, then S_1 will be a refinement of S_0. Without monotonicity, we would not be able to conclude this.

4.6.4 Inequational Logic

Let us look at another refinement property for disjunction:

$$\overline{[\,P_0 \mid Q_0\,] \vee [\,P_1 \mid Q_1\,] \sqsupseteq [\,P_0 \wedge P_1 \mid Q_0 \vee Q_1\,]}$$

This kind of *axiom* (in future write such laws without the over-bar) is extremely important because it allows us to introduce structure into a development: using this axiom we can refine an atomic specification into one which is a disjunction.

Similarly, for conjunction, one such refinement axiom is this:

$$[\,P_0 \mid Q_0\,] \wedge [\,P_1 \mid Q_1\,] \sqsupseteq [\,P_0 \wedge P_1 \mid Q_0 \wedge Q_1\,]$$

Note that this is a *proper* refinement. That is to say, the refinement generally fails in the other direction. In other words, the left and right hand sides are not always equal (inter-refinable). The reader might recall that, in Z, these two expressions are indeed equal (and therefore, certainly inter-refinable). Once again, we see emerging differences between Z and νZ.

Finally, in this section, let us return to the conditional. Here is a refinement axiom:

$$\texttt{if } D \texttt{ then } [\,D \wedge P \mid Q\,] \texttt{ else } [\,\neg D \wedge P \mid Q\,] \sqsupseteq [\,P \mid Q\,]$$

If one uses the conditional operator as a program – rather than specification – structuring tool, this axiom amounts to a principle for *program development.*

4.7 Case Studies and Methodology III

Our illustrative examples have served two useful functions: they have enabled us to demonstrate some methodological principles and have provided a framework of more expressive theories within which more practical applications might be undertaken. We have also noted that the schema logic of νZ is different from Z. In this section, we will elaborate further, looking at a couple of simple examples in depth, both in Z and νZ, examining both the differences in application and methodology that they exemplify.

4.7.1 Robust Operations

Consider a system comprising a number of resources, some of which are in either a live or idle state (but not both at the same time). In Z:

$$\begin{array}{|l}
\hline
\textit{System} \\
\textit{live}, \textit{idle} : \mathbb{P}\,\textit{Resource} \\
\hline
\textit{live} \cap \textit{idle} = \{\} \\
\hline
\end{array}$$

In νZ:

$$\begin{array}{|l}
\hline
\textit{System} \\
\textit{live}, \textit{idle} \in \mathbb{P}\,\textit{Resource} \\
\hline
\textit{live} \cap \textit{idle} = \{\} \\
\hline
\end{array}$$

First, the syntactic difference is fairly trivial: Z involves types and sets explicitly in specifications (types are certain sets), whereas in νZ, types belong in the meta-language (we have not had to mention them in this chapter) and specifications only deal with sets. Second, this is νZ, so we are introducing a theory, not just a definition: the νZ state schema induces rules as special cases of those for atomic state schemas given in Sect. 4.2.1 above. Using these rules, those for schema sets, and simplifying, we obtain the following theory:

$$\frac{z \in s.\textit{live}, z \in s.\textit{idle} \vdash \textit{false}}{s \in \textit{System}}$$

for fresh z.

$$\frac{s \in \textit{System} \quad r \in s.\textit{live} \quad r \in s.\textit{idle}}{\textit{false}}$$

The simplifications here include noting that well-typing makes two premises of the introduction rule (such as $s.live \in \mathbb{P}\,Resource$) and two elimination rules (corresponding to those premises) redundant, because they are always true; see [5] for details concerning types.

Obvious operations are to force awake, or make dormant, a resource. In Z:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Wake}} \\ \hline
\Delta System \\
r? : Resource \\ \hline
r? \in idle \\
live' = live \cup \{r?\} \\
idle' = idle - \{r?\} \\ \hline
\end{array}$$

and:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sleep}} \\ \hline
\Delta System \\
r? : Resource \\ \hline
r? \in live \\
live' = live - \{r?\} \\
idle' = idle \cup \{r?\} \\ \hline
\end{array}$$

In νZ, these are expressed as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Wake}} \\ \hline
\Delta System \\
r? : Resource \\ \hline
r? \in idle \\ \hline
live' = live \cup \{r?\} \\
idle' = idle - \{r?\} \\ \hline
\end{array}$$

and:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sleep}} \\ \hline
\Delta System \\
r? : Resource \\ \hline
r? \in live \\ \hline
live' = live - \{r?\} \\
idle' = idle \cup \{r?\} \\ \hline
\end{array}$$

And again, because this is νZ, we are introducing a theory, not just a definition. The rules for *Wake* (those for *Sleep* are similar) are:

$$\frac{\begin{array}{ll} s_0 \in \mathit{System} & \\ s_1 \in \mathit{System} & \\ s_0.r? \in s_0.\mathit{idle}, y_0 \in s_0.\mathit{live} & \vdash y_0 \in s_1.\mathit{live} \\ s_0.r? \in s_0.\mathit{idle} & \vdash s_0.r? \in s_1.\mathit{live} \\ s_0.r? \in s_0.\mathit{idle}, y_1 \in s_1.\mathit{live}, y_1 \neq s_0.r? & \vdash y_1 \in s_0.\mathit{live} \\ s_0.r? \in s_0.\mathit{idle}, s_0.r? \in s_1.\mathit{idle} & \vdash \mathit{false} \\ s_0.r? \in s_0.\mathit{idle}, y_2 \in s_1.\mathit{idle} & \vdash y_2 \in s_0.\mathit{idle} \\ s_0.r? \in s_0.\mathit{idle}, y_3 \in s_0.\mathit{idle}, y_3 \neq s_0.r? & \vdash y_3 \in s_1.\mathit{idle} \end{array}}{s_0 \star s_1' \in \mathit{Wake}}$$

for fresh variables y_0 to y_3.

$$\frac{s_0 \star s_1' \in \mathit{Wake}}{s_0 \in \mathit{System}} \qquad \frac{s_0 \star s_1' \in \mathit{Wake}}{s_1 \in \mathit{System}}$$

$$\frac{s_0 \star s_1' \in \mathit{Wake} \quad s_0.r? \in s_0.\mathit{idle} \quad s_2 \in s_0.\mathit{live}}{s_2 \in s_1.\mathit{live}}$$

$$\frac{s_0 \star s_1' \in \mathit{Wake} \quad s_0.r? \in s_0.\mathit{idle}}{s_0.r? \in s_1.\mathit{live}}$$

$$\frac{s_0 \star s_1' \in \mathit{Wake} \quad s_0.r? \in s_0.\mathit{idle} \quad s_2 \in s_1.\mathit{live} \quad s_2 \neq s_0.r?}{s_2 \in s_0.\mathit{live}}$$

$$\frac{s_0 \star s_1' \in \mathit{Wake} \quad s_0.r? \in s_0.\mathit{idle} \quad s_0.r? \in s_1.\mathit{idle}}{\mathit{false}}$$

$$\frac{s_0 \star s_1' \in \mathit{Wake} \quad s_0.r? \in s_0.\mathit{idle} \quad s_2 \in s_1.\mathit{idle}}{s_2 \in s_0.\mathit{live}}$$

$$\frac{s_0 \star s_1' \in \mathit{Wake} \quad s_0.r? \in s_0.\mathit{idle} \quad s_2 \in s_0.\mathit{idle} \quad s_2 \neq s_0.r?}{s_2 \in s_1.\mathit{idle}}$$

The distinction between the Z and νZ specifications is non-trivial, both logically and semantically. Superficially one easily notes that the proposition (the syntactic precondition) $r? \in \mathit{idle}$ is conjunctive in the Z specification and implicational in the νZ specification. More subtly: consider a before state s_0 in which $r?$ binds to a resource that is not in the *idle* set. Since that fails to satisfy in the Z specification ($s_0.r? \in s_0.\mathit{idle}$ is *false*), there is no after state s_1 such that $s_0 \star s_1' \in \mathit{Wake}$; the specification is a partial relation and bindings outside the precondition are not associated with any after states. In the

νZ specification, the explicit precondition is false. But its function is implicational and the specification is trivially satisfied and hence $s_0 \star s_1' \in \mathit{Wake}$ for all s_1; this is a *chaotic* interpretation for before states which lie outside the precondition.[3] That is, we can easily prove the following theorem using the introduction rule:

$$\frac{s_0 \in \mathit{System} \quad s_1 \in \mathit{System} \quad s_0.r? \notin s_0.\mathit{idle}}{s_0 \star s_1' \in \mathit{Wake}}$$

We will return to this later.

Suppose we wish to mark the success of waking a resource. In Z, this is captured by:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{WakeSuccess}} \\ \hline
\mathit{msg}! : \mathit{String} \\ \hline
\mathit{msg}! = \text{“}\mathit{WakeSucceeds}\text{”} \\ \hline
\end{array}$$

In νZ, it is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{WakeSuccess}} \\ \hline
\mathit{msg}! \in \mathit{String} \\ \hline
\mathit{true} \\ \hline
\mathit{msg}! = \text{“}\mathit{WakeSucceeds}\text{”} \\ \hline
\end{array}$$

However, we will omit the precondition when it is *true*, so this is just:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{WakeSuccess}} \\ \hline
\mathit{msg}! \in \mathit{String} \\ \hline
\mathit{msg}! = \text{“}\mathit{WakeSucceeds}\text{”} \\ \hline
\end{array}$$

The theory for this is straightforward:

$$\frac{t.\mathit{msg}! = \text{“}\mathit{WakeSucceeds}\text{”}}{t \in \mathit{WakeSuccess}} \qquad \frac{t \in \mathit{WakeSuccess}}{t.\mathit{msg}! = \text{“}\mathit{WakeSucceeds}\text{”}}$$

[3] A reader with some knowledge of Z will know that this chaotic interpretation also makes an appearance for before states outside the precondition, but only in Z's theory of refinement. The difference between νZ and Z is not so much *whether* such before states are interpreted chaotically or not but rather at what point in the mathematical story this occurs. Specifically, it arises *before* schema operators are interpreted in νZ but *after* in Z. There are further remarks on this topic at the end of the section.

Then one can combine this with *Wake* to form a wake operation, which also marks success. In both Z and νZ, this is:

$$Wakes =_{df} Wake \wedge WakeSuccess$$

But how do we know this is right? In Z, one typically reduces the composite expression to an atomic schema and inspects it. This relies on the usual equational logic of Z. In other words, we use the Z equation:

$$[D_0 \mid P_0] \wedge [D_1 \mid P_1] = [D_0;\ D_1 \mid P_0 \wedge P_1]$$

to construct:

$$\begin{array}{|l}
\hline
\Delta System \\
r? : Resource \\
msg! : String \\
\hline
r? \in idle \\
live' = live \cup \{r?\} \\
idle' = idle - \{r?\} \\
msg! = \text{“}WakeSucceeds\text{”} \\
\hline
\end{array}$$

There are several observations to make here. First, it is methodologically dubious to reduce the expression to an atomic schema: the whole point of the schema calculus is that one need not work with single, large and unstructured specifications. Second, the approach will not work in νZ in any event, because the equation we used does not generally hold, as we saw in Sect. 4.6.4 (actually it does in this example; but that is hardly relevant). Indeed, the inequations that do generally hold in νZ are more usefully used to *increase* structuring: either in order to make specifications into expressions of simpler components (with simpler theories) or to introduce design or implementation detail. So, in νZ one makes the same definition but there is no need to manipulate the syntax. One's confidence in the definition is, as we argued in much earlier sections of the chapter, raised by what the theory expresses and indeed, the consequences of that theory. We immediately have the derived theory for *Wakes*:

$$\frac{t \in Wake \quad t \in WakeSuccess}{t \in Wakes}$$

$$\frac{t \in Wakes}{t \in WakeSuccess} \qquad \frac{t \in Wakes}{t \in Wake}$$

These, in themselves, provide some confidence that the specification of *Wakes* captures what is intended.

However, things are not always so straightforward. Let us turn to another possibility. It would be quite natural to cater for circumstances in which the operations cannot be performed. In Z, we might have this, for example:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{WakeFail}\;}} \\
\Xi \mathit{System} \\
r? : \mathit{Resource} \\
\mathit{msg}! : \mathit{String} \\ \hline
r? \notin \mathit{idle} \\
\mathit{msg}! = \text{“}\mathit{WakeFailed}\text{”} \\ \hline
\end{array}$$

In νZ, we have:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{WakeFail}\;}} \\
\Xi \mathit{System} \\
r? \in \mathit{Resource} \\
\mathit{msg}! \in \mathit{String} \\ \hline
r? \notin \mathit{idle} \\ \hline
\mathit{msg}! = \text{“}\mathit{WakeFailed}\text{”} \\ \hline
\end{array}$$

The theory for this is:

$$\frac{\begin{array}{l} s_0 \in \mathit{System} \\ s_1 \in \mathit{System} \\ s_0.r? \notin s_0.\mathit{idle} \vdash \; s_0 = s_1 \\ s_0.r? \notin s_0.\mathit{idle} \vdash \; s_1'.\mathit{msg}! = \text{“}\mathit{WakeFailed}\text{”} \end{array}}{s_0 \star s_1' \in \mathit{WakeFail}}$$

and:

$$\frac{s_0 \star s_1' \in \mathit{WakeFail}}{s_0 \in \mathit{System}} \qquad \frac{s_0 \star s_1' \in \mathit{WakeFail}}{s_1 \in \mathit{System}}$$

$$\frac{s_0 \star s_1' \in \mathit{WakeFail} \quad s_0.r? \notin s_0.\mathit{idle}}{s_1'.\mathit{msg}! = \text{“}\mathit{WakeFailed}\text{”}} \qquad \frac{s_0 \star s_1' \in \mathit{WakeFail}}{s_0 = s_1}$$

Now we can prove a similar theorem to that which we had for *Wake* (which obviously also holds for *Wakes*):

$$\frac{s_0 \in \mathit{System} \quad s_1 \in \mathit{System} \quad s_0.r? \in s_0.\mathit{idle}}{s_0 \star s_1' \in \mathit{WakeFail}}$$

The idea would now be to combine this with *Wakes* in order to produce a robust specification that covers the situation in which the resource to be woken may be chosen arbitrarily. In Z, this would be:

$$\mathit{WakeRobust} =_{df} \mathit{Wakes} \vee \mathit{WakeFail}$$

Again, how do we know that this captures the intended behaviour? In this case, we use another equation:

$$[D_0 \mid P_0] \vee [D_1 \mid P_1] = [D_0;\ D_1 \mid P_0 \vee P_1]$$

and we reduce *WakeRobust* to an atomic schema in order to inspect. We obtain this:

$$\begin{array}{|l}
\Delta System \\
r? \in Resource \\
msg! \in String \\
\hline
(r? \in idle \wedge live' = live \cup \{r?\} \wedge idle' = idle - \{r?\}) \vee \\
(r? \notin idle \wedge msg! = \text{“}WakeFailed\text{”} \wedge \theta System = \theta System') \\
\hline
\end{array}$$

This *appears* to be satisfactory. But it is probably easy to see that, as schema expressions become larger, it will become less and less clear that the atomic schema obtained by this normalisation does indeed capture the intended requirements. Of course, this is precisely what Z was supposed to avoid: large incomprehensible and unstructured specifications.

How do we proceed in νZ? Let us try the same approach:

$$WakeRobust =_{df} Wakes \vee WakeFail$$

It is easy to prove the following theorem regarding *WakeRobust*, for all $s_0, s_1 \in$ *System*:

$$\frac{}{s_0 \star s_1' \in WakeRobust}$$

that is, *all* state transitions belong to *WakeRobust*. The proof is easy, using the law of the excluded middle:

$$\dfrac{\dfrac{}{t.r? \in t.idle \vee t.r? \notin t.idle} \quad \dfrac{\dfrac{\dfrac{}{t.r? \notin t.idle}\,(1)}{t \in Wakes}}{t \in WakeRobust} \quad \dfrac{\dfrac{\dfrac{}{t.r? \in t.idle}\,(1)}{t \in WakeFail}}{t \in WakeRobust}}{t \in WakeRobust}\,(1)$$

Note that the proof uses the two theorems we proved earlier of *Wakes* and *WakeFail* and the two introduction rules for schema disjunction.

This is obviously nonsense and shows definitively that our definition of *WakeRobust* is wrong (that is: it does not capture the intended requirements). Two perfectly sensible properties of *Wakes* and *WakeFail* (neither of which was evident from the definition itself) permit the derivation of an unwanted property for the definition of *WakeRobust*. Does this mean that there is something wrong with νZ? Or wrong with Z? Not at all – it merely underlines a point that we have made before: νZ and Z have different schema calculi, and this will be reflected in the ways they will be used in practice.

For νZ, the counterexample is methodologically very significant. First, it demonstrates the importance of reasoning early in the description of systems for developing reliable specifications. Second, it shows how implicit properties of specifications can be derived (made explicit) and how these are subsequently useful in further development. Third, it shows how the modular structure of specifications can be preserved in coming to understand them better (exhibiting their implicit properties).

We can hardly finish here. What is the appropriate specification in νZ of the robust operation? This is it:

$$\mathit{WakeRobust} =_{df} \texttt{if } r? \in \mathit{idle} \texttt{ then } \mathit{Wakes} \texttt{ else } \mathit{WakeFail}$$

So, it does involve disjunction, but in a guarded context. The theory is:

$$\frac{s_0.r? \in s_0.\mathit{idle} \quad s_0 \star s_1' \in \mathit{Wakes}}{s_0 \star s_1' \in \mathit{WakeRobust}} \qquad \frac{s_0.r? \notin s_0.\mathit{idle} \quad s_0 \star s_1' \in \mathit{WakeFail}}{s_0 \star s_1' \in \mathit{WakeRobust}}$$

$$\frac{\begin{array}{l} s_0 \star s_1' \in \mathit{WakeRobust} \\ s_0.r? \in s_0.\mathit{idle}, s_0 \star s_1' \in \mathit{Wakes} \quad \vdash P \\ s_0.r? \notin s_0.\mathit{idle}, s_0 \star s_1' \in \mathit{WakeFail} \vdash P \end{array}}{P}$$

The reason for the difference is, of course, that in Z, before states that are outside the precondition are partial points in the relation, whereas in νZ they are interpreted chaotically. In Z, the chaotic interpretation appears only when the theory of refinement is imposed upon the partial relation semantics. The problem is that then both under-constrained *and* over-constrained specifications in Z are interpreted in the same way (in its theory of refinement): chaotically. In νZ, under-constrained specifications are interpreted chaotically, whilst over-constrained specifications are interpreted partially. As a result, it is always possible to refine contradictions in Z specifications to arbitrary behaviours, whereas in νZ, contradictions in a specification can *never* be refined to defined behaviours.

4.7.2 Methodology III

The rules for *WakeRobust* in νZ constitute good evidence that we have captured the intended behaviour and it is worth reflecting for a moment upon the relationship between the theory and the specification. In some sense, the model might now be thought of as entirely redundant: the theory captures (completely) the essence of the specification, and so far as reasoning is concerned, one now has little need to know the definition at all: just the properties that the specification possesses are crucial. From this perspective, one might begin to look at νZ as a property-based, rather than a model-based specification framework, the model serving only to guarantee that the theory is consistent (that there is indeed a model). Going further in this direction, one might in fact *begin* with the intended properties and only then seek to construct a model which satisfies them, and do so for one of two reasons. Firstly, as we have just remarked, in order to ensure consistency, but more interestingly, as a first stage in a program development. Secondly, one may (using refinement and related techniques) transform an abstract model into a concrete implementation.

4.7.3 Promotion

Promotion is an important Z structuring idiom in which changes to a system with a large (global) state are specified by means of operations over some small (local) component state. The advantages of this approach are several. First, there is modularity: one needs, in specifying the local operation, only to attend to a relevant part of the system. Second, the local state need not be precisely a part of the global state, we can specify how the local and global states are to be related. Third, one can develop the local operations entirely separately from the global state, so that there is a useful separation of concerns.[4]

The general pattern involved in promotion is that each local operation *Lop* over the local state *LS* is promoted, using a connecting schema Φ, to a corresponding global operation *Gop* over the global state *GS* by means of the following definition:

$$Gop =_{df} \exists \Delta LS \bullet Lop \wedge \Phi$$

In νZ, of course, we might define a new schema operation, for a given pair of state spaces and a relation on them, to capture this:

$$Promote\ X =_{df} \exists \Delta LS \bullet X \wedge \Phi$$

Naturally this leads to a theory for the promotion schema:

$$\frac{s_2 \in LS \quad s_3 \in LS \quad s_2 \star s_3' \in U \quad s_0 \star s_2 \star s_1' \star s_3' \in \Phi}{s_0 \star s_1' \in Promote\ U}$$

Suppose that the alphabet of *LS* is $\{\cdots \mathtt{z}_i \cdots\}$, then the elimination rule for lists of fresh variables w_i and y_i is:

$$\frac{\begin{array}{l} s_0 \star s_1' \in Promote\ U \\ \langle\!|\ \cdots \mathtt{z}_i \Rrightarrow y_i \cdots\ |\!\rangle \in LS \\ \langle\!|\ \cdots \mathtt{z}_i \Rrightarrow w_i \cdots\ |\!\rangle \in LS \\ \langle\!|\ \cdots \mathtt{z}_i \Rrightarrow y_i \cdots\ |\!\rangle \star \langle\!|\ \cdots \mathtt{z}_i' \Rrightarrow w_i \cdots\ |\!\rangle \in U, \\ s_0 \star s_1' \star \langle\!|\ \cdots \mathtt{z}_i \Rrightarrow y_i \cdots\ |\!\rangle \star \langle\!|\ \cdots \mathtt{z}_i' \Rrightarrow w_i \cdots\ |\!\rangle \in \Phi\ \vdash P \end{array}}{P}$$

[4] Although we will not be able to illustrate it in this chapter, there is, in νZ, a fourth benefit: one may derive an implementation of the local operation separately from and without any knowledge of the global state.

We finish this section with a very simple example of promotion to illustrate the approach. Consider the following elementary local state:

$$\begin{array}{|l}
\hline \textit{LS} \\
v \in \mathbb{N} \\
\hline
\end{array}$$

with a simple theory (all well-typed states are in LS):

$$\frac{}{s \in LS}$$

and a local operation:

$$\begin{array}{|l}
\hline \textit{Inc} \\
\Delta LS \\
\hline
v' = v + 1 \\
\hline
\end{array}$$

with theory:

$$\frac{s_1.v = s_0.v + 1}{s_0 \star s_1' \in \mathit{Inc}}$$

$$\frac{s_0 \star s_1' \in \mathit{Inc}}{s_1.v = s_0.v + 1}$$

Here, references to LS disappear because there are no constraints on values in the local state. The same is true for the global state GS; this leads to simpler rules in the theories for Φ and Ginc below.

The global state is a pair:

$$\begin{array}{|l}
\hline \textit{GS} \\
p \in \mathbb{N} \times \mathbb{N} \\
\hline
\end{array}$$

with the theory:

$$\frac{}{s \in GS}$$

The idea is to promote Inc so that it operates over the global state, affecting only the first component. Note that selecting from pair is given by $(x, y).1 = x$ and $(x, y).2 = y$. The next stage is to define the linking promotion schema:

$$\begin{array}{|l}
\hline \varPhi \\
\Delta LS \\
\Delta GS \\
\hline
v = p.1 \\
\hline
v' = p'.1 \\
p.2 = p'.2 \\
\hline
\end{array}$$

with the theory:

$$\frac{\begin{array}{c} s_0.v = s_0.p.1 \;\vdash\; s_1.v = s_1.p.1 \\ s_0.v = s_0.p.1 \;\vdash\; s_0.p.2 = s_1.p.2 \end{array}}{s_0 \star s_1' \in \Phi}$$

and:

$$\frac{s_0 \star s_1' \in \Phi \quad s_0.v = s_0.p.1}{s_1.v = s_1.p.1} \qquad \frac{s_0 \star s_1' \in \Phi \quad s_0.v = s_0.p.1}{s_0.p.2 = s_1.p.2}$$

We now define the global operation:

$$GInc =_{df} Promote\ Inc$$

And the theory drops out immediately:

$$\frac{s_2 \star s_3' \in Inc \quad s_0 \star s_2 \star s_1' \star s_3' \in \Phi}{s_0 \star s_1' \in GInc}$$

and for fresh variables w_0 and w_1:

$$\frac{s_0 \star s_1' \in GInc \quad \langle\!| \, v \Rrightarrow w_0, v' \Rrightarrow w_1 \, |\!\rangle \in U, s_0 \star s_1' \star \langle\!| \, v \Rrightarrow w_0, v' \Rrightarrow w_1 \, |\!\rangle \in \Phi \;\vdash P}{P}$$

These simplify and it is easy to prove the following:

$$\frac{u_2 = u_0 + 1 \quad u_3 = u_1}{\langle\!| \, p \Rrightarrow (u_0, u_1), p' \Rrightarrow (u_2, u_3) \, |\!\rangle \in GInc}$$

$$\frac{t \in GInc}{t.p'.1 = t.p.1 + 1} \qquad \frac{t \in GInc}{t.p'.2 = t.p.2}$$

Note that these properties *no longer make any reference to observations of the local state or their values*, as is entirely appropriate. They not only provide means for further reasoning about the operation *GInc*, but they also provide evidence that the its specification captures the intended requirements.

4.8 Conclusions and Future Work

We have introduced the specification language νZ with an emphasis on applications and methodology. Although νZ has a number of advantages over Z, its most significant precursor, the main distinction we would like to re-emphasise, is that its specifications are theories and not (just) definitions.

We began with the observation (see [3] for strong arguments) that formal methods contribute most effectively to practical software engineering development when they are used *as early as possible*. This means that an ability to reason about specifications themselves, to explore their properties and consequences, is even more crucial than being able to use them for the correctness preserving development of implementations. νZ in fact permits and encourages both of these.

We have, however, avoided any significant treatment of program development in this chapter, although we have given some hints about the topic in our section on refinement; more detail can be found in [5]. For example, we have not discussed the *recursive schema operator* $\mu X \bullet U(X)$ at all: these are enormously powerful, permitting the definition of such operators as:

$$\texttt{while}\ D\ \texttt{do}\ X =_{df} \mu Y \bullet D \longrightarrow X \fatsemi Y \vee \neg D \longrightarrow \Xi$$

Integrating program development in νZ is one of its major motivations. To do this, one uses νZ to *specify* a programming language (which of course induces an associated programming logic). This is another sense in which νZ goes beyond Z, and why we refer to νZ as a *wide-spectrum logic.*

In both the areas of specification and program development, much pragmatic work is being undertaken; for example, Kajtazi has explored program development with positive results [6], but more remains to be done.

Acknowledgements

This chapter is based on a presentation given as part of a series of invited talks organised by the Formal Aspects of Computer Science specialist group of the British Computer Society on 8 November 2005. I would like to thank the FACS organisers, Paul Boca, Jonathan Bowen and Jawed Siddiqi, for the opportunity to present this material, and the audience for their patience, astute questions and comments. In addition, this work has benefited enormously from conversations with, and criticisms from, Besnik Kajtazi, Moshe Deutsch and Steve Reeves. Finally, the style and emphasis of this chapter were influenced significantly by the presentation Anthony Hall subsequently gave in the same seminar series; I have had occasion to mention that in what precedes.

References

1. D.Bert, J. P. Bowen, S. King, and M. Waldén, editors. *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, *Lecture Notes in Computer Science* vol. 2651. Springer, 2003.
2. J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors. *ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29–September 2, 2000, Proceedings*, *Lecture Notes in Computer Science* vol. 1878. Springer, 2000.
3. A. Hall. Realising the benefits of formal methods. *Journal of Universal Computer Science*, 13(5):669–678, 2007.
4. M. C. Henson and S. Reeves. Revising Z: II - logical development. *Formal Aspects of Computing*, 11(4):381–401, 1999.

5. M. C. Henson, M. Deutsch, and B. Kajtazi. The specification language νZ. *Formal Aspects of Computing*, 18(3):364–395, 2006.
6. B. Kajtazi. *Specification Refinement and Program Development in* νZ. PhD thesis, Department of Computer Science, University of Essex, UK, 2008.
7. R. Miarka, E. A. Boiten, and J. Derrick. Guards, Preconditions, and Refinement in Z. In Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, pp. 286–303, 2000.
8. S. Valentine, I. Toyn, S. Stepney, and S. King. Type Constrained Generics in Z. In Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, pp. 250–263, 2000.
9. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, NewYork, 1996.

5

The Computer Ate My Vote

Peter Y.A. Ryan

University of Luxembourg, Luxembourg
peter.ryan@uni.lu

Abstract Public confidence in voting technologies has been badly shaken over the past years by, amongst other events, the problems with the 2000 and 2004 US presidential elections and the 2007 French presidential election. Serious vulnerabilities have been exposed and documented in all the existing electronic voting systems. Many of these systems use proprietary, protected code and the voters and election officials are expected to take assurances of the suppliers and certifiers on trust.

In the face of this, many activists argue that all voting systems employing information technology in place of humans to process the votes must be flawed and we should return to pen-and-paper along with hand counting. The critiques of existing voting technologies are undoubtedly well-founded, but to infer from this that all technology must be flawed is an elementary error of logic. In this chapter, I present an alternative response and describe schemes that derive their trustworthiness from maximal transparency and auditability.

Designing voting systems that provide high levels of assurance of accuracy and ballot secrecy with minimal trust assumptions is an immensely challenging one. The requirements of accuracy and auditability are in direct conflict with those of ballot secrecy. A voting system must deliver high assurance of accuracy and privacy in a highly hostile environment: the system may try to cheat the voters, voters may try to circumvent the system, officials may try to manipulate the outcome and coercers may attempt to influence voters. Furthermore, we must recognise that this is not a purely technical problem: a technically perfect solution that is not usable or does not command the confidence of the voters is not a viable solution.

Recently significant progress has been made and a number of schemes developed that provide verifiability of the election. These seek to provide end-to-end verifiability of the outcome, i.e., the accuracy of the outcome is independent of the code or hardware that implements the ballot processing. The assurance is derived from maximal transparency and auditability. Voters are provided with mechanisms to check that their vote is accurately included in the final tally, all the while maintaining ballot secrecy. Thus, the assurance depends ultimately on the voters rather than the probity of election officials, suppliers of voting systems, etc.

In this chapter, I describe the requirements for voting systems, the required cryptographic building blocks and a variety of threats they have to deal with. I then describe the Prêt à Voter scheme, a particularly voter-friendly example of a

voter-verifiable, high assurance scheme. I also describe a number of enhancements to the basic scheme that are designed to counter those threats to which the basic version of Prêt à Voter is potentially vulnerable.

5.1 Introduction

From its very inception, democracy was recognised as a fragile and precious attribute of civilisation. The Ancient Greeks, prompted by the realisation that inevitably people would attempt to corrupt the outcome of elections, developed primitive but ingenious technological devices to shift the trust away from officials to mechanical devices – see for example "Ancient Greek Gadgets and Machines" by Brumbaugh [20].

In more recent times, in the US, problems with paper ballots prompted the deployment of various voting technologies, starting over a century ago with lever machines in 1897, followed by punch cards, optical scanners, touch screens, etc. The excellent book by Gumbel, "Steal this Vote" [21], documents an extraordinary litany of ingenious techniques to corrupt every voting system that has ever been deployed in the US. Recent reports, like the Johns Hopkinss report [28] and the Princeton report [16], demonstrate how vulnerable a poorly designed electronic voting system is to virtually undetectable corruption. Voter confidence in the US, and beyond, has been badly undermined as a result.

In contrast, many countries, including the UK, have stuck to using the time-honoured paper ballots and hand counting. There seems to be little evidence of significant manipulation and it is generally assumed that the process is reasonably trustworthy. In particular, it seems reasonable to assume that it would be very difficult to pull off a significant, undetected manipulation of the count.

Given the apparent trustworthiness of paper ballots, one might reasonably ask: why move to more sophisticated voting technologies? This reaction has much to commend it and has a significant number of rather vocal advocates. Indeed, the US experience certainly shows that careless introduction of digital technology can be disastrous. However, this should not be taken as proof that all voting technology must be flawed.

There are, in fact, a number of reasons to explore novel voting technologies. Firstly, I would like to put aside the oft quoted, but unconvincing argument that more convenient methods of voting, e.g. internet, will improve voter turnout. Whether or not this is true, I will not consider it as a valid reason for exploring more advanced voting technologies.

More compellingly, it is argued that the use of voting technologies may help guide voters through the more complex, multi-race ballots presented to voters, as with many US elections. Appropriate use of technology may also enable voters with disabilities to vote in privacy. Electronic counting could potentially help with speed, accuracy and efficiency.

In my view, the most compelling reason to explore novel voting systems, aside from the intellectual challenges, is the potential to create schemes that provide greater levels of assurance of accuracy and ballot secrecy through higher levels of transparency: removing the need to trust suppliers, software or officials.

Schemes such as those described below place the trust in the hands of the voters themselves and so could provide a sound basis on which to restore faith in the mechanisms of democracy. These schemes can be thought of as striving to place democracy on the firm foundations of mathematics. The Ancient Greeks would have looked favourably on such an endeavour!

We should be careful to draw a distinction between the use of digital technology in the context of *supervised* voting, i.e., with the enforced isolation of a booth in a polling station as opposed to remote voting, e.g. voting over the internet. In this chapter, I concentrate on supervised voting. This is already challenging enough without further complicating matters by having to address coercion problems associated with remote voting.

The material presented here is based on previously published material in [10, 35, 36, 39].

5.2 The Challenge

The challenge is to design systems that provide high assurance of the accuracy of the outcome, whilst at the same time guaranteeing the secrecy of ballots. The problem is that these two requirements are in conflict. Each taken in isolation would be trivial to achieve. In the absence of the secrecy requirement, a simple show of hands guarantees accuracy. Equally, if accuracy is not required, then secrecy is trivial: a constant function achieves this, i.e., the result is totally independent of the inputs.

Similarly, if we are prepared to place complete trust in the processes that collect and count the votes, then again there is no real problem. This, in essence, is how the current UK system works and, whilst there seem to be good grounds to believe that such trust is well-placed for the time being, there are contexts in which such trust would be wholly inappropriate. Who is to say that the UK will remain as benign a political environment? It would seem prudent, therefore, to explore voting systems that do not necessitate such trust.

The goal then is to provide assurance of accuracy and secrecy without having to place any trust in the devices, processes and officials that conduct the election. Put differently, we strive to design systems that will be resilient to insider threats as well as to outsider threats.

A further complication of secret ballot voting systems, which distinguishes them from conventional dependable systems, is that there is no extrinsic way to characterise the correctness of the outcome. By definition, there is no god-like view that can determine the correct outcome and it is, thus, possible for an election system to fail in a way that is not manifest. This is in contrast with, say, an algorithm for computing square roots, or for that matter, avionics software. The correctness of the outcome is clearly defined and independently verifiable, or turning this around, failures of such systems are manifest. Failures of voting software could go unnoticed and, even if suspected, may be impossible to demonstrate.

As if all this were not challenging enough, we must also take account of the socio-technical aspects of the problem: our systems must be transparent and simple enough to provide a reasonable degree of usability, public understanding and trust. A technically perfect solution that fails to gain the confidence of the electorate is not a viable solution. Furthermore, we must take account of the surrounding socio-technical systems that maintain the electoral register, authenticate voters, officials and scrutineers who supervise and observe the process, etc.

To a cryptographer, a voting system is a highly hostile environment: the system is trying to cheat the voters, the voters are trying to circumvent the system, officials are trying to manipulate the system, coercers are attempting to influence voters and voters trying to cheat coercers. Curiously, the last of these is one that we want to enable! Of course, all this is very much a worst case analysis, but if we can make our systems resistant against such adversaries, then they should also be robust in less hostile environments. Like Hollywood producers, cryptographers like their adversaries to be imbued with unlimited malevolence and cunning.

5.3 Assumptions

Voting systems are large socio-technical systems comprising not only various technical components but voting officials, voters, campaigners, potential coercers, etc. The process of an election goes far beyond merely collecting and counting votes and has social, political, legal and psychological aspects. For the purposes of this chapter, we will make a number of rather sweeping assumptions and we will draw rather precise boundaries around the portion of the voting system that will be discussed. In practice, of course, all these assumptions have to be closely examined and the scheme evaluated in the context of the wider socio-technical system.

We will confine our attention to the more technical aspects of the problem: the processes and mechanisms that capture and count the votes. We will assume, for example, that an accurate register of eligible voters is maintained and that suitable access to this database is ensured throughout the period of voting. We will assume that suitable mechanisms are in place to authenticate voters and ensure that they do not vote more than once. We will also assume for the most part that the cryptographic algorithms used are sufficiently strong.

5.4 Voting System Requirements

At the most abstract level, we require that elections be “free and fair”. Freeness can be taken to mean that, throughout the period of the election, legitimate members of the electorate are able to express their intent without hindrance or undue influence. Quite what constitutes "undue influence" is a rather delicate matter. Some would contend that political manipulation of the media, bandying about of vacuous promises and so on constitute undue influence. We will leave such matters to the political scientists and concentrate on eliminating the rather crude but better defined threats of vote buying and coercion.

Fairness is taken to mean that every voter is able to cast at most one vote and that all votes cast are accurately counted and reflected in the final tally. These rather vague statements are mapped down to more precise requirements on the various components of the system.

There is no universal consensus as to voting system requirements and in any case, they will vary from application to application and according to jurisdiction. Here, we informally describe the commonly accepted requirements of the vote capture and counting components of the system.

The primary goal of a voting system is integrity (a.k.a. accuracy): all legitimately cast votes should be accurately included in the count. Furthermore, accuracy, like justice, should not only be done but also be seen to be done. Thus, we will strive for *verifiability*, i.e., mechanisms that demonstrate that the count is accurate. Often, cryptographic schemes provide unconditional integrity, that is, they provide guarantees of integrity that do not depend on computational assumptions. In other words, integrity is guaranteed even against adversaries with unbounded computational power.

We also require that a voter's choice is kept secret, often termed *ballot secrecy* or *ballot privacy*. This requirement stems from the need to avoid threats of vote buying or coercion. Some forms of voting call for accountability, for example, parliamentary voting, and hence secrecy is not required. In fact, absolute ballot secrecy is not required or provided by the UK voting system. Rather, UK electoral law requires that it be possible for the authorities, with appropriate court orders, to trace ballots back to the (claimed) identity of the person who cast the ballot. It is still expected that no one other than appropriate authorities be able to establish how a particular voter voted.[1] We will take ballot secrecy as a fundamental requirement in what follows.

A rather novel requirement, not feasible in conventional systems, is that of *voter-verifiability*. Voters are able to confirm that their vote is accurately included in the count and, if not, to prove this to a judge. At the same time, the voter is not able to prove to a third party which way they actually voted. At first glance this seems impossible, but the schemes that we describe below do realise this requirement. Modern cryptography regularly makes the seemingly impossible quite routine!

[1] Cryptographic schemes can support the requirement to reveal links between receipts and votes, but this can only be done with the cooperation of a predefined threshold number of servers. These servers could be controlled by independent authorities and organisations. Thus, the checks and balances preventing abuses of such a capability are far more transparent and accountable than at present.

Besides the above technical requirements, voting systems must also be cost-efficient, easy for voters to use and sufficiently simple to gain a sufficient degree of public understanding and trust. Discussion of these requirements is outside the scope of this chapter, but we remark that the systems described here are aimed at being as conceptually simple as possible.

5.5 Sources of Assurance

It is important to understand the contrasting sources of confidence in technology. The most naive approach is simply to trust the technology or process, perhaps largely on the basis of past performance. In some contexts, this may be reasonable e.g., using an ATM, where fraud is detectable and forms of insurance and redress are available. Voting is definitely not in this category: elections are essentially one-off events, fraud may go undetected and the consequences are potentially global.

Assurance of “correct” behaviour of any system may be derived in various ways. It might be based on (claims of) prior evaluation, verification and testing of the system. Such analysis seeks to demonstrate that all possible system executions will satisfy the requirements. On the other hand, assurance might be derived from run-time monitoring of the system as it executes and end-to-end checking of outputs, ensuring that any departure from correct behaviour is prevented.

Both approaches have a useful role to play in certain contexts and typically both must be employed, but the emphasis may vary. For systems in which it may be hard to recover from a malfunction and you really want confidence about future performance, e.g. an avionics system, emphasis must be placed on the former. The drawback is that complete and correct analysis is extremely difficult to achieve and, even if it were achieved, system degradation, upgrades or alteration of the code, etc. may all serve to invalidate any analysis. Such assurance is, thus, very fragile.

End-to-end checking is independent of the software and hardware implementation and so is more robust: as long as we have a precise definition of “correct” behaviour and can implement a suitable monitoring mechanism, we can be confident that any errors will be detected and rectified. We are, thus, no longer prey to the problems of incomplete analysis or testing, or the mutability of software, etc. In the case of voting systems, we argue that assurance of integrity must be end-to-end. The correctness of the outcome should be mathematically verifiable, rather as the correctness of a square root algorithm is independently verifiable. Thus, we should verify the election, not the system.

This is not to say that evaluation and testing of the system should be ignored, but just that the accuracy of the outcome must not be contingent on the correctness, coverage and continuing validity of such evaluation. Naturally it is essential that an election run smoothly and not be plagued by malfunctions and technology glitches, so careful testing and evaluation will also be essential.

5.5.1 Assurance of Privacy

The discussion above applies to the sources of assurance of integrity. Secrecy and privacy properties are of quite a different nature to integrity properties. Integrity is a *safety* property, i.e.: can be characterised by defining what are acceptable system behaviours. Secrecy properties, by contrast, are usually defined in terms of the entire set of possible behaviours rather than individual behaviours. Thus, for example, the secrecy provided by a stream cipher depends on it being effectively indistinguishable to an adversary from a device able to generate all possible random streams with equal probability (i.e.: a one-time-pad). This is clearly not a property of any given behaviour but of the ensemble of possible system behaviours.

Consequently, it is typically not possible to detect failures with respect to a secrecy property by monitoring an individual system execution, rather as it is not sensible to ask if a given bit sequence is random. It also tends to be much harder to recover from secrecy failures ("it is hard to get the toothpaste back in the tube", as one US judge phrased it). Consequently, for secrecy, we need to place more emphasis on prior verification than we do for integrity.

5.6 Verifiable Voting Schemes

Many cryptographic schemes that seek to provide high levels of assurance of accuracy and secrecy have been proposed. Typically these strive to minimise the need to place trust in the implementations and seek assurance through maximal transparency. In accordance with the principle of "no security through obscurity", the details of these schemes are laid bare to universal scrutiny, so that any flaws in the design can be detected before deployment. Furthermore, the integrity of the election rests now on the validity of the mathematical arguments, rather than on mutable implementations.

This is a key and subtle point: the goal of a verifiable scheme is to ensure that any malfunction or corruption of the implementation is detectable. All the computations performed during the auditing phase are against publicly known functions and hence independently verifiable by anyone. Furthermore, such verification is performed on data that is frozen and committed rather than trying to observe an ephemeral process.

5.7 Related Work

There is a large and rapidly growing literature on cryptographic voting schemes and it is not appropriate to attempt to survey it all here. We will just mention the most closely related literature, concentrating on schemes designed for the supervised, rather than remote context. A more complete survey can be found in [2], and for anonymity mechanisms, mixes, etc. see the anonymity bibliography [1].

The first suggestion that cryptographic techniques could be applied to voting systems appears to be Chaum's 1981 paper on anonymising mixes [7]. Benaloh and Tunistra [6] introduce the notion of coercion-resistance along with a scheme using homomorphic tabulation that satisfies it. Later, Chaum [8] and Neff [29] introduced schemes that could be regarded as more practical than previous schemes. The original Prêt à Voter scheme [33,34] was inspired by the Chaum scheme, replacing the visual cryptography by the candidate permutation concept. Chaum has subsequently adopted this concept in a new scheme called PunchScan [18]. Recently, Rivest has proposed *ThreeBallot* [32] that provides voter-verifiability without using cryptography.

5.8 Cryptographic Primitives

Firstly, we introduce some cryptographic primitives that will be used later. In the interests of space and not bogging the reader down in unnecessary technical detail, we give very superficial descriptions, enough, we hope, to make their role clear. For more detailed descriptions consult any book on modern cryptography, e.g., Stinson [41]. Note also that this overview is not intended to be exhaustive: we stick to those required for Prêt à Voter. We assume familiarity with the basics of number theory. No offence will be taken if you are familiar with modern cryptography and choose to skip this section.

5.8.1 Public Key Cryptography

Some thirty years ago, the rather shadowy discipline of cryptography witnessed a major revolution: in 1976, Diffie and Hellman published a paper [14] that, for the first time in the open literature, proposed the possibility of public key cryptography. Arguably, this paper did for cryptography what Einstein's 1905 special relativity paper did for theoretical physics. Prior to 1976, it was implicitly assumed that secret communication could only be achieved between parties who already shared secret material. Diffie and Hellman overturned this assumption by suggesting that it should be possible for complete strangers who did not share any prior secrets to communicate secretly over open channels.

At first glance this seems absurd. To realise it requires the concept of one-way functions: functions that are easy to compute in one direction but intractable to compute in the other. One such function is that of multiplication: multiplying a pair of given numbers is straightforward, yet, given a number, finding its prime factors is in general extremely difficult.

Using the concept of one-way functions, it is possible to implement encryption algorithms with the following features: Anne creates a pair of keys, one that is made public, the other one she keeps secret. The public key can be used by anyone to encrypt a message for Anne, but decryption can only be performed with knowledge of the secret key. Thus, Anne, and only Anne, can decrypt such messages. From a “classical” cryptography perspective, this seems impossible: encryption and decryption keys are closely related, often identical. The key point is that knowledge of the public encryption key does not entail knowledge of the secret key.

The ideas of public key were, in fact, anticipated in secret at the Government Communications Headquarters (GCHQ). In 1970, James Ellis came up with the idea of *non-secret encryption* and in 1973, Clifford Cocks came up with an implementation of the concept that is essentially the same as the RSA algorithm we describe below. None of this was available in the open literature until recently, so it is fair to say that these concepts were independently re-invented.

5.8.2 RSA Encryption

The first realisation of the public key concept (in the open literature) is that proposed by Rivest, Shamir and Adleman [4]. The security of the scheme rests on the assumed difficulty of factorising. To set up her RSA keys, Anne identifies two large prime numbers, p and q. By “large” here we typically mean of the order of a hundred digits. Given that factorisation is hard, it would appear to be hard to find large primes. However, efficient algorithms exist for finding primes that do not involve factorising. Typically such algorithms do not yield primes with certainty but with arbitrarily high probabilities.[2]

Let $n := p \cdot q$. Now Anne chooses an encryption exponent e such that $gcd(e, n) = 1$, i.e., e must be co-prime to n. Now she finds d such that:

$$e \cdot d = 1 \pmod{\phi(n)}$$

where $\phi(n)$ is Euler's totient function; in this case, $\phi(n) = (p-1)\cdot(q-1)$. This computation can be performed very efficiently using the extended Euclidean algorithm. Now Anne can make n and e public whilst she keeps d (and p, q and $\phi(n)$) secret. Now Bob can encrypt a message m, which is suitably encoded as a number in Z_n^*:

$$c := m^e \pmod{n}$$

Anne can decrypt this by computing:

$$c^d = m^{e \cdot d} = m \pmod{n}$$

[2] A simple example of such an algorithm is based on Fermat's Little Theorem: if p is prime, then $\forall\, a \in Z_p^*, a^{p-1} = 1 \pmod{p}$. Thus, for a putative prime p, we choose a set of as at random and check whether Fermat's congruence holds for all of them. If it does, then we have, with high probability, identified a prime p.

To see why this works, note that, by construction, $e \cdot d = r \cdot \phi(n) + 1$ for some $r \in Z_n$. So, by Euler's generalisation of Fermat's Little Theorem:

$$\forall\, a \in Z_n^* : a^{\phi(n)} = 1 \pmod{n}$$

where Z_n^* is the set of residue classes modulo n with multiplicative inverse, i.e., $Z_n^* = \{a \in Z_n \mid gcd(a, n) = 1\}$.

It follows that:

$$m^{e \cdot d} = m^{\phi(n) \cdot r + 1} = m^{\phi(n) \cdot r} \cdot m^1 = 1 \cdot m = m \pmod{n}$$

as required.

Computing exponents in a finite field can be done very efficiently using repeated squaring and multiplication. Computing d from e is straightforward with the knowledge of the factorisation of n, and hence of $\phi(n)$. Without knowing the factorisation, there is no known efficient way to find d short of essentially exhaustive search. No efficient algorithm is known to factorise composite numbers that are the products of such large primes. Hence, it is believed that it is intractable to compute d given only n and e, i.e., without knowledge of the factorisation of n. Anne, by contrast, started with the primes p and q and so is not faced with the problem of discovering them by factorising n. So, as long as Anne guards the secret value d and the factorisation of n, she alone can decrypt messages encrypted using her public key. Anyone knowing n and e can encrypt messages for Anne.

5.8.3 ElGamal Encryption

RSA, as described above, is deterministic; encrypting the same plaintext twice yields the same ciphertext. We now describe a couple of randomising algorithms for which repeatedly encrypting a given plaintext will yield different ciphertexts each time.

The first algorithm is due to ElGamal [15]. Anne finds a large prime p and a primitive element α of the multiplicative group Z_p^* (α is said to be primitive if every element of Z_p^* can be expressed as a power of α). Anne chooses a random k from Z_p and computes:

$$\beta := \alpha^k \pmod{p}$$

The public keys are p, α and β, k is kept secret. Encryption of m yields a pair of terms computed thus:

$$c := (y_1, y_2) := (\alpha^r, m \cdot \beta^r) \pmod{p}$$

where r is chosen at random from Z_p. Anne, with the knowledge of k, is able to decrypt as follows:

$$m = y_2 / y_1^k \pmod{p}$$

The security of ElGamal rests on the presumed difficulty of taking discrete logs in a finite field. We remarked earlier that taking exponents is perfectly tractable, but the inverse operation of taking discrete logs is believed to be intractable. Thus, recovering the secret k exponent from knowledge of p, α and β is thought to be intractable.

A randomising algorithm like ElGamal allows the possibility of re-encryption: anyone who knows the public key can re-randomise the original encryption with a new random value r':

$$(y_1', y_2') := (\alpha^{r'} \cdot y_1, \beta^{r'} \cdot y_2)$$

which gives:

$$(y_1', y_2') := (\alpha^{r'+r}, \beta^{r'+r} \cdot m)$$

Clearly, this is equivalent to simply encrypting m with the randomisation $r + r'$ and decryption is performed exactly as before. We will see the utility of re-encryption when we come to describe anonymising mixes. Note that, crucially, the device performing the re-encryption does not use any secret keys and at no point in the re-encryption process is the plaintext revealed.

5.8.4 Paillier Encryption

Paillier is another randomising algorithm that, due to its homomorphic properties, is very handy in voting applications [30]. Key generation proceeds as follows: first generate an RSA integer $n = p \cdot q$ with p and q large primes and then compute the Carmichael function of n: $\lambda := lcm(p-1, q-1)$. Find a generator g of $Z^*_{n^2}$ such that $g = 1 \pmod{n}$. The public key (n, g) is published whilst λ forms the secret key.

The encryption of a message $m \in Z_n$ is computed as:

$$c = \mathcal{E}_{\mathcal{P}}(m, r) = g^m r^n \pmod{n^2}$$

where $\mathcal{E}_P(x;\ r)$ denotes the encryption of x and randomisation r. r is a freshly generated random value drawn from Z^*_n.

Decryption is given by:

$$m = \frac{L(c^\lambda (\mathrm{mod}\, n^2))}{L(g^\lambda (\mathrm{mod}\, n^2))} \pmod{n}$$

where, for convenience, we have defined $L(x) := (x - 1)/n$.

We will not go into exactly why this rather surprising formula works. The key ingredient is the fact that:

$$\forall\, a \in Z^*_{n^2} : a^{n.\lambda(n)} = 1 \pmod{n^2}$$

Judicious use of the binomial theorem to expand g, expressed as $1 + k \cdot n$, and noting that terms with n^2 or higher order in n vanish taken modulo n^2 serve to move the plaintext m term from the exponent to a linear term.

Due to the way that the plaintext is carried in the exponent, the Paillier algorithm enjoys the homomorphic property:

$$\mathcal{E}_P(a;\ r) \times \mathcal{E}_P(b;\ s) = \mathcal{E}_P(a + b;\ r \times s)$$

That is, the product of the encryption of two plaintexts equals the encryption of the sum of the plaintexts.

5.8.5 Threshold Cryptography

It is often important not to have to depend on a single entity to perform cryptographic operations, such as decryption and signing. This prompts the development of techniques and algorithms to distribute the knowledge of the secret key amongst a set of entities Φ in such a way that only a quorum of Φ can perform the operation. An (m, k) threshold scheme, for example, allows k or more entities from a set of size m, $(k < m)$, to collaborate to perform the operation. Any smaller set of size $< k$ will be unable to perform the operation or obtain any information about the secret key. The classic example of such a scheme is the secret sharing scheme due to Shamir [40].

5.8.6 Anonymising Mixes

Anonymising mixes were first proposed by Chaum [7] and play a key role in many voting schemes. They come in two flavours: decryption and re-encryption, but hybrids are also possible.

In decryption mixes, a deterministic algorithm such as RSA is used. The plaintexts are encrypted in layers under the public keys of a set of mix servers. The batch of encrypted terms is passed to the first server that strips off the outer layer of encryption and performs a secret shuffle of the resulting terms. The resulting, partially decrypted, shuffled terms are passed on to the next server that strips off a further layer, shuffles the resulting terms and passes this along to the next. Thus, the batch of encrypted terms is put through a series of such mixes that progressively decrypt and shuffle. At the end of these mixes, the raw decrypted votes pop out but with any link to the original terms obliterated by the multiple shuffles.

For re-encryption mixes, a randomising algorithm, such as ElGamal or Paillier, is used. Instead of striping off layers of encryption at each step of the mix, the mix servers re-randomise the encryptions. The mix servers do not hold any secret keys, as re-randomisation can be performed by any entity that knows the public key. A sequence of such mixes can then be followed by a (threshold) decryption by servers that hold the appropriate secret keys.

The fact that all the terms passing through a mix undergo a transformation, either decryption or re-encryption, at each stage ensures that they cannot be simply traced through the mix by pattern matching, as long as at least one of the servers remains honest.

5.8.7 Homomorphic Tabulation

Anonymising mixes are well-suited to taking a batch of encrypted ballots and outputting the decrypted ballots in a shuffled order. An alternative way to perform an anonymising tabulation of a batch of encrypted ballots is to use *homomorphic tabulation*. Here, a cryptographic primitive is employed that exhibits algebraic homomorphic properties that enable the count to be performed without needing to decrypt individual ballots. Recall that the Paillier algorithm has the property that the product of the encryption of a set of values equals the encryption of the sum of the values:

$$\prod_{i}^{n} \mathcal{E}_P(x_i) = \mathcal{E}_P\left(\sum_{i}^{n} x_i\right)$$

This property can be exploited to extract the overall count without having to decrypt individual votes. For a simple yes/no referendum, yes votes are represented by (randomised) encryptions of $+1$, whilst no votes are encryptions of 0. Suppose that n votes are cast. The product of the encryptions of votes is formed and the result decrypted. If the overall sum is greater than $n/2$, the ayes carry it, if the sum is less than $n/2$, the nays have it. For elections with choices of more than two candidates or options, more subtle encodings are required [5].

Tabulation using mixes can, thus, be thought of as analogous to conventional counting of paper ballots whilst homomorphic tabulation can be thought of as roughly analogous to the operation of lever machines: only the cumulative totals for each candidate are output. Care has to be taken to ensure that ballots are only ever encryptions of valid votes, in this simple case, of either 0 or 1, otherwise a single ballot can seriously skew the count. Zero-knowledge style proofs, see below, may be employed here to verify that the plaintext values are drawn from the appropriate set.

5.8.8 Cut and Choose

Cut-and-choose is a common device to avoid having to trust a device that performs cryptographic operations, typically encryption. The obvious approach is to require the keys or randomisations to be revealed so that the encryption can be checked. In many situations, notably voting, this is not satisfactory as the proof could be transferred to another party, i.e., a coercer. Verifying the encrypted term renders it useless in the rest of the protocol.[3] What we want is a way for the verifier to be confident that the encryption is correct without being able to prove this to anyone else.

The term comes from the simple protocol for ensuring fairness in the sharing out of a cake: one party makes the cut and the other gets to choose.

[3] This reminds me of being puzzled as a kid by talk of “testing the bomb”.

The effect is to motivate the first party to try to cut as fairly as possible. Analogously, an encryption device is required to commit to several independent encryptions of the given plaintext. It is then challenged to reveal the keys or randomising factors for all but one, randomly chosen by the verifier. If all the challenged encryptions prove to be valid, then it is a good bet that the un-revealed encryption is valid too and can be used in the rest of the protocol.

5.8.9 Zero-Knowledge Proofs

Cut-and-choose protocols involve generating surplus ciphertexts, auditing a randomly selected subset and discarding the audited elements, as their cover has been blown. A more subtle way of establishing confidence in a claim, for example that a given ciphertext really is an encryption of a claimed plaintext, is to use *zero-knowledge* proofs.

An interactive zero-knowledge proof (ZKP) is a protocol in which one party, the prover P, demonstrates the truth of a claim or knowledge of a fact to another, the verifier V, without V learning anything other than the truth of the statement or claim. Such protocols typically involve a sequence of random challenges issued by the verifier to the prover.

A typical example of such a protocol is the Chaum–Pedersen protocol [9] that is designed to prove plaintext equivalence of a pair of ElGamal encryptions without revealing either the plaintext, the secret key or the re-randomising factor. This situation crops up where a server has performed a re-encryption on an ElGamal ciphertext and wants to prove the correctness without revealing either the plaintext or the re-randomisation factor.

Suppose that P presents V with a pair of ElGamal ciphertexts (y_1, y_2) and (z_1, z_2) and claims that they are related by a re-encryption. They can both compute $w := z_1/y_1$ and $u := z_2/y_2$. Now the truth of the claim that they are related by re-encryption is equivalent to showing that (α, β, w, u) is a DDH tuple, i.e., $\exists\ x$ and k such that $w = \alpha^x$, $u = \alpha^{x \cdot k}$ and $\beta = \alpha^k$. Here, k is thought of as the secret ElGamal key and x the re-encryption factor. Where the prover P is a mix server demonstrating plaintext equivalence, P will know the re-encryption factor x but not the decryption key k.

The three-step protocol follows the standard pattern for ZKPs: P generates some fresh randomness, s, which serves to blind the secret and makes a commitment to s. V responds with a random challenge, c, to which P can respond in a way that is verifiable by V, only if the secret value x exists and is known to P.

1. $s \in Z_q^* : P \rightarrow V : (a, b) := (\alpha^s, \beta^s)$
2. $c \in Z_q^* : V \rightarrow P : c$
3. $P \rightarrow V : t = s + c.x$

Now V can check:

$$\alpha^t = a.w^c \quad \text{and} \quad \beta^t = b.u^c$$

Informally, we see that the secret random factor s chosen by P serves to conceal the secret value x from V. If P does not know x, or indeed, the claimed equivalence is false and such an x does not exist, it will be virtually impossible for him, aside from an absurdly lucky guess, to respond to v's challenge value c with a value t that will pass V's checks.

A variant of this protocol can be used to demonstrate the correctness of a claimed decryption of a given ElGamal ciphertext. Again, the proof can be reduced to the proof of a DDH tuple. In this case, P knows k but not the randomising factor x, so we simply interchange their role in the protocol. Suppose that we have the ElGamal ciphertext $(y_1, y_2) = (\alpha^k, m.\beta^k)$ and P claims that this decrypts to m'. To check that $m = m'$, we require P prove that the tuple $(\alpha, \beta, y_1, y_2/m')$ is a DDH tuple, which it will be if and only if $m = m'$.

A similar protocol to prove correct decryption of a Paillier ciphertext can be found at [12] in the case in which the prover knows the randomisation. For Paillier, it turns out that knowledge of the secret key allows the prover to recover the randomisation as well as the plaintext. Thus, there is no need for a separate protocol for the case in which the prover is ignorant of the randomisation. This is in contrast to ElGamal, where knowledge of the secret key does not help recovering the randomisation.

5.8.10 Digital Signatures

These are (roughly) the digital analogue of conventional signatures. Anne can digitally sign a text M by computing a publicly agreed crypto hash of M and encrypting this under her private key. This encrypted hash is appended to the text M:

$$Sig_A(M) := M, \{Hash(M)\}_{SK_A}$$

If Bob knows Anne's public key, he can verify the signature as follows: he applies Anne's public key to the encrypted term and applies the hash function to the plaintext M term. If the outcome of these two computations agree, he may be confident that the text M was indeed signed by Anne and has not been altered. This assumes that Anne's private key has not been compromised.

5.9 Voter-Verifiable Schemes

Cryptography opens up novel and surprising possibilities, in particular the notion of *voter-verifiability*. A system that enjoys voter-verifiability enables voters to confirm to their own satisfaction that their vote is accurately recorded and counted whilst at the same time denying them any way of proving how they voted to a third party. The key idea to achieving voter-verifiability is to provide voters with a physical receipt at the time of casting that carries their vote in encrypted form. Voters are later able to confirm, via a *secure Web Bulletin Board* (WBB) or similar,[4] that their receipt has been correctly entered into a robust, anonymising tabulation process. We need mechanisms to ensure that:

- The voter's choice is accurately encoded in the receipt.
- All receipts posted to the WBB are accurately input into the tabulation.
- The tabulation process accurately decrypts all posted receipts.

If all three assertions are confirmed, then each voter may, by confirming that their receipts are accurately posted to the WBB, be confident that their votes are counted as cast. Furthermore, if their receipts fail to appear correctly on the WBB, they have tangible proof of this. The fact that their votes are encrypted in the receipt ensures that they cannot use it to prove how they voted to a coercer or vote-buyer.

Over the past few years, a number of cryptographic schemes have been proposed to implement this concept and provide voting systems with high assurance of accuracy and ballot secrecy. Such schemes strive to empower the voters by enabling them to contribute to the dependability of the process. Dependability by the people for the people one might say.

The Prêt à Voter suite of schemes constitutes a class of particularly voter-friendly, voter-verifiable schemes and we outline a simple version in the next section. Later we will explore a number of threats to which this basic version is vulnerable. This will lead us to develop a number of enhanced versions. It is hoped that this style of presentation will result in a gentle introduction to the key ingredients of the Prêt à Voter approach whilst making clear the subtleties involved in designing such schemes.

5.10 Outline of Prêt à Voter

Here we outline the main ingredients of the Prêt à Voter scheme [10, 35, 39]. The key innovation of the Prêt à Voter approach is the way votes are encoded in a randomised frame of reference, i.e., a randomised candidate list. An important observation about this way of encoding the vote is that, in contrast to previous schemes, there is no need for the voters to communicate their votes to an encryption device. What is encrypted is the information that defines the frame of reference for any given ballot form, and this can be computed in advance. We will return to the significance of this observation later when we discuss the threat model. Incidentally, this encoding has another advantage: the randomisation of the candidate list results in fairness as a fixed ordering tends to favour candidates near the top of the list.

5.10.1 The Voting Ceremony

At the polling station, our voter Anne pre-registers and chooses at random a ballot form from a pile of forms individually sealed in envelopes. Example

[4] The list of encrypted receipts could be published for example in The Times.

Candidates	Your vote
Obelix	
Idefix	
Asterix	
Panoramix	
	$7rJ94K$
Destroy	Retain

Fig. 5.1. Prêt à Voter ballot form

Candidates	Your vote
Asterix	
Idefix	
Panoramix	
Obelix	
	$N5077t3$
Destroy	Retain

Fig. 5.2. Another Prêt à Voter ballot form

Your Vote
X
$7rJ94K$
Retain

Fig. 5.3. Prêt à Voter ballot receipt encoding a vote for "Idefix"

forms are shown in Figs. 5.1 and 5.2. Note that the order of the candidates and the cryptographic values vary from form to form.

In the privacy of the booth, Anne removes the ballot from its envelope and makes her selection in the usual way by placing a cross in the right hand column against the candidate of choice, or in the case of a single transferable vote (STV) system for example, she marks her ranking against the candidates. Once the selection has been made, she detaches and discards the left hand strip that carries the candidate order. The remaining right hand strip now constitutes the receipt, as shown in Fig. 5.3.

Anne now exits the booth with this receipt, registers with an official and casts her receipt in the presence of the official: the ballot receipt is placed against an optical reader or similar device that records the cryptographic value at the bottom of the strip, which we will refer to henceforth as the ballot *onion* and denote Θ, and an index value ι indicating the cell into which the X was marked. A digital signature is computed over $\{\iota, \Theta\}$ and this is printed on Anne's receipt.

The digitized copies of the receipts are transmitted to a central tabulation server which posts them to a secure WBB. This is an append-only, publicly visible facility. Only the tabulation server, and later the tabulation tellers can write to this and, once written, anything posted will remain unchanged. Voters are encouraged to visit this WBB and confirm that their receipt appears correctly and, if their receipt does not appear or appears incorrectly (i.e., with the X in the wrong position), they can appeal. Note that, as the voters hold physical authenticated receipts, they have demonstrable grounds for complaint if their receipt fails to appear.

At the time of casting the vote, in addition to the digital copy of the receipt, it is possible to make an additional paper copy that is verified by the voter and officials and added to an encrypted paper audit trail. This is similar to the notion of a Voter Verified Paper Audit Trail (VVPAT), but with encrypted receipts rather than plaintext ballots. Now the audit trail can be printed on a paper roll like a till receipt without jeopardizing privacy.[5] It is easier to implement a record on a single roll and much harder to manipulate than a bundle of ballots. Such a Verified Encrypted Paper Audit Trail (VEPAT) can be used to supplement the voter's checks: independent auditors can check the correspondence between the set of receipts in the audit record and the receipts posted to the WBB.

It is also possible to have representatives of *Helper Organisations* [2] at hand at the polling stations. They could offer a receipt verification service: checking digital signatures and possibly checking posting of receipts to the WBB.

5.10.2 Vote Counting

The value printed on the bottom of the RH column of the ballot forms, which we henceforth refer to as the ballot *onion*, is the key to extraction of the vote. Buried cryptographically in this value is the information needed to reconstruct the candidate order and so interpret the vote value encoded on the receipt. This information is encrypted under the secret keys shared using a threshold scheme amongst a number of tellers. Thus, only a threshold set of the tellers acting in concert are able to reconstruct the candidate order and so interpret the vote value encoded on the receipt.

Each form will have a unique, secret seed value ρ drawn from some seed space S. The candidate permutation is computed using a function σ that is publicly agreed prior to the election: $\sigma : S \rightarrow \Pi_C$, where Π_C is the set of permutations on the candidate set C. Thus, each ballot form may be thought of as a tuple:

$$(\pi, \Theta)$$

[5] With a conventional VVPAT system with un-encrypted ballots, using a till roll record introduces the possibility of correlation between the order on the roll and the order voters entered the booth, undermining ballot privacy.

where π is the candidate order and the encrypted term $\Theta = \mathcal{E}(\rho)$ is the onion. The ballot is well-formed if and only if $\pi = \sigma(\rho)$. A receipt has the form:

$$(\iota, \Theta)$$

where ι is an index value indicating where the voter placed her X or a vector giving her rankings.

After a suitable period, once any problems are suitably resolved, we can start the counting process. Here we will describe counting using *anonymising mixes* to guarantee ballot secrecy and *partial random checking* to guarantee correctness. Other approaches are possible, for example, if Paillier encryption is used, it would be feasible to use homomorphic tabulation [3].

We assume that the candidate list information has been encrypted in the ballot onions using a randomising algorithm that supports re-encryption, e.g., ElGamal or Paillier. We also assume for the moment that the index values, indicating the position of the X for example, have been absorbed into the onion term to give a pure ElGamal or Paillier term. Details will be presented a little later. Tabulation proceeds in two phases: first a mixing phase to provide privacy (rather like shaking the ballot box) followed by a decryption phase (unsealing and unlocking the ballot box). The first phase will be performed by a set of *mix tellers*. The mix tellers do not need to know any secret keys in order to perform a re-encryption, only the public key under which the onion encryption was performed.

Suppose that we have j mix tellers, each of which will perform two re-encryption mixes before handing on to the next teller. Requiring each mix teller to perform two mixes is a convenient device to facilitate the audit process. More precisely, the first mix teller takes in the batch of receipts, re-encrypts each and then performs a secret shuffle on the resulting batch of terms. The resulting batch of transformed, shuffled terms is posted to the next column of the WBB. The first teller then takes the output of its first mix and repeats the process, but with an independent shuffle, and again posts the result to the next column of the WBB. Once the first teller has finished performing its two mixes, the second teller takes the output batch of the previous teller's second mix and performs two further mixes, and so on through the full set of mix tellers.

At the end of this process, the batch of receipts would have undergone $2j$ re-encryptions and shuffles and are ready to be decrypted. Decryption will then be performed by a threshold set of decryption tellers who hold secret shares for the onion encryption.

Once all this is completed, the final decrypted anonymised votes appear in the final column of the WBB and these can be tallied in a conventional fashion and can be verified by anyone.

5.11 Auditing the Election

So far we have described the process under the assumption that all the steps are executed faithfully. We do not want the integrity of the election to depend on the correct behaviour of the entities involved and so we now introduce the mechanisms that will detect any malfunction or corruption.

5.11.1 Auditing the Ballot Generation Authority

The first place where things could go wrong is in the creation of the ballot forms. If a ballot form is incorrectly constructed, in the sense that if the candidate list shown on the form does not correspond to the order given by the seed value buried in the onion value on the form, then the voter's choice will not be accurately encoded. We, therefore, need a mechanism to detect incorrectly constructed forms, without revealing keys, seed values or randomisation values.

If, as we have done above, we assume that the ballot forms are created in advance, we can perform a random audit on a proportion of the forms. So we require the ballot creation authority (or authorities) to create an excess number of forms, perhaps twice as many as actually required, and allow independent organisations to make random selections of an appropriate proportion. For example, we might allocate five auditing organisations and allow each to select up to 10%. For these selected forms, the seed and/or randomisation factors are revealed, thereby allowing the auditors to recompute the Θ and π and confirm that they agree with those printed on the forms.

A rather elegant way of revealing the audit information for selected forms whilst ensuring that it is kept secret for ballot forms that are used to cast votes is the "Scratch and Vote" mechanism of Adida and Rivest [2]. Here, the audit information is printed on the ballot forms but concealed by a scratch strip. Revealing the information by removing the strip automatically invalidates the form for voting. In the '05 version of Prêt à Voter, this information was revealed by having the decryption tellers online to extract the seeds for audited forms. The Adida/Rivest approach avoids the need to have the tellers online.

The process of auditing a ballot form is accomplished by recomputing the values on the form from the cryptographic values. Thus, the onion is recomputed from the seed value, randomisation and the teller public keys. The candidate order is also recomputed applying the publicly known function σ to the seed value. If these agree with the values printed on the form, we may conclude that the form was correctly formed.

In addition to the audits performed by appropriate authorities before, during and after the election, we can allow the voters to choose forms at random for audit. We will discuss such possibilities in detail when we come on to discuss on-demand creation of ballot forms.

5.11.2 Auditing the Mix Tellers

Next, we need to confirm that the mix tellers perform all their actions correctly, i.e.: each column is a re-encryption and permutation of the previous column, without revealing the permutations. Numerous techniques have been proposed to achieve this, for example the approach proposed by Neff [29]. For a comprehensive survey, see [2] or [13]. The technique that we describe here, as it is both rather intuitive and flexible, is that of *randomised partial checking* [23]. Half the links are randomly chosen to be revealed and verified. The choice of links, whilst essentially random, is carefully constrained in such a way as to ensure that no decrypted vote can be traced back to the original ballot receipt.

The selection process is best described as follows: recall that each mix teller performs two mixes and so has three columns of the WBB associated with it: an input column, a middle column (the output of its first mix) and a final output column (the output of its second mix, which also serves as the input for the next teller). The auditor goes down the middle column and for each term makes a random *left/right* choice. If *left* is chosen, the teller is required to reveal the incoming link (the source term in the previous column) and the associated re-randomisation factor for that link. Similarly, if *right* is chosen, the teller must reveal the outgoing link and associated re-randomisation factor. This enables the auditors to confirm that the target term is indeed a genuine re-encryption of the claimed source term. These calculations may be confirmed by anyone. The process is repeated with independent auditing selections for each of the tellers.

Thus, each link has a 50/50 chance of being audited and yet our construction ensures that no complete link across the two mixes associated with the teller is ever revealed.

Figure 5.4 illustrates this. The first column, marked *posting 1*, is the input to the first teller. The second column is the output of its first mix. The third column is both the output of the first teller's second mix and the input to the first mix of the second teller. The auditor makes random L/R selections for each term in the second column and the teller must reveal the incoming or outgoing link information accordingly. Similarly for the fourth column, which is the middle column for the second teller.

5.11.3 Auditing the Decryption Tellers

Finally, we also need to confirm that the final decryptions are performed correctly. Here we can be more direct and audit every decryption as we do not need to worry about anonymity at this stage. Given that we are using randomising encryptions here, the process of checking the correctness of the decryptions is not quite trivial: we cannot simply perform the encryption of the claimed plaintext and check the result agrees with the ciphertext, as would be possible for a deterministic algorithm. And of course we do not want to

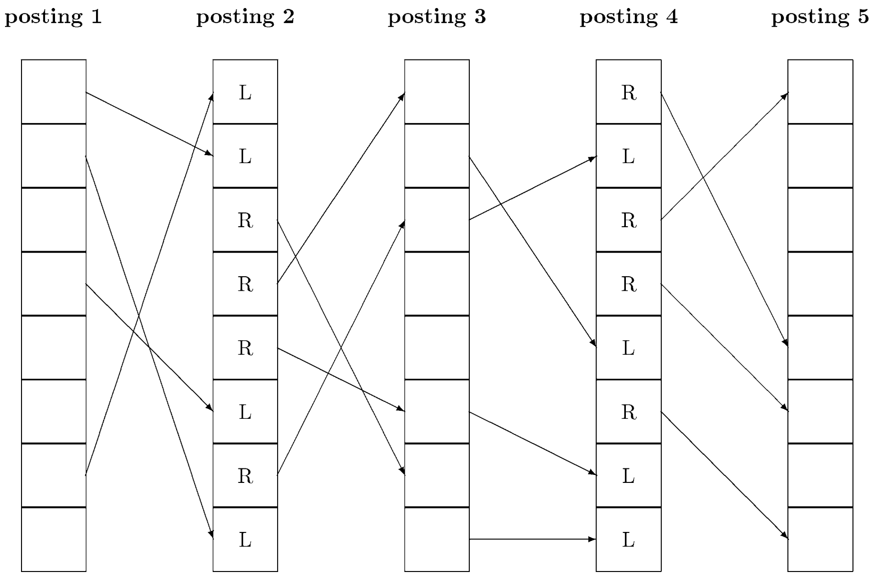


Fig. 5.4. Revealed links for auditing of two tellers

reveal the secret keys. Here we can use the ZKP techniques, e.g., Chaum Pedersen for ElGamal or Damgård et al. for Paillier [12], to check that all partial decryption shares are valid. The final combination of shares is publicly verifiable.

5.12 Threats and Trust Models

We now discuss a number of known threats against voting systems, in particular voter-verifiable schemes. For some of these, the scheme described above is already robust. For others, such as ensuring the secrecy of the information created by the ballot generation authority, chain of custody, etc., we will need to introduce enhancements, described in the following section.

5.12.1 Leaky Ballot Creation Authority

So far, we have assumed that a single authority is responsible for the generation and printing of the ballot forms and proposed random pre-auditing to ensure that ballot forms are correctly constructed and suitably randomised. This auditing removes the need to trust the authority with respect to the integrity requirement, but we still need to trust the authority to preserve the secrecy of the seed value and the onion/candidate list association. Clearly, if the ballot authority were to leak this information, the scheme would become susceptible to coercion or vote buying on a large scale.

5.12.2 Chain of Custody

Here the threat is to the integrity and secrecy of the ballot forms between the time of their creation and use. We need to provide mechanisms to ensure that none of the onion/candidate list associations are leaked during storage and distribution. We also need to guard against corrupt ballot forms being smuggled into the system.

Various counter-measures are possible, for example, anti-counterfeiting measures could be used to prevent fakes being introduced. Ballot forms could be kept in sealed envelopes to be revealed only by the voters in the booth. Alternatively, a scratch card style mechanism along the lines suggested in [38] could be used to conceal the onion value until it is revealed at the time of vote casting. The ballot forms would also need to be stored and distributed in locked, sealed boxes. Further random audits could be performed throughout the voting period, perhaps giving voters the opportunity to audit forms too.

All of these counter-measures are rather procedural in nature and so require various trust assumptions. We will see shortly how on-demand printing and distributed construction of encrypted ballots with post-auditing provide arguably stronger counter-measures, i.e., that require weaker trust assumptions.

5.12.3 Chain Voting

Conventional pencil and paper elections, in particular the UK system, are vulnerable to a style of vote buying known as chain voting. Here, the ballot forms are a controlled resource: on entering the polling station, the voter is registered and marked off on the electoral roll. They are given a ballot form which they take to the booth, mark and then cast in the ballot box. In principle, officials should observe the voters casting their ballot.

The attack works as follows: the coercer smuggles a blank ballot form out of the polling station. The controls on the distribution of the forms should make this difficult, but in practice, there are many ways in which it could be achieved, for example, bribing an official or by sleight of hand. Having marked the form for the candidate of their choice, the coercer intercepts a voter entering the polling station. The voter is told that if, when they exit the polling station, they hand a fresh, blank form back to the coercer they will receive a reward. The coercer can now use this form on the next victim and so on. Note that, once the attack is initialised, the controls on the ballot forms works in the coercer's favour: if the voter emerges from the polling station with a blank form, it is a strong indication that they did indeed cast the marked form they were given by the coercer.

For conventional paper ballot systems, there are known counters to this threat [24]. However, for voter-verifiable schemes with preprinted ballot forms, the problem rears its head with a vengeance as the coercer who has obtained prior sight of Prêt à Voter ballot form is able to check via the WBB that the ballot has been cast as required.

5.12.4 Side Channels and Subliminal Channels

A common concern with electronic voting is that the device that captures the voter's choice may somehow leak this information, over a hidden wire, wifi, electromagnetic radiation, etc. Recently a model of Nedap touch screen machine was decertified in the Netherlands precisely because it was found that votes could be detected several metres away by measuring EM radiation. All touch screen machines and most cryptographic schemes are potentially liable to such side-channel attacks.

More subtly, the device might exploit some form of subliminal channel to encode information in a legitimate channel, for example, the WBB. Karlof et al. [27] describe the possibility of semantic and random channels in the original Chaum scheme and a version of the VoteHere scheme.

Most electronic voting systems necessarily involve the voter's choice being communicated to a device that records or encrypts the vote and so are potentially vulnerable. An important feature of Prêt à Voter is that the voter does not need to communicate her choice to any device. As remarked earlier, the vote is encoded in a randomised frame of reference. All that is encrypted is the information that defines the frame and this encryption can be performed in advance, without any knowledge of the vote value or indeed the identity of the voter who will eventually use the ballot form. This means that side-channel and subliminal channel attacks are neatly side-stepped: the device simply does not learn the information and so cannot leak it even if it had access to suitable channels.

5.12.5 Kleptographic Channels

Whilst Prêt à Voter avoids side-channels and subliminal channels, a further vulnerability can occur where a single entity is responsible for creating cryptographic variables. So called *kleptographic* channels as described in [42] are a possibility in Prêt à Voter if we use a single authority for the generation of the ballot forms. The possible relevance of such attacks to cryptographic voting schemes is described in [19]. The idea is that the entity may carefully choose the values of the crypto variables in order to leak information to a colluding party.

In the case of Prêt à Voter, the Authority might choose the seed values in such a way that a keyed cryptographic hash of the onion value indicates the candidate order. The keyed hash would be shared with a colluding party. Executing such an attack would require some searching and computation to find suitable seed values, but could pass unnoticed: the distribution of seed values would look perfectly random to anyone ignorant of the hash function. We will see shortly how introducing a pseudo-random generation of seeds or a distributed construction for the ballot forms in which a number of entities contribute to the entropy of the crypto variables counters this style of attack.

5.12.6 Retention of the Candidate List

To avoid coercion threats, it is essential to ensure that the association between the receipt and the LH strip, that carries the candidate list, is eliminated as soon as the voter has made her choice. Failing this, a coerced voter may be induced to smuggle the LH portion of the ballot form out of the polling station to show the coercer how she voted. Various measures are possible, for example, a mechanical device that enforces the destruction of the LH strip. Perhaps the neatest approach is to ensure an ample supply of *decoy* LH strips, so that a voter who is being coerced could simply pick up an alternative strip showing a candidate order that meets the coercer's requirements. A further simple technique is to segment the LH strip horizontally between the candidate names, so that the strip will fall apart once detached from the RH strip.

5.12.7 Collusion Between Mix Tellers and Auditors

Collusion between the mix tellers and the auditors can undermine integrity. If a teller knows in advance which links will be audited, they can corrupt ballots along the unaudited links with impunity. Drawing the tellers and auditors from hostile organisations helps reduce the likelihood of collusion. A publicly verifiable mechanism to generate the selections based, for example, on a public lottery is another possibility. An alternative approach, based on the Fiat and Shamir heuristic [17], is to derive the audit selections from a pre-determined cryptographic hash of the posted information. The integrity guarantees provided in this way are only conditional.

If re-encryption mixes are used, then it is feasible to rerun the mixes, or indeed, run several mixes in parallel using independent tellers and independently audit these using different auditing authorities. The fact that mixes and audits can be rerun without undermining privacy is one of the key advantages of using re-encryption mixes over decryption mixes. With decryption mixes, the set of terms generated at each stage will be the same each time the mix is rerun and so it is not possible independently to audit different runs of the mixes without revealing too much information.

5.12.8 Ballot Stuffing

Another danger, common to all voting systems, is that of voting officials smuggling in extra ballots into the count. Voters checking their receipts cannot detect ballot stuffing. One simple counter is simply to check that the number of votes counted matches the number registered as having been cast. A more sophisticated approach is to post the names on the WBB of voters who have participated, so that anyone listed who did not participate can object. This counter-mechanism has dangers of its own, e.g. forced abstention attacks: voters are told that if they vote at all they will be punished. The balance between such threats is difficult to evaluate and probably depends on context. Having independent auditors check the correspondence between a VEPAT and the posted receipts would also help.

5.13 Enhancements and Counter-Measures

For many of the threats above, Prêt à Voter is either resistant or we have already suggested counter-measures. To address the remaining threats, we now describe some enhancements to the basic scheme.

5.13.1 On-Demand Generation of Prêt à Voter Ballot Forms

Generating the ballot forms in advance has several attractive features: simplicity and the ability to catch problems early. It does, however, have the disadvantage that we have to worry about chain of custody problems. In this section, we present a mechanism to generate the forms ab initio, including generating the seed entropy, in the booth. In the next section, we will describe a distributed construction of encrypted ballot forms.

We assume that the public key of the tellers, $PK_T = (g, n)$, is suitably certified and publicised and that there is a publicly agreed function, σ, from the seed space in to the set of permutations of the candidates. For the purposes of this section, we will further assume that each booth device creates two public key pairs, PK_{b1}, SK_{b1} and PK_{b2}, SK_{b2}.

Anne is now given a form at random that carries only a unique, random serial number ξ. When she enters the booth, she feeds this form into a device that reads the serial number. It applies its first secret, signing key SK_{b1} to ξ to generate the seed value ρ from which it computes the candidate order π. It also computes the onion value θ by encrypting the seed value ρ with the public threshold teller key PK_T. The randomisation used in this encryption is generated by signing ξ with its second secret key SK_{b2}.

It prints the candidates on the LH column of the form in the appropriate order. On the RH strip, it prints the onion value to give a conventional Prêt à Voter form. To facilitate auditing, we can use a variant of the Adida/Rivest offline audit mechanism [3]: on the left hand side of the ballot form, the device prints information to enable checking the well-formedness of ballot forms selected for audit. We will discuss this in more detail later; for the moment, we denote the audit information by μ. For a form used for voting, this information will be discarded along with the candidate order, but it will be preserved on a form destined for audit.

Thus, the seed is computed as:

$$\rho := \{\xi\}_{SK_{b1}} \pmod{n}$$

the randomisation factor as:

$$\zeta := \{\xi\}_{SK_{b2}} \pmod{n}$$

ξ	

Fig. 5.5. Proto-ballot form with serial number

Idefix	
Asterix	
Obelix	
Panoramix	
ξ	Θ
μ	

Fig. 5.6. Prêt à Voter ballot form showing the candidate list, onion and audit information

the candidate order as:

$$\pi := \sigma(\rho)$$

and finally the onion value:

$$\theta := \{\rho, \zeta\}_{PK_T}$$

The figures illustrate this: Fig. 5.5 shows a typical proto-ballot form bearing only a serial number. Figure 5.6 shows the form after the booth device has computed and printed the candidates and audit information μ on the LH side, and the onion value at the bottom of the RH side.

The generation of the seed and randomisation as deterministic signatures on the serial number serves to counter kleptographic attacks in which the device leaks information over subliminal channels by careful selection of the seed value [19]. The proposed mechanism ensures that the device has no freedom in the choice of the seed values.

5.13.2 Distributed Generation of Paillier Encrypted Ballot Forms

The disadvantage of the approach above is that the booth device necessarily learns the seed values and the association of the candidate order and onion value. We could implement these devices in such a way as to ensure that all information is erased once the form is printed. However, guaranteeing this may be difficult and, furthermore, there are dangers of the information being leaked over side-channels. This section addresses these issues.

Onions are generated in advance by a set of l *Clerks* in such a way that each contributes to the entropy of the cryptographic values from which the candidate list is derived. Furthermore, these values remain encrypted throughout. As a result, all the clerks would have to collude to determine the seeds values.

As before, we assume a set of decryption tellers who hold the key shares for a threshold Paillier algorithm with Teller public key PK_T: (g, n). This public key is known to the Clerks and is used in the construction of the encrypted ballot forms.

The first clerk C_0 generates a batch of initial seeds $\bar{s}_i^0$ drawn at random from Z_n^*. From these, C_0 generates a batch of initial onions by Paillier encrypting each $\bar{s}_i^0$ under the Teller key:

$$\mathcal{E}(\bar{s}_i^0, x_i^0) = (g^{\bar{s}_i^0} \cdot (x_i^0)^n) \pmod{n^2}$$

for fresh random values $\bar{x}_i^0$ drawn from Z_n^*.

The remaining $l-1$ Clerks now perform re-encryption mixes and transformations on this batch of onions: each Clerk takes the batch of onions output by the previous Clerk and performs a combined re-encryption along with an injection of fresh seed entropy into the seed values. For each onion, the seed entropy drawn from Z_n^* is injected into the seed value of the onion. The seed entropy and randomising factors will be independently chosen for each onion.

More precisely, for ith onion of the $j-1$th batch $\Theta_i^{j-1} := \mathcal{E}(s_i^{j-1}, x_i^{j-1})$, the jth Clerk C_j generates fresh, random values $\bar{x}_i^j$ and $\bar{s}_i^j \in Z_n^*$ and multiplies Θ_i^{j-1} by $\mathcal{E}(\bar{s}_i^j, \bar{x}_i^j)$:

$$\Theta_i^j = \mathcal{E}(s_{i^j}, x_i^j) = \mathcal{E}(s_i^{j-1}, x_i^{j-1}) \cdot \mathcal{E}(\bar{x}_i^j, \bar{x}_i^j) = \mathcal{E}(s_i^{j-1} + \bar{s}_i^j, x_i^{j-1} \times \bar{x}_i^j)$$

where

$$\begin{aligned} x_i^j &:= x_i^{j-1} \times \bar{x}_i^j \pmod{n^2} \\ s_i^j &:= s_i^{j-1} + \bar{s}_i^j \pmod{n} \end{aligned}$$

Having transformed each onion in this way, the Clerk C_j then performs a secret shuffle on the batch and outputs the result to the next Clerk, C_{j+1}.

So the final output after $l-1$ mixes is a batch of onions of the form: $\Theta_i := \mathcal{E}(s_i, x_i) = (\alpha^{s_i} \cdot (x_i)^n)$ where:

$$x_i = x_i^l \quad \text{and} \quad s_i = s_i^l$$

$$s_i = \sum_{j=0}^{l} \bar{s}_i^j \pmod{n} \qquad x_i = \prod_{j=0}^{l} \bar{x}_i^j \pmod{n^2}$$

As the seed values, and hence the candidate orders remain encrypted, none of the Clerks knows the final seed values or randomisations and they would all have to collude to determine them. These onions can now be stored and distributed in this form, thus avoiding the chain of custody problems mentioned above. Kleptographic channels are also avoided as no single entity is able to choose the seed values in such a way as to leak information.

We could now proceed to use these onions, which we will refer to as *receipt onions*, directly in the construction of the ballot forms, much as described in the previous section. However, we would like to ensure that the booth device does not learn the value of the onion used in the receipt as this would result in a single device learning the association of receipt onion and candidate order. Furthermore, we would like to avoid giving the booth devices the capability to decrypt the onions unaided. To this end, further steps can be taken to transform the *receipt* onions constructed above into *booth onions* that have the same seed values and are decryptable by the booth devices. We can set things up in such a way as to ensure that the booth device only sees the booth onion and not the *receipt onions* by, for example, concealing the receipt onion with a scratch strip. Due to space constraints, we omit the details but refer the interested reader to the Technical Report [37].

5.14 Auditing "On-Demand" Ballot Forms

The mechanisms described above allow for the on-demand printing of ballot forms in the booth. This has advantages in terms of removing the need to trust a single entity to keep the ballot form information secret and avoids chain of custody problems. On the other hand, it means that we can no longer rely on pre-auditing the ballot forms. Given that we want to avoid having to trust the device in the booth, we must introduce alternative techniques to detect and deter any corruption or malfunction in the creation of the ballot forms. To address this, we introduce a cut-and-choose element into the protocol.

To this end, voters could be furnished with two or perhaps more encrypted ballot forms. For each of these, the booth device decrypts the onion, computes the candidate order and prints this on the form. The voter selects one at random to cast her vote: the others are audited and destroyed, or retained as evidence of corruption if they fail the audit. Care must be taken to avoid introducing dangers of double voting or chain voting, etc. Double voting is probably fairly easily countered by the supervised casting of ballot receipts in the presence of officials who ensure that only one form is cast and the voter's name is marked as having voted.

Chain voting threats might be a little more delicate to counter: here a malicious voter, Yves, secretes a decrypted ballot form and smuggles it out of the polling station to initiate a chain voting attack. Keeping a log of serial numbers issued to a voter, in the manner of known counters to chain voting [25], would help here. If we employ the scratch strip technique of Sect. 5.13.2, this has the effect of concealing the receipt onion value and so thwarting chain voting attacks. The receipt onion would only be revealed at the time of casting the ballot and verified in the presence of officials.

The double sided forms of [36] provide a mechanism to keep a clear account of the distribution of ballot forms. Here, each side of a form carries an independent onion. For each side, the onion is decrypted and the candidate

Obelix		
Asterix		
Idefix		
Panoramix		
499052712	$7rJ94K$	

Fig. 5.7. Dual Prêt à Voter ballot form; side 1

Asterix		
Idefix		
Obelix		
Panoramix		
499052712	$Yu78gf$	

Fig. 5.8. Dual Prêt à Voter ballot form; side 2

Obelix	
Asterix	
Idefix	
Panoramix	
499052712	$7rJ94K$

Fig. 5.9. Dual Prêt à Voter ballot receipt; auditable side

X	
$Yu78gf$	

Fig. 5.10. Dual Prêt à Voter receipt; vote carrying side

order printed. This results in two independent Prêt à Voter ballot forms being printed, one on each side of the form. The voter arbitrarily selects one side to vote and the other for audit. The forms actually have a third, blank column opposite the candidate list on the other side, as shown in Figs. 5.7 and 5.8. Thus, detaching the candidate list on the voted side detaches the blank column of the flip side, thereby leaving an intact form for audit. The two sides of the resulting receipt where the voter has cast a vote for Asterix on the second side are shown in Figs. 5.9 and 5.10.

5.15 Checking the Construction of the Ballot Forms

If we use the on-demand generation of seeds and randomisation described in Sect. 5.13.1, the booth device will know the randomisation and can reveal it for audit purposes. Alternatively, ZKPs may be provided to show that the claimed seed value is the correct plaintext of the onion.

Ballot forms could be audited at several points in the process. A first check would be performed at the time of casting the receipt: official auditing devices would be available at the registration desk.

As we do not want to trust the official audit devices, we introduce *voter helper organisations* [2]: representatives of the various parties, etc. provide auditing devices at the polling stations so that further independent checks can be performed just after casting their receipt. These audit devices could also check the validity of the digital signatures applied at the time of casting.

Additionally, auditing could take place on material posted to the WBB. All the information on both sides of the receipts would be posted. The voted and auditable sides should be posted to separate regions of the WBB in such a way as to lose any association between the two sides. The voted sides would be processed via the tabulation mixes whilst the auditable sides are available to be verified by anyone. Note that, in principle, anyone can write an auditing program: the algorithms are all public and indeed quite standard.

Audit forms posted to the WBB could themselves be decrypted by the tellers in a sort of tabulation mix, except that here we would carry the candidate order through the mix and check that the order computed from the decrypted seeds matches the order carried through the mix. This strategy avoids any need to reveal audit information and it might be tempting to adopt it as the sole ballot auditing mechanism. In practice, it is probably preferable to use it as a supplementary mechanism to provide added assurance, as any corruption would be detected very late in the process. Audit forms would be required to carry booth identifiers, so that any corruption detected can be traced back to the offending booth device and isolated.

5.16 Re-Encryption/Tabulation Mixes

Paillier or ElGamal encrypted terms can be put through a conventional re-encryption mix, but Prêt à Voter ballot receipts do not have the form of straight encrypted terms, so they cannot be treated quite so straightforwardly. The ballot receipts have the form of a pair: an index and an encrypted term. An obvious approach would be to send the receipt terms through the mix re-encrypting the onions whilst leaving the index values unchanged. The problem with this is that an adversary is able to partition the mix according to the index values. There may be situations in which this is acceptable, for example, large elections in which the number of voters vastly exceeds the number of voting options. In general, this is unsatisfactory.

A more satisfactory solution, at least for the case of a simple selection of one candidate from the list, is described here. In this case, we restrict ourselves to simple cyclic shifts from the base ordering of the candidates. For single candidate choice voting, this is sufficient to ensure that the receipts do not reveal the voter's selection. More general styles of election, for example, in which voters are required to indicate a ranking of the candidates, would require full permutations of the candidate list. We will discuss how to achieve full mixing in the more general case in the next section.

Suppose that s is the seed for a given ballot form and ν is the number of candidates, then we can simply let $s \pmod{\nu}$ be the (upward) shift of the candidate list. We can absorb the index value ι on the ballot receipt into the Pailler onion:

$$\{\iota, (\alpha^s.y^n)\} \rightarrow (\alpha^\iota.\alpha^s.y^n) = (\alpha^{\iota+s}.y^n)$$

This gives a pure Paillier encryption of the value $\iota + s$ which, taken modulo ν, gives the voter's original candidate choice in the canonical base ordering. These pure Paillier terms can now be sent through a conventional re-encryption mix as described earlier.

5.17 Handling Full Permutations and STV Style Elections

Handling full permutations of the candidate list is straightforward in the 2005 version of Prêt à Voter that uses RSA onions and decryption mixes [10]. The permutation of the candidates shown on the original ballot form is built up as the product of permutations derived from seed values buried in each layer of the onion. During the tabulation mix, as we reveal a new seed value at each stage of the mix, we simply apply the inverse permutation derived from this seed. As long as we arrange for these inverse permutations to be applied in the reverse order, we undo the effect of the permutation applied in the construction of the ballot form. The overall effect is to output the indices (or rank vectors) in the canonical, base ordering [10].

Dealing with full permutations of the candidate list where randomising encryption and re-encryption mixes are used, it is not clear how to generalise the approach of Sect. 5.16. One solution is simply to have one onion against each candidate, encrypting the corresponding candidate code. For a single candidate selection, the ballot receipt would in effect simply be the onion value against the chosen candidate.

Using one onion per candidate also allows us to deal with STV style counting. However, if this is done naively, we may be open to the *Italian attack*: a coercer requires the voter to cast a ballot with a particular pattern of choices in the low order rankings. This, in effect, identifies the ballot, and in a verifiable scheme, is dangerous: the coercer can check that a vote of the appropriate form appears on the WBB.

A neat counter to this threat, due to Heather [22], works as follows. Ballot receipts are formed as vectors of pairs comprising a rank value and an onion encrypting a candidate index. Where voters have not filled in all lower rankings, the remaining onions are simply inserted in the vector in the random order of the ballot form. These receipts are posted as usual and can be checked as before. Before mixing starts, we introduce an initial normalising step in which each ballot vector is re-ordered into rank order. Once this is done, the ranking can be dropped and the ballots are sent through re-encryption mixes as tuples of onions.

Let Θ_i denote an onion encrypting the index of candidate i Thus, for example, a receipt might take the form of a vector:

$$\{(3, \Theta_3), (1, \Theta_2), (4, \Theta_4), (2, \Theta_1)\}$$

would be normalised to:

$$\{(1, \Theta_2), (2, \Theta_1), (3, \Theta_3), (4, \Theta_4)\}$$

and then to:

$$\{(\Theta_2), (\Theta_1), (\Theta_3), (\Theta_4)\}$$

These ballot vectors are now fed into a sequence of re-encryption mixes in which each onion is independently re-encrypted but the vector order is preserved. Thus, the tuple above would be transformed to:

$$\{(\Theta_2'), (\Theta_1'), (\Theta_3'), (\Theta_4')\}$$

where the prime indicates re-encryption. Once the ballots have gone through an appropriate number of re-encryption mixes, we move to the decryption phase. Now we can adopt a lazy decryption strategy: first we just decrypt the top, ranked first, onions for each vector and perform the first phase of the STV counting. When votes are transferred from eliminated candidates, the list of onions in a vector are rotated so that the second onion goes to the top of the list whilst the previously top vote is encrypted and drops down to the bottom of the list. Thus, the vector above might be partially decrypted to:

$$\{Candidate2, (\Theta_4), (\Theta_1), (\Theta_3)\}$$

and, on transfer transformed to:

$$\{(\Theta_4'), (\Theta_1'), (\Theta_3'), (\Theta_2')\}$$

Thus, at any stage of the counting, for each ballot vector at most one candidate is revealed. The mixing between phases of counting ensures that the pattern of rankings for any ballot is never revealed, thereby countering "Italian" style attacks.

5.18 Conclusions

Confidence in voting systems, particularly electronic systems, has been severely shaken in recent years by, most notably, the debacles in the 2000 and 2004 US presidential elections. It is rather troubling that the nation that proclaims itself the most technologically advanced and the torchbearer for democracy has had such a troubled history of voting technology.

Many activists point to the evident deficiencies of many of the existing voting technologies and from this, conclude that all voting technology must be deficient and so we must return to good old fashioned paper ballots and hand counting. Whilst the critiques of many current voting technologies are certainly valid, to condemn all voting technology as a result is not valid. And of course, pencil and paper is not immune to corruption.

In this chapter, I have described an alternative response, based on systems providing a high degree of transparency and auditability. Significant strides have been made towards voting systems that are trustworthy and yet sufficiently simple to be usable and to gain the confidence of the electorate. These *end-to-end* systems strive to avoid the need to place trust in devices, software, processes or officials. Furthermore, the trustworthiness of the system ultimately rests with the electorate themselves.

I have presented the key ingredients of the Prêt à Voter schemes and their key properties. I have also described some of the known threats and presented enhancements to counter these threats. Arguably some of these threats are somewhat exotic and so the additional complexity of the enhanced versions might not on balance be worthwhile, at least in some contexts.

There is doubtless scope for further innovations and simplifications of these schemes. More analysis of these schemes is required, and such analysis will have to be extended to take full account of the surrounding socio-technical system. Whilst such schemes appear to provide high levels of trustworthiness, at least in the eyes of experts, it is not clear that the public at large would understand and appreciate the rather subtle arguments and so be prepared to trust in them to the same extent as, say, familiar pen and paper systems. This leads us to some delicate socio-technical questions:

- To what extent can the properties of cryptographic systems be explained to the general public?
- To what extent is it necessary for the general public to understand in order to have sufficient trust in such systems?
- To what extent are the assurances of independent experts enough to engender trust?
- To what extent might it be necessary to compromise on trustworthiness in order to achieve understandability by, for example, replacing cryptographic mechanisms with simpler technology or processes? [31]

Somewhat paradoxically, the very transparency and auditability of the scheme may be an obstacle: the fact that errors and corruption can be detected and rectified in a very public way may damage confidence. In conventional systems, most errors go undetected and, if detected, are handled quietly behind the scenes.

In this chapter, I have advanced the case for high assurance schemes like Prêt à Voter. Such schemes can provide far higher levels of assurance than traditional pen and paper with much lower dependence on software, hardware, processes and officials. Thus, they have the potential to restore confidence in the processes of democracy.

5.19 Future Work

The schemes that we have discussed so far are all supervised: voting takes place in a controlled environment such as a polling station and the casting of the vote occurs in the enforced isolation of a polling booth. In remote systems, voters are able to cast their votes over a suitable channel, e.g., postal, internet, telephone, interactive TV, etc. Here, the isolation of the voter whilst casting her vote cannot be enforced and the threats of vote buying or coercion are consequently much more serious. It is sometimes argued that the difficulty in ensuring coercion-resistance is enough to render remote voting inadmissible.

Clarkson et al. [11] have devised a remote implementation of Prêt à Voter with increased resistance to coercion, based on an approach of Juels et al. [26]. Voters are supplied with *tokens* that they present along with their receipt when they cast their votes. A coerced voter can use a corrupted token and later re-vote with their valid token. The validity of tokens is established only after anonymisation and so the coercer is not able to distinguish valid from invalid tokens. There remain difficulties with this approach, not least the issue of voter understanding and trust in such a mechanism. If voters do not believe in the mechanism they will again be open to coercion.

Another area that has seen very little exploration to date is that of effective recovery strategies. Verifiable schemes define mechanisms for detecting errors or corruption, but typically say little about how to respond when errors are detected. Clearly we do not want to abort an election if only a smattering of errors are detected, especially if these are negligible compared to the margin of the count. It will be necessary to determine appropriate thresholds and recovery strategies.

Acknowledgements

The author would like to thank Ben Adida, Ran Canetti, David Chaum, Michael Clarkson, Bob Cunningham, Joshua Guttman, James Heather, Markus Jakobsson, Thea Peacock, Jean-Jacques Quisquater, Ron Rivest, Steve Schneider, Paul Syverson, Vanessa Teague and Thomas Tjøstheim, as well as the members of the Newcastle Security Group, for helpful comments and discussions.

This chapter is dedicated to my late parents, to whom I owe everything.

References

1. *Anonymity Bibliography.* http://freehaven.net/anonbib/.
2. B. Adida. Advances in cryptographic voting systems, PhD thesis, MIT, July 2006.
3. B. Adida and R. L. Rivest. Scratch and vote: Self-contained paper-based cryptographic voting. In *Workshop on Privacy in the Electronic Society*, Alexandria, Virginia, 2006.
4. L. Adleman R. Rivest, and A. Shamir. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
5. O. Baudron, P.-A. Fouque, D. Pontecheval, G. Poupard, and J. Stern. Practical multi-candidate election system. In *Symposium on Principles of Distributed Computing*, pp. 274–283. ACM, 2001.
6. J. Beneloh and D. Tuinstra. Receipt-free secret-ballot elections. In *Symposium on Theory of Computing*, pp. 544–553. ACM, 1994.
7. D. Chaum. Untraceable mail, return addresses and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.
8. D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security and Privacy*, 2(1):38–47, 2004.
9. D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pp. 89–105. Springer, 1993.
10. D. Chaum, P. Y. A. Ryan, and S. Schneider. A practical, voter-verifiable election scheme. In *European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, vol. 3679. Springer, 2005.
11. M. Clarkson and A. Myers. Coercion-resistant remote voting using decryption mixes. In *Workshop on Frontiers of Electronic Elections*, Milan, 2005.
12. I. Damgard, M. Jurik, and J. Nielsen. A generalization of Paillier's public-key system with applications to electronic voting, 2003. Available from www.daimi.au.dk/~ivan/GenPaillier_finaljour.ps.
13. G. Danezis. Better anonymous communications, PhD thesis, University of Cambridge, July 2004. http://www.cl.cam.ac.uk/~gd216/thesis.pdf.
14. W. Diffie and M. E. Hellman. New directions in cryptography. In *Transactions on Information Theory*, vol. IT-22, November 1976, pp. 644–654. IEEE, 1976.
15. Y. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Transaction on Information Theory*, vol. 31, pp. 469–472. IEEE, 1985.
16. A. J. Feldman et al. Security analysis of the Diebold Accuvote-TS voting machine. Princeton, September 2006. itpolicy.princeton.edu/voting/ts-paper.pdf.
17. A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology – Crypto '86*, pp. 186–194, New York. Springer, 1987.
18. K. Fisher et al. Punchscan, introduction and definition. http://www.cisa.umbc.edu/papers/FisherWote2006.pdf.
19. M. Gogolewski et al. Kleptographic attacks on e-election schemes. In *International Conference on Emerging Trends in Information and Communication Security*, 2006. http://www.nesc.ac.uk/talks/639/Day2/workshop-slides2.pdf.

20. R. S. Brumbaugh, Ancient Greek Gadgets and Devices. Crowell, New York, 1966. ISBN 0837174279.
21. Andrew Gumbel. Steal This Vote!. Nation Books, New York, 2005.
22. J. Heather. Implementing STV securely in prêt à voter. In *Proceedings of Computer Security Foundations*, pp. 157–169, 2007.
23. M. Jakobsson, A. Juels, and R. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security Symposium*, pp. 339–353, 2002.
24. D. W. Jones. A brief illustrated history of voting, 2003. `http://www.cs.uiowa.edo/~jones/voting/pictures`.
25. D. W. Jones. Threats to voting systems, 2005. `http://vote.nist.gov/threats/papers/threats_to_voting_systems.pdf`.
26. A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant Electronic Elections. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, pp. 61–70. November 2005.
27. C. Karlof, N. Sastry, and D. Wagner. Cryptographic voting protocols: A systems perspective. In *USENIX Security Symposium*, Lecture Notes in Computer Science, vol. 3444, pp. 186–200. Springer, 2005.
28. T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *Symposium on Security and Privacy*. IEEE, 2004.
29. A. Neff. A verifiable secret shuffle and its application to e-voting. In *Conference on Computer and Communications Security*, pp. 116–125. ACM, 2001.
30. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. *Proceedings of the Eurocrypt '99*, Lecture Notes in Computer Science, vol. 1592, pp. 223–238. IACR, Springer, 1999.
31. B. Randell and P. Y. A. Ryan. Voting technologies and trust. *IEEE Security and Privacy*, 4(5):50–56, 2006.
32. R. L. Rivest. The three ballot voting system, 2006. theory.lcs.mit.edu/rivest/Rivest-TheThreeBallotVotingSystem.pdf.
33. P. Y. A. Ryan. Towards a dependability case for the Chaum voting scheme. *DIMACS Workshop on Electronic Voting – Theory and Practice*, 2004.
34. P. Y. A. Ryan. A variant of the chaum voting scheme. Technical Report CS-TR-864, University of Newcastle upon Tyne, 2004.
35. P. Y. A. Ryan. A variant of the Chaum voting scheme. In *Proceedings of the Workshop on Issues in the Theory of Security*, pp. 81–88. ACM, 2005.
36. P. Y. A. Ryan. Putting the human back in voting protocols. In *Fourteenth International Workshop on Security Protocols*, Lecture Notes in Computer Science. Springer, 2006.
37. P. Y. A. Ryan. The computer ate my vote. Technical Report Newcastle University CS-TR-988, University of Newcastle upon Tyne, 2007.
38. P. Y. A. Ryan and T. Peacock. Prêt à voter: a systems perspective. Technical Report CS-TR-929, University of Newcastle upon Tyne, 2005.
39. P. Y. A. Ryan and S. Schneider. Prêt à voter with re-encryption mixes. In *European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, vol. 4189. Springer, 2006.
40. A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

41. D. R. Stinson. *Cryptography, Theory and Practice.* Chapman and Hall, CRC, Boca Raton, 2006.
42. A. Young and M. Yung. The dark side of black-box cryptography, or: Should we trust capstone? In *Crypto'96*, Lecture Notes in Computer Science, pp. 89–103. Springer, 1996.

6

Formal Methods for Biochemical Signalling Pathways

Muffy Calder[1], Stephen Gilmore[2], Jane Hillston[2], and Vladislav Vyshemirsky[1]

[1] Department of Computing Science, University of Glasgow, Glasgow, Scotland
muffy@dcs.gla.ac.uk
[2] Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland
stg,jeh@inf.ed.ac.uk

Abstract We apply quantitative formal methods to a domain from the life sciences: biochemical signalling pathways. The key idea is to model pathways as stochastic continuous time distributed systems. Components of the system are molecular species (rather than individual molecules) and are modelled by concurrent processes that interact with each other via biochemical reactions. Through an application, we show how high level languages and analysis techniques rooted in computer science theory add significantly to the analysis presently available to computational biologists.

6.1 Introduction

This chapter considers a different and novel application for quantitative formal methods, biochemical signalling pathways. The methods we use were developed for modelling *engineered* systems such as computer networks and communications protocols, but we have found them highly suitable for modelling and reasoning about *evolved* networks such as biochemical signalling pathways.

Biochemical signalling pathways are communication mechanisms for the control and coordination of cells in living organisms. Cells "sense" a stimulus and then communicate an appropriate signal to the nucleus, which makes a response. The response depends upon the way in which the signals are communicated in the pathway. Signalling pathways are complicated communication mechanisms, with feedback, and embedded in larger networks. Understanding how these pathways function is crucial, since their malfunction results in a large number of diseases such as cancer, diabetes and cardiovascular disease. Good *predictive* models can guide experimentation and drug development for pathway interventions.

Historically, pathway models either encode static aspects, such as which components in a pathway (proteins) have the potential to interact or provide simulations of system dynamics using either ordinary differential equations (ODEs) [10, 32] or stochastic simulations of individuals using Gillespie's algorithm [13]. Here, we introduce a novel approach to analytic pathway modelling. The key idea is that *pathways have stochastic computational content*. We consider pathways as *distributed systems*, viewing the components as *processes* that can interact with each other via biochemical reactions. The reactions have *duration*, defined by (performance) *rates*; therefore, we model using high level formal languages whose underlying semantics is continuous time Markov chains (CTMCs). A distinctive aspect of our work is that we do not model individual molecules, but *species* of molecules, i.e., we model molecular concentrations.

Biological modelling is complex and error-prone. We believe that high-level stochastic modelling languages can complement the efficient numerical methods currently in widespread use by computational biologists. Process algebras have a comprehensive theory for reasoning and verification. They are also supported by state-of-the-art tools for analysis, which realise the theory mechanically and support ambitious modelling studies that include the essential representational detail demanded for physically accurate work.

We have developed models using two different high level formal languages: PEPA [19] and PRISM [21]. These languages allow us to concentrate on modelling behaviour at a high level of abstraction, focusing on compositionality, communication and interaction, rather than working at the low level detail of a CTMC or system of ODEs. Both languages have extensive tool sets and are suited to modelling and analysis of biochemical pathways, but in different ways. The former is a process algebra and so the models are easily and clearly expressed, using the built-in operators. Markovian analysis is supported by the tool set. The PRISM language represents systems using an imperative language of reactive modules. It has the capability to express a wide range of stochastic process models including both discrete- and continuous-time Markov chains and Markov decision processes. A key feature of both languages is multi-way synchronisation, essential for our approach.

In the next section, we give a brief overview of background material, presenting only the essential details of stochastic process theory needed to appreciate what follows. In Sect. 6.3, we give an introduction to our modelling approach. In Sect. 6.4, we present the syntax and semantics of the stochastic process algebra which we use, PEPA, and discuss how individual reactions and reaction pathways are modelled. In Sect. 6.5, we present an example, the ERK signalling pathway. Biochemical pathways are commonly modelled using ODE models; we compare with these in Sect. 6.6. We relate the above to a method based on model checking properties in temporal logic in Sect. 6.7. Section 6.8 contains a discussion. Further and related work is presented in Sect. 6.9 and we conclude in Sect. 6.10.

Parts of the present work were previously presented in the papers [4–6].

6.2 Preliminaries

In this section, we present background material on the stochastic processes that we use in our work, particularly Markov processes. The literature on Markov processes is vast but as introductory texts we would recommend [20, 25]. Introductions to the application of stochastic processes in the physical sciences include [12, 14]. In this brief introduction, we attempt to explain some of the fundamental assumptions on which Markovian analysis is based such as "memorylessness". By doing this, we hope to guide the reader towards an understanding of the applicability of our results and their scope.

6.2.1 Continuous Time Markov Chains

A fundamental kind of quantitative model is a CTMC, a finite-state stochastic process that evolves in continuous time. CTMCs are widely used in quantitative modelling because the numerical procedures for working with them are widely known [31].

Such a system can be easily thought of as a labelled transition system (LTS) where the labels describing the transitions from one state to another record information such as the *rate* at which this transition can occur. Logically, one might think of such an LTS as a graph where the presence of an arc from vertex i to vertex j with label r indicates the possibility of moving from state i to state j with rate r, and the absence of an arc indicates that it is impossible for the system to move directly from state i to state j. Algorithmically, it is more productive for numerical purposes to represent this information instead as a matrix, where an entry r in position ij records the rate information and a zero at position ij indicates the impossibility of moving directly from state i to state j. In practice, for typical models these matrices are usually sparse with the zero entries greatly outnumbering the non-zeros. If the model of the system is expressed in this way, the calculation of performance measures such as availability and utilisation can be obtained by using procedures of numerical linear algebra to compute the steady-state probability distribution over the states of this finite-state system and calculating the measure of interest from this. More complex measures such as response time require more sophisticated analysis procedures.

CTMCs associate an exponentially distributed random variable with each transition from state to state. The random variable expresses quantitative information about the rate at which the transition can be performed. Formally, a random variable is said to have an exponential distribution with parameter λ (where $\lambda > 0$), if it has the probability distribution function

$$F(x) = \begin{cases} 1 - \mathrm{e}^{-\lambda x} & \text{for } x > 0, \\ 0 & \text{for } x \leq 0 \end{cases}$$

The mean, or *expected value*, of this exponential distribution is $1/\lambda$. The time interval between successive events is $\mathrm{e}^{-\lambda t}$.

The *memoryless property* of the exponential distribution is so called because the time to the next event is independent of when the last event occurred. The exponential distribution is the only distribution function which has this property.

A Markov process with discrete state space (x_i) and discrete index set is called a *Markov chain.* The future behaviour of a Markov chain depends only on its current state, and not on how that state was reached. This is the *Markov, or memoryless, property.* Every finite-state Markov process can be described by its *infinitesimal generator matrix*, Q. Q_{ij} is the total transition rate from state i to state j.

Stochastic models admit many different types of analysis. Some have lower evaluation cost, but are less informative such as *steady-state analysis.* Steady-state analysis tells us only about the *stationary*, or *equilibrium*, probability distribution over all of the states of the system. This contains no information about probabilities of states near to the start of the time evolution of the system. Other types of analysis have higher evaluation cost, but are more informative such as *transient analysis.* This tells us about the probability distribution over all of the states of the system at all time points, and measures such as *first passage times* can be computed from this. Passage times are needed for the calculation of response times between an input stimulus and its associated output response.

We will be dealing with *time-homogeneous* Markov processes, where time does not influence probability. These processes will also be *irreducible*, meaning that it is possible to get to any state from any other. Finally, the states of these processes will be *positive-recurrent*, meaning that every state can be visited infinitely often. A stationary probability distribution, $\pi(\cdot)$, exists for every time-homogeneous irreducible Markov process whose states are all positive-recurrent [20]. At equilibrium, the probability flow into every state is exactly balanced by the probability flow out, so the equilibrium probability distribution can be found by solving the *global balance equation*

$$\pi Q = 0$$

subject to the normalisation condition

$$\sum_i \pi(x_i) = 1$$

From this probability distribution can be calculated performance measures of the system such as throughput and utilisation.

An alternative is to find the *transient* state probability row vector $\pi(t) = [\pi_0(t), \ldots, \pi_{n-1}(t)]$, where $\pi_i(t)$ denotes the probability that the CTMC is in state i at time t.

6.2.2 Continuous Stochastic Logic

CSL [1, 2] is a continuous time logic that allows one to express a probability measure that a temporal property is satisfied, in either transient behaviours

Table 6.1. Continuous stochastic logic (CSL) operators

Operator	CSL syntax
True	*true*
False	*false*
Conjunction	$\phi \wedge \phi$
Disjunction	$\phi \vee \phi$
Negation	$\neg\phi$
Implication	$\phi \Rightarrow \phi$
Next	$P_{\star p}[\mathbf{X}\phi]$
Unbounded until	$P_{\star p}[\phi\mathbf{U}\phi]$
Bounded until	$P_{\star p}[\phi\mathbf{U}^{\leq t}\phi]$
Bounded until	$P_{\star p}[\phi\mathbf{U}^{\geq t}\phi]$
Bounded until	$P_{\star p}[\phi\mathbf{U}^{[t_1,t_2]}\phi]$
Steady-state	$S_{\star p}[\phi]$

or in steady state behaviours. We assume a basic familiarity with the logic, which is based upon the computational tree logic CTL [9]. Properties are expressed by state or path formulae. The operators include the usual propositional connectives, plus the binary temporal operator *until* operator **U**. The *until* operator may be time bounded or unbounded. Probabilities may also be bounded. $\star p$ specifies a bound, for example, $P_{\star p}[\phi]$ is true in a state s if the probability that (state property) ϕ is satisfied by the paths from state s meets the bound $\star p$. Examples of bounds are > 0.99 and < 0.01. A special case of $\star p$ is no bound, in which case we calculate a probability.

Properties are *transient*, that is, they depend on time; or they are *steady state*, that is, they hold in the long run. Note that in this context, steady state solutions are not (generally) single states, but rather a network of states (with cycles) which define the probability distributions in the long run. Table 6.1 gives an overview of CSL, ϕ is a state formula.

We use the PRISM model checker [21] to check the validity of CSL properties. In PRISM, we write $P_{=?}[\phi]$ to return the probability of the transient property ϕ, and $S_{=?}[\phi]$ to return the probability of the steady state property ϕ. The default is checking from the initial state, but we can apply a filter thus, $P_{=?}[\psi\{\phi\}]$, which returns the probability, from the (first) state satisfying ϕ, of satisfying ψ.

6.3 Modelling Biochemical Pathways

Biochemical pathways consist of proteins, which interact with each other through chemical reactions. The *pathway* is a sequence of reactions, which may have feedback and other dependencies between them. The *signal* is a high concentration, or abundance, of particular molecular species, and by the sequence of reactions, the signal is carried down the pathway. In modelling terms, a reaction is an activity between reactants to produce products; so, reactants play the role of producer(s), and products the role of consumer(s).

From the description above, we can see that we can view a pathway as a distributed system: all stages of the pathway may be activated at once. We associate a concurrent, computational process with each of the proteins in the pathway. In other words, in our approach *proteins are processes* and in the underlying CTMC, *reactions are transitions*. Processes (i.e., proteins) interact or communicate with each other *synchronously* by participating in reactions which build up and break down proteins. The availability of reactants plays a crucial role in the progress of the pathway. This is usually expressed in terms of the rate of reaction being proportional to the concentration of the reactants. In basic terms, a producer can participate in a reaction when there is enough species for a reaction and a consumer can participate when it is ready to be replenished. A reaction occurs only when all the producers and consumers are ready to participate.

It is important to note that we view the protein *species* as a process, rather than each *molecule* as a process. Thus, the local states of each process reflect different levels of concentration for that species. This corresponds to a *population* type model (rather than an *individual* type model) and more readily allows us to reflect the dynamics of the reactions. In traditional population models, species are represented as real valued molar concentrations. In our approach, the concentrations are discretised, each level representing an interval of concentration values. The granularity of the discretisation can vary; the coarsest possible being two values (representing, for example, enough and not enough, or *high* and *low*). Time is the only continuous variable, all others are discrete.

6.4 Modelling Pathways in PEPA

We assume some familiarity with process algebra; a brief overview of the stochastic process algebra PEPA is below, see [19] for further details.

6.4.1 Syntax of the Language

As in all process algebras, PEPA describes the behaviour of a system as a set of processes or components, which undertake activities. All activities in PEPA are timed. In order to capture variability and uncertainty, the duration of each action is assumed to be a random variable, which is exponentially distributed. For example, using the prefix combinator, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r (average delay $1/r$), and it subsequently behaves as S. The component $P + Q$ represents a system which may behave either as P or as Q. The activities of both P and Q are enabled. The first activity to complete distinguishes one of them; the other is discarded. The system will behave as the derivative resulting from the evolution of the chosen component. The expected duration of the first activities of P and Q will influence the outcome of the choice (the *race policy*), so the outcome is probabilistic.

It is convenient to be able to assign names to patterns of behaviour associated with components. Constants are components whose meaning is given by a defining equation. The notation for this is $X \overset{def}{=} E$. The name X is in scope in the expression on the right hand side, meaning that, for example, $X \overset{def}{=} (\alpha, r).X$ performs α at rate r forever. PEPA supports multi-way cooperations between components: the result of synchronising on an activity α is, thus, another α, available for further synchronisation. We write $P \underset{L}{\bowtie} Q$ to denote cooperation between P and Q over L. The set which is used as the subscript to the cooperation symbol, the *cooperation set* L, determines those activities on which the *cooperands* are forced to synchronise. For action types not in L, the components proceed independently and concurrently with their enabled activities. We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty. PEPA was developed originally for performance modelling of computer and communication systems. In this context, it is assumed that synchronised activities respect the notion of *bounded capacity*: a component cannot perform an activity at a rate greater than its own specification for the activity. Therefore, in PEPA the rate for the synchronised activities is the *minimum* of the rates of the synchronising activities. For example, if process A performs α with rate λ_1 and process B performs α with rate λ_2, then the rate of the shared activity when A cooperates with B on α is $\min(\lambda_1, \lambda_2)$. We use the distinguished symbol $\top$ to indicate that a component is *passive* with respect to an activity, i.e., for all rates k, $\min(k, \top) = k$.

6.4.2 Semantics of the Language

PEPA has a structured operational semantics that generates a labelled multi-transition system for any PEPA expression. It is a multi-transition system because the multiplicity of activities is important and may change the dynamic behaviour of the model. Via the structured operational semantics, PEPA models give rise to CTMCs. The relationship between the process algebra model and the CTMC representation is the following. The process terms (P_i) reachable from the initial state of the PEPA model by applying the operational semantics of the language form the states of the CTMC. For every set of labelled transitions between states P_i and P_j of the model $\{(\alpha_1, r_1), \ldots, (\alpha_n, r_n)\}$, add a transition with rate r where r is the sum of $r_1, \ldots, r_n$. The activity labels (α_i) are necessary at the process algebra in order to enforce synchronisation points, but are no longer needed at the Markov chain level.

Algebraic properties of the underlying stochastic process become algebraic laws of the process algebra. We obtain an analogue of the *expansion law* of untimed process algebra [24]:

$$(\alpha, r).P \parallel (\beta, s).Q = (\alpha, r).(P \parallel (\beta, s).Q) + (\beta, s).((\alpha, r).P \parallel Q)$$

only if the exponential distribution is used. Due to memorylessness, we do not need to adjust the rate s to take account of the time which elapsed during this occurrence of α (and analogously for r and β).

The strong equivalence relation over PEPA models is a congruence relation as is usual in process algebras and is a bisimulation in the style of Larsen and Skou [23]. It coincides with the Markov process notion of *lumpability* (a lumpable partition is the only partition of a Markov process which preserves the Markov property [20]). This correspondence makes a strong and unbreakable bond between the concise and elegant world of process algebras and the rich and beautiful theory of stochastic processes.

The fact that the strong equivalence relation is a semantics-preserving congruence has practical applications also. The relation can be used to aggregate the state space of a PEPA model, accelerating the production of numerical results and allowing larger modelling studies to be undertaken [16].

6.4.3 Reactions

As an example of how reactions are modelled, consider a simple single, reversible reaction, as illustrated in Fig. 6.1. This describes a reversible reaction between three proteins: $Prot1$, $Prot2$ and $Prot3$, with forward rate $k1$ and reverse rate $k2$.

In the forward reaction (from top to bottom), $Prot1$ and $Prot2$ are the producers, $Prot3$ is the consumer; in the backward reaction, the converse is true. Using this example, we illustrate how proteins and reactions are represented in PEPA.

Consider the coarsest discretisation. We refer to the two values as *high* and *low* and subscript protein processes by H and L, respectively. Thus, when there are n proteins, there are $2n$ equations. Assuming the forward reaction is called $r1$ and the reverse reaction $r2$, the equations are given in Fig. 6.2. We set the rate of consumers to be the passive rate $\top$.

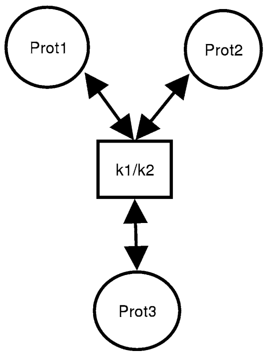


Fig. 6.1. Simple biochemical reaction

$$\begin{aligned} Prot1_H &\overset{def}{=} (r1, k1).Prot1_L & \quad Prot1_L &\overset{def}{=} (r2, \top).Prot1_H \\ Prot2_H &\overset{def}{=} (r1, k1).Prot2_L & \quad Prot2_L &\overset{def}{=} (r2, \top).Prot2_H \\ Prot3_H &\overset{def}{=} (r2, k2).Prot3_L & \quad Prot3_L &\overset{def}{=} (r1, \top).Prot3_H \end{aligned}$$

Fig. 6.2. Simple biochemical reaction in PEPA: model equations

$$Prot1_H \underset{\{r1,r2\}}{\bowtie} Prot2_H \underset{\{r1,r2\}}{\bowtie} Prot3_L$$

Fig. 6.3. Simple biochemical reaction in PEPA: model configuration

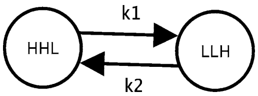


Fig. 6.4. Continuous time Markov chain (CTMC) for PEPA model of a simple biochemical reaction

The model configuration, given in Fig. 6.3, defines the (multi-way) synchronisation of the three processes. Note that initially, the producers are *high* and the consumer is *low*.

The model configuration defines a CTMC. Figure 6.4 gives a graphical representation of the underlying CTMC, with the labels of the states indicating protein values.

6.4.4 Pathways and Discretisation

A pathway involves many reactions, relating to each other in (typically) non-linear ways. In PEPA, pathways are expressed by defining alternate choices for protein behaviours using the + operator. Consider extending the simple example. Currently, *Prot*3 is a consumer in $r1$. If it was also the consumer in another reaction, say $r3$, then this would be expressed by the equation:

$$Prot3_L \overset{def}{=} (r1, \top).Prot3_H + (r3, \top).Prot3_H$$

It is possible to model with finer grained discretisations of species, for example, processes can be indexed by any countable set thus:

$$\begin{aligned} Prot1_N &\overset{def}{=} (r1, N * k1).Prot1_{N-1} \\ Prot1_{N-1} &\overset{def}{=} (r1, (N-1) * k1).Prot1_{N-2} + (r2, \top).Prot1_N \\ &\dots \\ Prot1_1 &\overset{def}{=} (r1, k1).Prot1_0 \qquad\qquad + (r2, \top).Prot1_2 \\ Prot1_0 &\overset{def}{=} \qquad\qquad\qquad\qquad\quad (r2, \top).Prot1_1 \end{aligned}$$

Note that the rates are adjusted to reflect the relative concentrations in different states/levels of the discretisation. N need not be fixed across the model, but can vary across proteins, depending on experimental evidence and measurement techniques.

In a model with two levels of discretisation, we specify non-passive rates (the known rate constant) for each occurrence of a reaction event in a producer process; since PEPA defines the rate of a synchronisation to be the rate of the slowest synchronising component, the rate for a given reaction will be exactly that rate constant. In the simple example, this means that initially, the three occurrences of $r1$ will synchronise, with rate $k1 = \min(k1, k1, \top)$.

Finally, we note that for any pathway, in the model configuration, the synchronisation sets must include the pairwise shared activities of all processes. In the example configuration shown in Fig. 6.3, the two synchronisation sets are identical. This is rarely the case in a pathway, where each protein is typically involved in a different set of reactions.

6.5 An Example: ERK Signalling Pathway

Cells are the fundamental structural and functional units of living organisms. We are interested in eukaryotic cells, which have an outer membrane and inner compartments including a nucleus. Cell signalling is the mechanism whereby cell behaviour is controlled and coordinated: signalling molecules are detected at the outer membrane of a cell, signals are conveyed to the nucleus via a pathway, and the cell makes a response.

Our example is one of the most important and ubiquitous cell signalling pathways: the ERK pathway [11] (also called Ras/Raf, or Raf-1/MEK/ERK pathway). This pathway is of great interest because abnormal functioning leads to uncontrolled growth and cancer. The overall basic behaviour of the pathway is that signals are conveyed to the nucleus through a "cascade" of proteins (protein kinases) called Raf, MEK and ERK. The simple cascade is illustrated in Fig. 6.5.

When the first protein Raf is activated by the incoming signal, the key subsequent activity is the conveyance of the signal along the pathway; in this case, the signal is conveyed by *phosphorylation*. Phosphorylation is the addition of a phosphate group to a protein. The source of the phosphoryl groups is adenosine triphosphate (ATP), which becomes adenosine diphosphate (ADP); fortunately, ATP is in such abundance that we do not need to consider it explicitly. The Raf, MEK and ERK proteins are called kinases

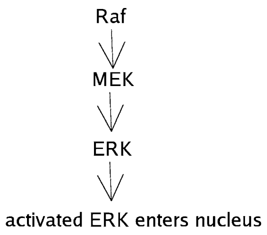


Fig. 6.5. Simplified ERK signalling pathway

because they transfer phosphoryl groups from ATP to each other. Proteins called phosphatases have the inverse function. Figure 6.6 gives a high level overview of the process of phosphorylation.

Phosphorylation brings about a change in protein activity; in the context of a pathway, phosphorylation/dephosphorylation represents on/off behaviour or the presence/absence of a signal. In the ERK pathway, the protein kinases become progressively phosphorylated, or activated, as the signal is conveyed to the nucleus.

A phosphorylated protein is indicated by a (single or double) suffix, for example, ERK-P is singly phosphorylated ERK, whereas ERK-PP is doubly phosphorylated ERK. Activated Raf is indicated by a "*" suffix.

The full behaviour of the ERK pathway is complex; here, we focus on a portion of pathway behaviour: how an additional protein, called Raf kinase inhibitor protein (RKIP), inhibits, or regulates, the signalling behaviour of the pathway. Namely, RKIP interferes with the cascade by reacting with activated Raf.

The effect of RKIP is the subject of ongoing research; we have based our model on the recent experimental results of Cho et al. [8].

A graphical representation of the pathway, taken from [8] (with a small modification, see Sect. 6.5.2), is given in Fig. 6.7. Each node is labelled by a

Fig. 6.6. Signalling

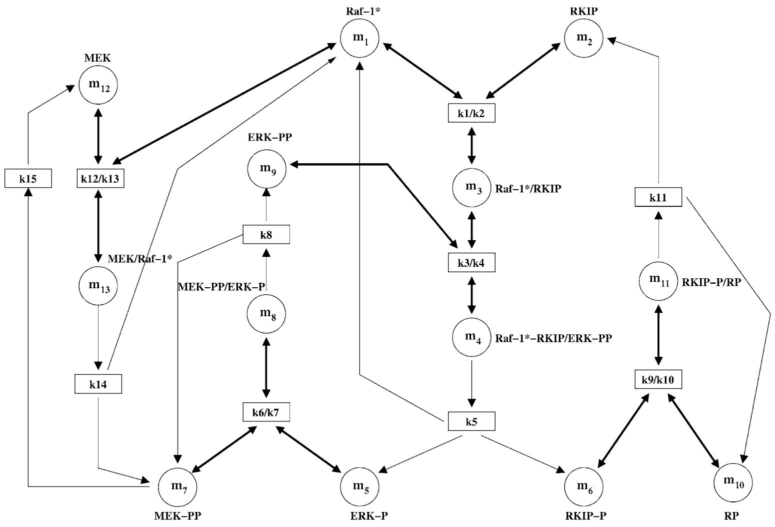


Fig. 6.7. Raf kinase inhibitor protein (RKIP) inhibited ERK pathway

protein *species*, that is, a molar concentration. The particular form of Raf in this investigation is called Raf-1; MEK, ERK and RKIP are as described above, and RP is an additional protein phosphatase. There are numerous additional complex proteins built up from simpler ones; the complexes are named after their constituents. For example, Raf-1*/RKIP is a protein built up from Raf-1* and RKIP.

In Fig. 6.7, species concentrations for Raf-1*, etc. are given by the variables $m1$, etc. Initially, all concentrations are unobservable, except for m_1, m_2, m_7, m_9 and m_{10} [8]. Note that in this pathway, not all reactions are reversible; the non-reversible reactions are indicated by uni-directional arrows.

6.5.1 PEPA Model

Figure 6.8 gives the PEPA equations for the coarsest discretisation of the pathway, with the model configuration in Fig. 6.9.

There are two equations for each protein, for high and low behaviour. The pathway is highly non-linear, as indicated by numerous occurrences of the choice operator. For example, when Raf-1* is high, it can react with RKIP in the $k1react$ reaction *or* it can react with MEK in the $k12react$ reaction. Some choices simply reflect the reversible nature of some reactions, for example, RKIP is replenished by the $k2react$ reaction, which is the reverse of $k1react$.

Unlike our simple reaction example, in the pathway model, the synchronisation sets in the model configuration are all distinct. Although we derived the synchronisation sets by hand, it would be possible to do so algorithmically, by inspection of the model equations.

6.5.2 Analysis

There are two principal reasons to apply formal languages to describe systems and processes. The first is the avoidance of ambiguity in the description of the problem under study. The second, but not less important, is that formal languages are amenable to automated processing by software tools. We used the PEPA Workbench [15] to analyse the model.

First, we used the Workbench to test for deadlocks in the model. A deadlocked system is one which cannot perform any activities (in many process algebras, this is denoted by a constant such as *exit* or *stop*). In our context, signalling pathways should not be able to deadlock, this would indicate a malfunction. Initially, there were several deadlocks in our PEPA model: this is how we discovered an incompleteness in the published description of [8], with respect to the treatment of MEK. After correspondence with the authors, we were able to correct the omission and develop a more complete and deadlock-free model.

Second, when we had a deadlock-free model, we used the Workbench to generate the CTMC (28 states) and its long-run probability distribution. The steady-state probability distribution is obtained using a number of routines

$$\begin{aligned}
\text{Raf-1}^*_{\text{H}} &\overset{def}{=} (\textit{k1react}, k_1).\text{Raf-1}^*_{\text{L}} + (\textit{k12react}, k_{12}).\text{Raf-1}^*_{\text{L}}\\
\text{Raf-1}^*_{\text{L}} &\overset{def}{=} (\textit{k5product}, \top).\text{Raf-1}^*_{\text{H}} + (\textit{k2react}, \top).\text{Raf-1}^*_{\text{H}}\\
&\quad + (\textit{k13react}, \top).\text{Raf-1}^*_{\text{H}} + (\textit{k14product}, \top).\text{Raf-1}^*_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{RKIP}_{\text{H}} &\overset{def}{=} (\textit{k1react}, k_1).\text{RKIP}_{\text{L}}\\
\text{RKIP}_{\text{L}} &\overset{def}{=} (\textit{k11product}, \top).\text{RKIP}_{\text{H}} + (\textit{k2react}, \top).\text{RKIP}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{MEK}_{\text{H}} &\overset{def}{=} (\textit{k12react}, k_{12}).\text{MEK}_{\text{L}}\\
\text{MEK}_{\text{L}} &\overset{def}{=} (\textit{k13react}, \top).\text{MEK}_{\text{H}} + (\textit{k15product}, \top).\text{MEK}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{MEK/Raf-1}^*_{\text{H}} &\overset{def}{=} (\textit{k14product}, k_{14}).\text{MEK/Raf-1}^*_{\text{L}}\\
&\quad + (\textit{k13react}, k_{13}).\text{MEK/Raf-1}^*_{\text{L}}\\
\text{MEK/Raf-1}^*_{\text{L}} &\overset{def}{=} (\textit{k12react}, \top).\text{MEK/Raf-1}^*_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{MEK-PP}_{\text{H}} &\overset{def}{=} (\textit{k6react}, k_6).\text{MEK-PP}_{\text{L}} + (\textit{k15product}, k_{15}).\text{MEK-PP}_{\text{L}}\\
\text{MEK-PP}_{\text{L}} &\overset{def}{=} (\textit{k8product}, \top).\text{MEK-PP}_{\text{H}} + (\textit{k7react}, \top).\text{MEK-PP}_{\text{H}}\\
&\quad + (\textit{k14product}, \top).\text{MEK-PP}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{ERK-PP}_{\text{H}} &\overset{def}{=} (\textit{k3react}, k_3).\text{ERK-PP}_{\text{L}}\\
\text{ERK-PP}_{\text{L}} &\overset{def}{=} (\textit{k8product}, \top).\text{ERK-PP}_{\text{H}} + (\textit{k4react}, \top).\text{ERK-PP}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{ERK-P}_{\text{H}} &\overset{def}{=} (\textit{k6react}, k_6).\text{ERK-P}_{\text{L}}\\
\text{ERK-P}_{\text{L}} &\overset{def}{=} (\textit{k5product}, \top).\text{ERK-P}_{\text{H}} + (\textit{k7react}, \top).\text{ERK-P}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{MEK-PP/ERK-P}_{\text{H}} &\overset{def}{=} (\textit{k8product}, k_8).\text{MEK-PP/ERK-P}_{\text{L}}\\
&\quad + (\textit{k7react}, k_7).\text{MEK-PP/ERK-P}_{\text{L}}\\
\text{MEK-PP/ERK-P}_{\text{L}} &\overset{def}{=} (\textit{k6react}, \top).\text{MEK-PP/ERK-P}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{Raf-1}^*\text{/RKIP}_{\text{H}} &\overset{def}{=} (\textit{k3react}, k_3).\text{Raf-1}^*\text{/RKIP}_{\text{L}}\\
&\quad + (\textit{k2react}, k_2).\text{Raf-1}^*\text{/RKIP}_{\text{L}}\\
\text{Raf-1}^*\text{/RKIP}_{\text{L}} &\overset{def}{=} (\textit{k1react}, \top).\text{Raf-1}^*\text{/RKIP}_{\text{H}}\\
&\quad + (\textit{k4react}, \top).\text{Raf-1}^*\text{/RKIP}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{Raf-1}^*\text{/RKIP/ERK-PP}_{\text{H}} &\overset{def}{=} (\textit{k5product}, k_5).\text{Raf-1}^*\text{/RKIP/ERK-PP}_{\text{L}}\\
&\quad + (\textit{k4react}, k_4).\text{Raf-1}^*\text{/RKIP/ERK-PP}_{\text{L}}\\
\text{Raf-1}^*\text{/RKIP/ERK-PP}_{\text{L}} &\overset{def}{=} (\textit{k3react}, \top).\text{Raf-1}^*\text{/RKIP/ERK-PP}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{RKIP-P}_{\text{H}} &\overset{def}{=} (\textit{k9react}, k_9).\text{RKIP-P}_{\text{L}}\\
\text{RKIP-P}_{\text{L}} &\overset{def}{=} (\textit{k5product}, \top).\text{RKIP-P}_{\text{H}} + (\textit{k10react}, \top).\text{RKIP-P}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{RP}_{\text{H}} &\overset{def}{=} (\textit{k9react}, k_9).\text{RP}_{\text{L}}\\
\text{RP}_{\text{L}} &\overset{def}{=} (\textit{k11product}, \top).\text{RP}_{\text{H}} + (\textit{k10react}, \top).\text{RP}_{\text{H}}
\end{aligned}$$

$$\begin{aligned}
\text{RKIP-P/RP}_{\text{H}} &\overset{def}{=} (\textit{k11product}, k_{11}).\text{RKIP-P/RP}_{\text{L}}\\
&\quad + (\textit{k10react}, k_{10}).\text{RKIP-P/RP}_{\text{L}}\\
\text{RKIP-P/RP}_{\text{L}} &\overset{def}{=} (\textit{k9react}, \top).\text{RKIP-P/RP}_{\text{H}}
\end{aligned}$$

Fig. 6.8. PEPA model definitions for the reagent-centric model

$$
\begin{aligned}
&(\text{Raf-1}^*_{\text{H}} \bowtie_{\{\mathit{k1react},\mathit{k2react},\mathit{k12react},\mathit{k13react},\mathit{k5product},\mathit{k14product}\}}\\
&\quad(\text{RKIP}_{\text{H}} \bowtie_{\{\mathit{k1react},\mathit{k2react},\mathit{k11product}\}}\\
&\quad\quad(\text{Raf-1}^*/\text{RKIP}_{\text{L}} \bowtie_{\{\mathit{k3react},\mathit{k4react}\}}\\
&\quad\quad\quad(\text{Raf-1}^*/\text{RKIP/ERK-PP}_{\text{L}}) \bowtie_{\{\mathit{k3react},\mathit{k4react},\mathit{k5product}\}}\\
&\quad\quad\quad\quad(\text{ERK-P}_{\text{L}} \bowtie_{\{\mathit{k5product},\mathit{k6react},\mathit{k7react}\}}\\
&\quad\quad\quad\quad\quad(\text{RKIP-P}_{\text{L}} \bowtie_{\{\mathit{k9react},\mathit{k10react}\}}\\
&\quad\quad\quad\quad\quad\quad(\text{RKIP-P/RP}_{\text{L}} \bowtie_{\{\mathit{k9react},\mathit{k10react},\mathit{k11product}\}}\\
&\quad\quad\quad\quad\quad\quad\quad(\text{RP}_{\text{H}} \parallel\\
&\quad\quad\quad\quad\quad\quad\quad\quad(\text{MEK}_{\text{L}} \bowtie_{\{\mathit{k12react},\mathit{k13react},\mathit{k15product}\}}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad(\text{MEK/Raf-1}^*_{\text{L}} \bowtie_{\{\mathit{k14product}\}}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad(\text{MEK-PP}_{\text{H}} \bowtie_{\{\mathit{k8product},\mathit{k6react},\mathit{k7react}\}}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad(\text{MEK-PP/ERK-P}_{\text{L}} \bowtie_{\{\mathit{k8product}\}}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad(\text{ERK-PP}_{\text{H}})))))))))))
\end{aligned}
$$

Fig. 6.9. PEPA model configuration for the reagent-centric model

from numerical linear algebra. The distribution varies as the rates associated with the activities of the PEPA model are varied, so the solution of the model is relative to a particular assignment of the rates. Initially, we set all rates to unity (1.0). Our main aim was to investigate how RKIP affects the production of phosphorylated ERK and MEK, i.e., if it reduces the probability of having a high level of phosphorylated ERK or MEK.

We used the PEPA state-finder to aggregate the probabilities of all states when ERK-PP is high or low for a given set of rates. That is, it aggregated the probabilities of states whose (symbolic) description has the form $* \bowtie$ ERK-PP$_{\text{H}}$ where $*$ is a wildcard standing for any expression. We then repeated this with a different set of rates and compared results. We observed that the probability of being in a state with ERK-PP$_{\text{H}}$ *decreases* as the rate $k1$ is increased, and the converse for ERK-PP$_{\text{L}}$ *increases.* For example, when $k1 = 1$, the probability of ERK-PP$_{\text{H}}$ is 0.257, when $k1 = 100$, it drops to 0.005. We can also plot throughput (rate $\times$ probability) against rate. Figures 6.10 and 6.11 show two sub-plots which detail the effect of increasing the rate $k1$ on the $k14$*product* and $k8$*product* reactions – the production of (doubly) phosphorylated MEK and (doubly) phosphorylated ERK, respectively. These are obtained by scaling $k1$, keeping all other rates to be unity. The graphs show that increasing the rate of the binding of RKIP to Raf-1* dampens down the $k14$*product* and $k8$*product* reactions, and they quantify this information. The efficiency of the reduction is greater in the former case; the graph falls away more steeply. In the latter case, the reduction is more gradual and the throughput of $k8$*product* peaks at $k1 = 1$. Note that since $k5$*product* is on the same (sub)pathway as $k8$*product*, both ERK-PP and ERK-P are similarly affected. Thus, we conclude that the rate at which RKIP binds to

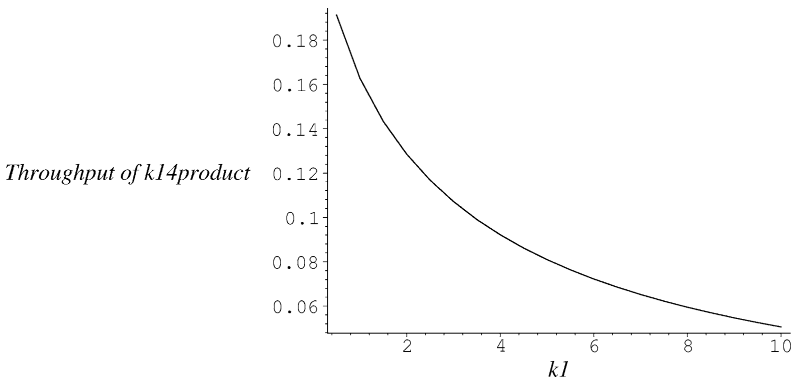


Fig. 6.10. Plotting the effect of $k1$ on $k14$*product*

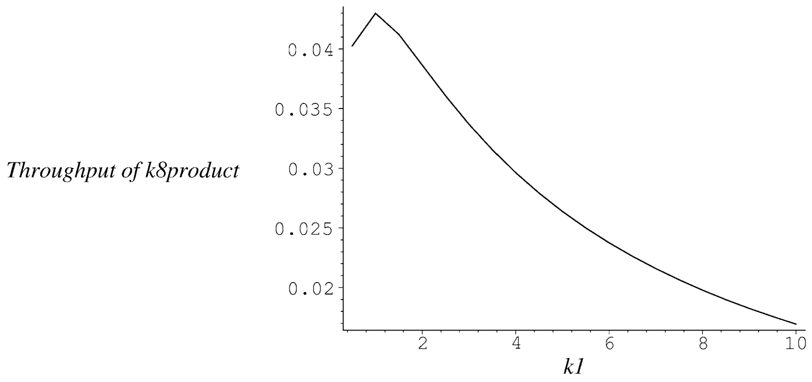


Fig. 6.11. Plotting the effect of $k1$ on $k8$*product*

Raf-1* (thus suppressing phosphorylation of MEK) affects the ERK pathway, as predicted (and observed); RKIP does indeed regulate the ERK pathway.

In the next section, we give an overview of traditional pathway models and how they relate to our approach.

6.6 Modelling Pathways with Differential Equations

The algebraic formulation of the PEPA model makes clear the interactions between the pathway components. There is a direct correspondence between topology and the model, models are easy to derive and to alter. This is not apparent in the traditional pathway models given by sets of ODEs. In these models, equations define how the concentration of each species varies over time, according to mass action kinetics. There is one equation for each protein species. The overall rate of a reaction depends on both a rate (constant) and the concentration masses. Both time and concentration variables are continuous. ODEs do not give an indication of the structure or topology of the pathway, and consequently, the process to define them is often error prone. Set against this, efficient numerical methods are available for the numerical integration of ODEs even in the difficult quantitative setting of chemically reacting systems, which are almost always stiff due to the presence of widely differing timescales in the reaction rates.

Fortunately, the ODEs can be derived from PEPA models – in fact, from models which distinguish only the coarsest discretisation of concentration. The high/low discretisation is sufficient because we need to know only when a reaction *increases* or *decreases* the concentration of a species. Moreover, the PEPA expressions reveal which species are required in order to complete an activity.

An example illustrates the relationship between ODEs and the PEPA model. In the PEPA equations for the ERK pathway, we can easily observe that Raf-1* increases (low to high, second equation) with rates (from synchronisations) k_5, k_2, k_{13} and k_{14}; it decreases (high to low, first equation) with rates k_1 and k_{12}. Mass action kinetics shows how the rates and masses affect the amount of Raf-1*. The equation for Raf-1*, given in terms of the (continuous) concentration variables m_1, etc., is

$$\frac{\mathrm{d}m_1}{\mathrm{d}t} = (k_5 \cdot m_4) + (k_2 \cdot m_3) + (k_{13} \cdot m_{13}) + (k_{14} \cdot m_{13}) - (k_1 \cdot m_1 \cdot m_2) - (k_{12} \cdot m_1 \cdot m_{12}) \tag{6.1}$$

This equation defines the change in Raf-1* by how it is *increased*, i.e., the positive terms, and how it is *decreased*, i.e., the negative terms. These differential equations can be derived directly and automatically from the PEPA model. Algorithms to do so are given in [4].

While the style of modelling in the stochastic process algebra approach embodied by PEPA is concise and elegant, the bounded capacity, min style semantics for synchronisation means we have not been able to represent accurate rates for mass action kinetics. For example, in a PEPA model for a reaction between two producers and one consumer, the overall rate of a synchronised transition is the *minimum* of the three given rates. In mass action kinetics, the rate would be a (function of the) *product* of the given rates. We could overcome this by defining a very large number of constants, representing every possible rate × mass product combination; alternatively, we turn to the PRISM high level language for CTMCs, which implements synchronisation rates by products. This formalism, which is state-based, is less concise for modelling, but it affords additional analysis by way of model checking properties expressed using CSL.

6.7 Modelling Pathways in PRISM

The PRISM language [21] represents systems using an imperative language of reactive modules. It has the capability to express a wide range of stochastic process models including both discrete- and continuous-time Markov chains and Markov decision processes. A key feature is multi-way synchronisation, essential for our approach. In (CTMC) PRISM models, activities are called transitions. (Note, the PRISM name denotes both a modelling language and the model checker, the intended meaning should be clear from the context.) These correspond directly to CTMC transitions and they are labelled with performance rates and (optional) names. For each transition, like PEPA, the rate is defined as the parameter of an exponential distribution of the transition duration. PRISM is state-based; *modules* play the role of PEPA processes and define how state variables are changed by transitions. Like PEPA, transitions with common names are synchronised; transitions with distinct names are not synchronised.

6.7.1 Reactions

Similar to our PEPA models, proteins are represented by PRISM modules and reactions are represented by transitions. Below, we give a brief overview of the language, illustrating each concept with reference to the simple reaction example from Fig. 6.1; the reader is directed to [21] for further details of PRISM.

The PRISM model for the simple example is given in Fig. 6.12. The first thing to remark is that in the PRISM model, the discretisation of concentration is an explicit parameter, denoted by N. In this example, we set it to 3. K is simply a convenient abbreviation for N^{-1}.

Second, consider the first three modules which represent the proteins $Prot1$, $Prot2$ and $Prot3$. Each module has the form: a state variable which denotes the protein concentration (we use the same name for process and variable, the type can be deduced from context) followed by a non-deterministic choice of transitions named $r1$ and $r2$. A transition has the form *precondition* $\rightarrow$ *rate: assignment*, meaning when the precondition is true, then perform the assignment at the given *rate,* i.e., rate is the parameter of an exponential distribution of the transition duration. The assignment defines the value of a state variable *after* the transition. The new value of a state variable follows the usual convention – the variable decorated with a single quote.

In this model, the transition rates have been chosen carefully, to correspond to mass action kinetics. Namely, when the transition denotes consumer behaviour (decrease protein by 1), the protein is multiplied by K, and when the transition denotes producer behaviour (increase protein by 1), the rate is simply 1. These rates correspond to the fact that in mass action kinetics, the overall rate of the reaction depends on a rate constant and the

```
const int N = 3;
const double K = 1/N;

module Prot1
    Prot1: [0..N] init N;
    [r1] (Prot1>0) -> Prot1*K: (Prot1' = Prot1 - 1);
    [r2] (Prot1<N) -> 1: (Prot1' = Prot1 + 1);
endmodule

module Prot2
    Prot2: [0..N] init N;
    [r1] (Prot2>0) -> Prot2*K: (Prot2' = Prot2 - 1);
    [r2] (Prot2<N) -> 1: (Prot2' = Prot2 + 1);
endmodule

module Prot3
    Prot3: [0..N] init 0;
    [r1] (Prot3 < N) -> 1: (Prot3' = Prot3 + 1);
    [r2] (Prot3>0) -> Prot2*K: (Prot3' = Prot3 - 1);
endmodule

module Constants
    x: bool init true;
    [r1] (x=true) -> k1*N: (x'=true);
    [r2] (x=true) -> k2*N: (x'=true);

endmodule
```

Fig. 6.12. Simple biochemical reaction in PRISM: modules

concentrations of the reactants *consumed* in the reaction. (We will discuss this further in Sect. 6.7.2.) Note that unlike PEPA where the processes are recursive, here PRISM modules describe the circumstances under which transitions can occur.

The fourth module, *Constants*, simply defines the constants for reaction kinetics. These were obtained from experimental evidence [8]. The module contains a "dummy" state variable called x, and (always) enabled transitions which define the rates.

The four modules run concurrently, as given by the system description in Fig. 6.13. In PRISM, the rate for the synchronised transition is the *product* of the rates of the synchronising transitions. For example, if process A performs α with rate λ_1, and process B performs α with rate λ_2, then the rate of α when A is synchronised with B is $\lambda_1 \cdot \lambda_2$. To illustrate how this determines the rates in the underlying CTMC, consider the first reaction from the initial state of the example, i.e., reaction $r1$. There are four enabled (i.e., precondition is

```
system
   Prot1 || Prot2 || Prot3 || Constants
endsystem
```

Fig. 6.13. Simple biochemical reaction in PRISM: system

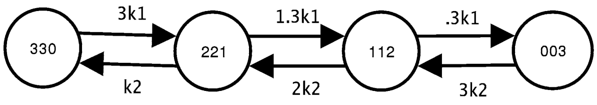


Fig. 6.14. CTMC for PRISM model of simple biochemical reaction

true) transitions with the name $r1$. They will all synchronise, and when they do, the resulting transition has rate

$$(Prot1 \cdot K) \cdot (Prot2 \cdot K) \cdot 1 \cdot (k1 \cdot N) \tag{6.2}$$

Since $Prot1$ and $Prot2$ are initialised to N ($Prot3$ is initialised to 0), this equates to

$$(N \cdot K) \cdot (N \cdot K) \cdot 1 \cdot (k1 \cdot N) = k1 \cdot N \tag{6.3}$$

The reaction $r1$ can occur N times, until all the $Prot1$ and $Prot2$ have been consumed. Figure 6.14 gives a graphical representation of the underlying CTMC when $N = 3$. Again, the state labels indicate protein values, i.e., $x_1 x_2 x_3$ denotes the state where $Prot1 = x_1$, $Prot2 = x_2$, etc. Note that the transition rates decrease as the amount of producer decreases.

6.7.2 Reaction Kinetics

In this section, we show how the PRISM model implements mass action kinetics. Consider the mass action kinetics for $Prot3$ in the simple example, given by the ODE

$$\frac{\mathrm{d}m_3}{\mathrm{d}t} = (k_1 \cdot m_1 \cdot m_2) - (k_2 \cdot m_3) \tag{6.4}$$

The variables m_1, etc. are continuous, denoting concentrations of $Prot1$, etc. Integrating (6.4) by the simplest method, Euler's method, defines a new value for m_3 thus:

$$m_3' = m_3 + (k1 \cdot m_1 \cdot m_2 \cdot \Delta t) - (k_2 \cdot m_3 \cdot \Delta t) \tag{6.5}$$

In our discretisation, concentrations can only increase in units of one molar concentration, i.e., $1/N$, so

$$\Delta t = \frac{1}{N \cdot ((k1 \cdot m_1 \cdot m_2) - (k_2 \cdot m_3))} \tag{6.6}$$

Recall that PRISM implements rates as the memoryless negative exponential, that is, for a given rate λ, $P(t) = 1 - \mathrm{e}^{-\lambda t}$ is the probability that the action will be completed before time t. Taking λ as $1/\Delta t$, in this example, we have

$$\lambda = (N \cdot k1 \cdot m_1 \cdot m_2) - (N \cdot k_2 \cdot m_3) \tag{6.7}$$

The continuous variables m_1, etc. relate to the PRISM variables $Prot1$, etc. as follows:

$$m_1 = Prot1 \cdot K \tag{6.8}$$

$$m_2 = Prot2 \cdot K \tag{6.9}$$

etc.

So, substituting into (6.7) yields

$$\lambda = (N \cdot k1 \cdot (Prot1 \cdot K) \cdot (Prot2 \cdot K)) - (N \cdot k_2 \cdot (Prot3 \cdot K)) \tag{6.10}$$

Given the initial concentrations of $Prot1$, $Prot2$ and $Prot3$, this equates to

$$\lambda = (N \cdot k1 \cdot (N \cdot K) \cdot (N \cdot K)) - (N \cdot k_2 \cdot (0 \cdot K)) \tag{6.11}$$

which simplifies to the rate specified in the PRISM model, (6.3) in Sect. 6.7.1.

Comparison of ODE and CTMC Models

It is important to note that the ODE model is deterministic, whereas our CTMC models are stochastic. How do we compare the two? We investigated simulation traces of both, over 200 data points in the time interval [1 . . . 100], using MATLAB for the former. While PRISM is not designed for simulation, we were able to derive simulation traces, using the concept of *rewards* (see [21]).

When comparing the two sets of traces, the accuracy of the CTMC traces depends on the choice of value for N. Intuitively, as N approaches infinity, the stochastic (CTMC) and deterministic (ODE) models will converge. For many pathways, including our example pathway, N can be surprisingly small (e.g. 7 or 8) to yield very good simulations in reasonable time (few minutes) on a state of art workstation. The two are indistinguishable for practical purposes. More details about simulation results and comparison between the stochastic and deterministic models are given in [6].

One advantage of our approach is that the modeller chooses the granularity of N. Usually this will depend on the accuracy of and confidence in experimental data or knowledge. In many cases, it is sufficient to use a high/low model, particularly when the data or knowledge is incomplete or very uncertain.

The full PRISM model for the example pathway is given in the Appendix. We now turn our attention to the analysis of the example pathway using a temporal logic.

6.7.3 Analysis of Example Pathway Using the PRISM Model Checker

Temporal logics are powerful tools for expressing properties which may be generic, such as state reachability, or application specific in which case they represent application characteristics. Here, we concentrate on the latter, specifically considering properties of biological significance.

The two properties we consider are: what is the probability that a protein concentration reaches a certain level and then remains at that level thereafter, and what is the probability that one protein "peaks" before another? The former is referred to as *stability* (i.e., the protein is stable), and the latter as *activation sequence.*

Since we have a stochastic model, we employ the logic CSL (see Sect. 6.2.2) and the symbolic probabilistic model checker PRISM [26] to compute steady state solutions and check validity. Using PRISM we can analyse open formulae, i.e., we can perform *experiments* as we vary instances of variables in a formula expressing a property. Typically, we will vary reaction rates or concentration levels. We consider two properties below, the first is a steady state property and we vary a reaction rate, and the second is a transient property and we vary a concentration. All properties were checked within a few minutes on a state of art workstation; hence, run times are omitted.

Protein Stability

Stability properties are useful during model fitting, i.e., fitting the model to experimental data. As an example, consider the stability of Raf-1* as the reaction rate $k1$ (the rate of $r1$ which binds Raf-1* and RKIP) varies over the interval $[0\ldots1]$. Let stability in this case be defined as concentration 2 or 3. The stability property is expressed by:

$$S_{=?}[(\text{Raf-1}^* \geq 2) \wedge (\text{Raf-1}^* \leq 3)] \tag{6.12}$$

Now consider the probability that Raf-1* is stable at concentrations 0 and 1; the formula for this is:

$$S_{=?}[(\text{Raf-1}^* \geq 0) \wedge (\text{Raf-1}^* \leq 1)] \tag{6.13}$$

Figure 6.15 gives results for both these properties, when $N = 5$. From the graph, we can see that the likelihood of property (6.12) (solid line) is greatest when $k1 = 0.03$ and then it decreases; the likelihood of property (6.13) (dashed line) increases dramatically, becoming very likely when $k1 > 0.4$.

We note that the analysis presented in Sect. 6.5.2 is for stability. For example, assuming $N = 1$, the probability that ERK-PP is high would be expressed in PRISM by $S_{=?}[\text{ERK-PP} \geq 1)]$.

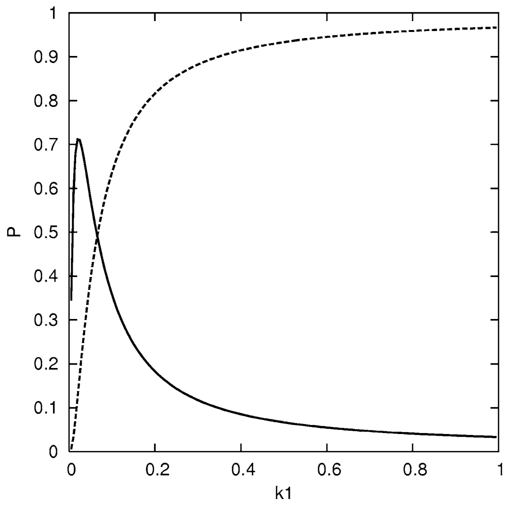


Fig. 6.15. Stability of Raf-1* at levels {2,3} and {0,1}

Activation Sequence

As an example of activation sequence, consider the two proteins Raf-1*/RKIP and Raf-1*/RKIP/ERK-PP, and their two peaks C and M, respectively. Is it possible that the (concentration of the) former peaks before the latter? This property is given by:

$$P_{=?}[(\text{Raf-1}^*/\text{RKIP/ERK-PP} < M)\ \mathbf{U}\ (\text{Raf-1}^*/\text{RKIP} = C)] \qquad (6.14)$$

The results for C ranging over $0, 1, 2$ and M ranging over $1 \dots 5$ are given in Fig. 6.16: the line with steepest slope represents $M = 1$, and the line which is nearly horizontal is $M = 5$. For example, the probability Raf-1*/RKIP reaches concentration level 2 before Raf-1*/RKIP/ERK-PP reaches concentration level 5 is more than 99%, the probability Raf-1*/RKIP reaches concentration level 2 before RAF1/RKIP/ERK-PP reaches concentration level 2 is almost 96%.

6.7.4 Further Properties

Examples of further temporal properties concerning the accumulation (or diminution) of proteins illustrate the use of bounds. Full details of their analysis can be found in [6].

The (accumulation) property

$$P_{=?}[(\mathit{true})\ \mathbf{U}^{\leq 120}\ (\mathit{Protein} > C)\{(\mathit{Protein} = C)\}] \qquad (6.15)$$

expresses the possibility that *Protein* can reach a level higher than C within a time bound, once it has reached concentration C.

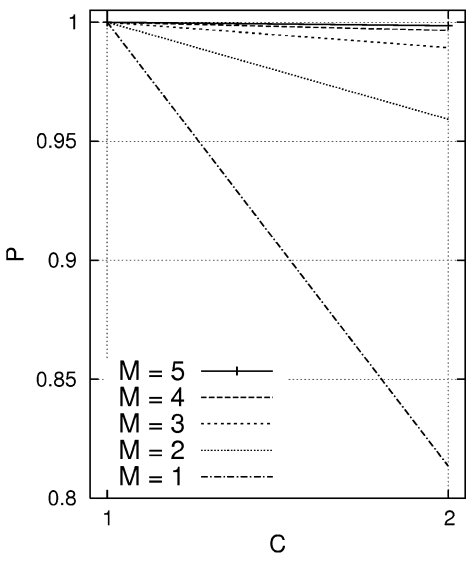


Fig. 6.16. Activation sequence

The (diminution) property

$$P_{\geq 1}[(true)\ \mathbf{U}\ ((Protein = C) \wedge (P_{\geq 0.95}[\mathbf{X}(Protein = C - 1)]))] \qquad (6.16)$$

expresses the high likelihood of decreasing *Protein*, i.e., the concentration reaches C, and after the next step, it is very likely to be $C - 1$.

6.8 Discussion

Modelling biochemical signalling pathways has previously been carried out using sets of non-linear ODEs or stochastic simulation based on Gillespie's algorithm. These can be seen as contrasting approaches in several respects. The ODE models are deterministic and present a *population* view of the system. This aims to characterise the average behaviour of large numbers of individual molecules of each species interacting, capturing only their concentration. Alternatively, in Gillespie's approach, each molecule is modelled explicitly and stochastically, capturing the probabilities with which reactions occur, based on the likelihood of molecules of appropriate species being in close proximity. This gives rise to a CTMC, but one whose state space is much too large to be solved explicitly. Hence, simulation is the only option, each realisation of the simulation giving rise to one possible behaviour of the system. Thus, the results of many runs must be aggregated in order to gain insight into the typical behaviour.

Our approach represents a new alternative that develops a representation of the behaviour of the system, which is intermediate between the previous techniques. We retain the stochastic element of Gillespie's approach, but the CTMC that we give rise to can be considerably smaller because we model at the level of *species* rather than *molecules*. Keeping the state space manageable means that we are able to solve the CTMC explicitly and avoid the repeated runs necessitated by stochastic simulation. Moreover, in addition to the quantitative analysis on the CTMC, as illustrated here with PEPA, we are able to conduct model checking of stochastic properties of the model. This provides more powerful reasoning mechanisms than stochastic simulation.

In our models, the continuous variable, or *concentration*, associated with each species is discretised into a number of *levels*. Thus, each component representing a species has a distinct local state for each level of concentration. The more levels that are incorporated into the model, i.e., the finer the granularity of the discretisation, the closer the results of the CTMC will be to the ODE model. However, finer granularity also means that there will be more states in the CTMC. Thus, we are faced with a trade-off between accuracy and tractability. Since not all species must have the same degree of discretisation, we may choose to represent some aspects of the pathway in finer detail than others.

6.8.1 Scalability

Whilst being of manageable size from a solution perspective, the CTMCs we are dealing with are too large to contemplate constructing manually. The use of high level modelling languages such as PEPA and PRISM to generate the underlying CTMC allows us to separate system structure from performance. Our style of modelling, focussed on species or molar concentrations thereof, rather than molecules, means that most reactions involve three or more components. The multi-way synchronisation of PEPA and PRISM is ideally suited to this approach. Ultimately, we will encounter state space explosion, arising from either the granularity of the discretisation or the number of species, but it has not been a problem for the pathways we have studied thus far (with up to approx. 20 species).

6.8.2 Relationship Between PEPA and PRISM

PEPA and PRISM have provided complementary formalisms and tool sets for modelling and reasoning with CTMCs. While models are easily and clearly expressed in PEPA, it is difficult to represent reaction rates accurately. This is not surprising, given PEPA was designed for modelling performance of (bounded capacity) computer systems, not biochemical reactions. PRISM provides a better representation of reaction rates and the facility to check CSL properties. A key feature of both languages is multi-way synchronisation, essential for our approach. To some extent the source language is not important, since a PRISM model can be derived automatically from any PEPA model, using the PEPA workbench, though it is necessary to handcode the rates (because PEPA implements synchronisation by minimum, PRISM by product). Here, we have handcoded the PRISM models to make the concentration variable explicit. This has enabled us to perform experiments easily over a wide range of models.

6.9 Related and Further Work

Work on applying formal system description techniques from computer science to biochemical signalling pathways was initially stimulated by [17, 27–29]. Subsequently there has been much work in which the stochastic π-calculus is used to model biological systems, for example, [7] and elsewhere. This work is based on a correspondence between molecules and processes. Each *molecule* in a signalling pathway is represented by a component in the process algebra representation. Thus, in order to represent a system with populations of molecules, many copies of the process algebra components are needed. This leads to underlying CTMC models with enormous state spaces – the only possible solution technique is simulation based on Gillespie's algorithm.

In our approach, we have proposed a more abstract correspondence between *species* and processes (cf. modelling classes rather than individual objects). Now the components in the process algebra model capture a pattern of behaviour of a whole set of molecules, rather than the identical behaviour of thousands of molecules having to be represented individually. From such models, we are able to generate underlying models, suitable for analysis, in a number of different ways. When we consider populations of molecules, considering only two states for each species (*high* and *low*), we are able to generate a set of ODEs from a PEPA model. With a moderate degree of granularity in the discretisation of the concentration, we are able to generate an underlying CTMC explicitly. This can then be subjected to steady state or transient numerical analysis, or model checking of temporal properties expressed in CSL, as we have seen. Alternatively, interpreting the high/low model as establishing a pattern of behaviour to be followed by each molecule, we are able to derive a stochastic simulation based on Gillespie's algorithm.

In the recent work by Heath et al. [18, 22], the authors use PRISM to model the FGF signalling pathway. However, they model individuals and do not appear to have a representation of population dynamics.

6.10 Conclusions

Mathematical biologists are familiar with applying methods based on reaction rate equations and systems of coupled first-order differential equations. They are familiar too with the stochastic simulation methods in the Gillespie family, which have their roots in physically rigorous modelling of the phenomena studied in statistical thermodynamics. However, the practice in the field of computational biology is often either to code a system of differential equations directly in a numerical computing platform such as Matlab, or to run a stochastic simulation.

It might be thought that differential equations represent a direct mathematical formulation of a chemical reacting system and might be more straightforward to use than mathematical formulations derived from process algebras. Set against this, though, is the absence of a ready apparatus for reasoning about the correctness of an ODE model. No equivalence relations exist to compare models and there is no facility to perform even simple checks such as deadlock detection, let alone more complex static analysis such as liveness or reachability analysis. The same criticisms unfortunately can also be levelled at stochastic simulation.

We might like to believe that there is now sufficient accumulated expertise in computational biological modelling with ordinary differential equations that such mistakes would simply not occur, or we might think that they would be so subtle that modelling in a process algebra such as PEPA or a state-based modelling language such as PRISM could not uncover them. We can, however, point to at least one counterexample to this. In a recent PEPA modelling study, we found an error in the analysis of a published and widely cited ODE model. The authors of [30] develop a complex ODE model of epidermal growth factor (EGF) receptor signal pathways in order to give insight into the activation of the MAP kinase cascade through the kinases Raf, MEK and ERK-1/2. Our formalisation in [3] was able to uncover a previously unexpected error in the way the ODEs had been solved, which led to the production of misleading results. In essence, the error emerged because through the use of a high-level language, we were able to compare different analyses of the same model, and then do some investigations to discover the cause of discrepencies between them.

High-level modelling languages rooted in computer science theory add significantly to the analysis methods that are presently available to practicing computational biologists, increasing the potential for stronger and better modelling practice leading to beneficial scientific discoveries by experimentalists making a positive contribution to improving human and animal health and quality of life. We believe that the insights obtained through the principled application of strong theoretical work stand as a good advertisement for the usefulness of high-level modelling languages for analysing complex biological processes.

Appendix: PRISM Model of Example Pathway

The system description is omitted – it simply runs all modules concurrently. The rate constants are taken from [8].

```
const int N = 7;
const double M = 2.5/N;
```

```
module RAF1
    RAF1: [0..N] init N;
    [r1]  (RAF1 > 0) -> RAF1*M: (RAF1' = RAF1 - 1);
    [r12] (RAF1 > 0) -> RAF1*M: (RAF1' = RAF1 - 1);
    [r2]  (RAF1 < N) -> 1: (RAF1' = RAF1 + 1);
    [r5]  (RAF1 < N) -> 1: (RAF1' = RAF1 + 1);
    [r13] (RAF1 < N) -> 1: (RAF1' = RAF1 + 1);
    [r14] (RAF1 < N) -> 1: (RAF1' = RAF1 + 1);
endmodule

module RKIP
    RKIP: [0..N] init N;
    [r1]  (RKIP > 0) -> RKIP*M: (RKIP' = RKIP - 1);
    [r2]  (RKIP < N) -> 1: (RKIP' = RKIP + 1);
    [r11] (RKIP < N) -> 1: (RKIP' = RKIP + 1);
endmodule

module RAF1/RKIP
    RAF1/RKIP: [0..N] init 0;
    [r1]  (RAF1/RKIP < N) -> 1: (RAF1/RKIP' = RAF1/RKIP + 1);
    [r2]  (RAF1/RKIP > 0) -> RAF1/RKIP*M:
              (RAF1/RKIP' = RAF1/RKIP - 1);
    [r3]  (RAF1/RKIP > 0) -> RAF1/RKIP*M:
              (RAF1/RKIP' = RAF1/RKIP - 1);
    [r4]  (RAF1/RKIP < N) -> 1: (RAF1/RKIP' = RAF1/RKIP + 1);
endmodule

module ERK-PP
    ERK-PP: [0..N] init N;
    [r3]  (ERK-PP > 0) -> ERK-PP*M: (ERK-PP' = ERK-PP - 1);
    [r4]  (ERK-PP < N) -> 1: (ERK-PP' = ERK-PP + 1);
    [r8]  (ERK-PP < N) -> 1: (ERK-PP' = ERK-PP + 1);
endmodule

module RAF1/RKIP/ERK-PP
    RAF1/RKIP/ERK-PP: [0..N] init 0;
    [r3]  (RAF1/RKIP/ERK-PP < N) -> 1:
              (RAF1/RKIP/ERK-PP' = RAF1/RKIP/ERK-PP + 1);
    [r4]  (RAF1/RKIP/ERK-PP > 0) ->
              RAF1/RKIP/ERK-PP*M:
                  (RAF1/RKIP/ERK-PP' = RAF1/RKIP/ERK-PP - 1);
    [r5]  (RAF1/RKIP/ERK-PP > 0) ->
              RAF1/RKIP/ERK-PP*M:
                  (RAF1/RKIP/ERK-PP' = RAF1/RKIP/ERK-PP - 1);
endmodule

module ERK
    ERK: [0..N] init 0;
    [r5]  (ERK < N) -> 1: (ERK' = ERK + 1);
    [r6]  (ERK > 0) -> ERK*M: (ERK' = ERK - 1);
    [r7]  (ERK < N) -> 1: (ERK' = ERK + 1);
endmodule

module RKIP-P
    RKIP-P: [0..N] init 0;
    [r5]  (RKIP-P < N) -> 1: (RKIP-P' =RKIP-P + 1);
    [r9]  (RKIP-P > 0) -> RKIP-P*M: (RKIP-P' =RKIP-P - 1);
    [r10] (RKIP-P < N) -> 1: (RKIP-P' =RKIP-P + 1);
endmodule

module RP
    RP: [0..N] init N;
    [r9]  (RP > 0) -> RP*M: (RP' = RP - 1);
    [r10] (RP < N) -> 1: (RP' = RP + 1);
    [r11] (RP < N) -> 1: (RP' = RP + 1);
endmodule

module MEK
    MEK: [0..N] init N;
    [r12] (MEK > 0) ->  MEK*M: (MEK' = MEK - 1);
    [r13] (MEK < N) -> 1: (MEK' = MEK + 1);
    [r15] (MEK < N) -> 1: (MEK' = MEK + 1);
endmodule

module MEK/RAF1
    MEK/RAF1: [0..N] init N;
    [r14] (MEK/RAF1> 0) ->  MEK/RAF1*M: (MEK/RAF1' = MEK/RAF1 - 1);
    [r15] (MEK/RAF1> 0) ->  MEK/RAF1*M: (MEK/RAF1' = MEK/RAF1 - 1);
    [r12] (MEK/RAF1 < N) -> 1: (MEK/RAF1' = MEK/RAF1 + 1);
endmodule

module MEK-PP
    MEK-PP: [0..N] init N;
    [r6]  (MEK-PP > 0) ->  MEK-PP*M: (MEK-PP' = MEK-PP - 1);
    [r15] (MEK-PP > 0) ->  MEK-PP*M: (MEK-PP' = MEK-PP - 1);
    [r7]  (MEK-PP < N) -> 1: (MEK-PP' = MEK-PP + 1);
    [r8]  (MEK-PP < N) -> 1: (MEK-PP' = MEK-PP + 1);
    [r14] (MEK-PP < N) -> 1: (MEK-PP' = MEK-PP + 1);
endmodule

module  MEK-PP/ERK
    MEK-PP/ERK: [0..N] init 0;
    [r7]  (MEK-PP/ERK > 0) ->  MEK-PP/ERK*M:
              (MEK-PP/ERK' =  MEK-PP/ERK - 1);
    [r8]  (MEK-PP/ERK > 0) ->  MEK-PP/ERK*M:
              (MEK-PP/ERK' =  MEK-PP/ERK - 1);
    [r6]  (MEK-PP/ERK < N) -> 1: (MEK-PP/ERK' =  MEK-PP/ERK + 1);
endmodule
```

```
module RKIP-P/RP
    RKIP-P/RP: [0..N] init 0;
    [r9]  (RKIP-P/RP < N) -> 1: (RKIP-P/RP' = RKIP-P/RP + 1);
    [r10] (RKIP-P/RP > 0) -> RKIP-P/RP*M:
              (RKIP-P/RP' = RKIP-P/RP - 1);
    [r11] (RKIP-P/RP > 0) -> RKIP-P/RP*M:
              (RKIP-P/RP' = RKIP-P/RP - 1);
endmodule

module Constants
    x: bool init true;
    [r1]  (x) -> 0.53/M: (x' = true);
    [r2]  (x) -> 0.0072/M: (x' = true);
    [r3]  (x) -> 0.625/M: (x' = true);
    [r4]  (x) -> 0.00245/M: (x' = true);
    [r5]  (x) -> 0.0315/M: (x' = true);
    [r6]  (x) -> 0.8/M: (x' = true);
    [r7]  (x) -> 0.0075/M: (x' = true);
    [r8]  (x) -> 0.071/M: (x' = true);
    [r9]  (x) -> 0.92/M: (x' = true);
    [r10] (x) -> 0.00122/M: (x' = true);
    [r11] (x) -> 0.87/M: (x' = true);
    [r12] (x) -> 0.05/M: (x' = true);
    [r13] (x) -> 0.03/M: (x' = true);
    [r14] (x) -> 0.06/M: (x' = true);
    [r15] (x) -> 0.02/M: (x' = true);
endmodule
```

References

1. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1:162–170, 2000.
2. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Computer Aided Verification*, pp. 358–372, 2000.
3. M. Calder, A. Duguid, S. Gilmore, and J. Hillston. Stronger computational modelling of signalling pathways using both continuous and discrete-state methods. In C. Priami, editor, *Computational Methods in Systems Biology 2006*, Lecture Notes in Computer Science, vol. 4210, pp. 63–77. Springer, 2006.
4. M. Calder, S. Gilmore, and J. Hillston. Automatically deriving ODEs from process algebra models of signalling pathways. In *Computational Methods in Systems Biology 2005*, pp. 204–215. LFCS, University of Edinburgh, 2005.
5. M. Calder, S. Gilmore, and J. Hillston. Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. *Transactions on Computational Systems Biology VII*, 4230:1–23, 2006.

6. M. Calder, V. Vyshemirsky, R. Orton, and D. Gilbert. Analysis of signalling pathways using continuous time Markov chains. *Transactions on Computational Systems Biology VI*, 4220:44–67, 2006.
7. D. Chiarugi, M. Curti, P. Degano, and R. Marangoni. VICE: A VIrtual CEll. In *Proceedings of the 2nd International Workshop on Computational Methods in Systems Biology*, Paris, France. Springer, April 2004.
8. K.-H. Cho, S.-Y. Shin, H.-W. Kim, O. Wolkenhauer, B. McFerran, and W. Kolch. Mathematical modeling of the influence of RKIP on the ERK signaling pathway. In C. Priami, editor, *Computational Methods in Systems Biology (CSMB'03)*, Lecture Notes in Computer Science, vol. 2602, pp. 127–141. Springer, 2003.
9. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logics of Programs: Workshop*, Lecture Notes in Computer Science, vol. 131, Yorktown Heights, New York. Springer, May 1981.
10. H. de Jong. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9(1):67–103, 2002.
11. W. H. Elliott and D. C. Elliott. *Biochemistry and Molecular Biology.* Oxford University Press, Oxford, 2002.
12. C. W. Gardiner. *Handbook of Stochastic Methods for Physics, Chemistry and the Natural Sciences.* Springer, Berlin, 2004.
13. D. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
14. D. T. Gillespie. *Markov Processes: An Introduction for Physical Scientists.* Academic Press, San Diego, 1991.
15. S. Gilmore and J. Hillston. The PEPA Workbench: A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Lecture Notes in Computer Science, vol. 794, pp. 353–368, Vienna. Springer, May 1994.
16. S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, 2001.
17. P. J. E. Goss and J. Peccoud. Quantitative modeling of stochastic systems in molecular biology by using stochastic petri nets. *Proceedings of National Academy of Science, USA*, 95(12):7650–6755, 1998.
18. J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. In C. Priami, editor, *Proceedings of 4th International Workshop on Computational Methods in Systems Biology*, Lecture Notes in Bioinformatics, vol. 4210, pp. 32–47, Trento. Springer, 18–19th October 2006.
19. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, Cambridge, MA, 1996.
20. J. G. Kemeny and J. L. Snell. *Finite Markov Chains.* Van Nostrand, Princeton, NJ, 1960.
21. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proceedings of the 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, Lecture Notes in Computer Science, vol. 2324, pp. 200–204. Springer, 2002.

22. M. Kwiatkowska, G. Norman, D. Parker, O. Tymchyshyn, J. Heath, and E. Gaffney. Simulation and verification for computational modelling of signalling pathways. In *Proceedings of the 2006 Winter Simulation Conference*, pp. 1666–1674, 2006.
23. K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.
24. R. Milner. *Communication and Concurrency.* Prentice-Hall, Upper Saddle River, NJ, 1989.
25. J. R. Norris. *Markov Chains.* Cambridge University Press, Cambridge, MA, 1997.
26. D. Parker, G. Norman, and M. Kwiatkowska. PRISM 2.1 Users' Guide. The University of Birmingham, September 2004.
27. C. Priami, A. Regev, W. Silverman, and E. Shapiro. Application of a stochastic name passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.
28. A. Regev. *Computational Systems Biology: A Calculus for Biomolecular Knowledge.* PhD thesis, Tel Aviv University, 2002.
29. A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using π-calculus process algebra. In *Pacific Symposium on Biocomputing 2001 (PSB 2001)*, pp. 459–470, 2001.
30. B. Schoeberl, C. Eichler-Jonsson, E. D. Gilles, and G. Muller. Computational modeling of the dynamics of the MAP kinase cascade activated by surface and internalized EGF receptors. *Nature Biotechnology*, 20:370–375, 2002.
31. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, Princeton, NJ, 1994.
32. E. O. Voit. *Computational Analysis of Biochemical Systems.* Cambridge University Press, Cambridge, MA, 2000.

7

Separation Logic and Concurrency

Richard Bornat

School of Information and Engineering Sciences, Middlesex University, UK
R.Bornat@mdx.ac.uk

Abstract Concurrent separation logic is a development of Hoare logic adapted to deal with pointers and concurrency. Since its inception, it has been enhanced with a treatment of permissions to enable sharing of data between threads, and a treatment of variables as resource alongside heap cells as resource. An introduction to the logic is given with several examples of proofs, culminating in a treatment of Simpson's 4-slot algorithm, an instance of racy non-blocking concurrency.

7.1 Introduction

Computing is the offspring of an historic collision, starting in the 1930s and continuing to this day, between mechanical calculation and formal logic. Mechanical calculation gave us hardware and formal logic gave us programming.

Programming is hard, probably just because it is formal,[1] and never seems to get any easier. Each advance in formalism, for example when high-level languages were invented or types were widely introduced, and each methodological improvement, for example structured programming or Dijkstra's treatment of concurrent programming discussed below, is taken by programmers as an opportunity to extend their range, rather than improve their accuracy. Advances in formalism and improvements in methodology are intended to help to prevent bugs and make the programmer's life easier, but the response of programmers has always been to write bigger and more complicated programs with about the same number of bugs as their previous smaller, simpler programs. There may be grounds to say that programming is just exactly as hard as we dare make it (but, of course, no harder!).

For some time now, perhaps since the late 1960s, concurrent programming has been the hardest kind of programming. At the moment there is a great deal of scientific interest in the subject. New approaches – in particular,

[1] Our programs are executed by machines which do not and could never understand our intentions. The executions of our programs are entirely determined by the arrangement of symbols we write down. This is a very severe discipline indeed.

non-blocking algorithms [20] and transactional memory [14] – are being tried out. New theoretical advances are beginning to give formal treatments to old problems. And hardware designers, in their relentless search for speed, are providing us with multiprocessor chips, forcing us to confront the problems of concurrent and parallel programming in our program designs. At once we are being pressed to solve a problem that we have not been able to solve before, and as we are developing enthusiasm that we might be able to make a dent in it. These are exciting times!

In the strange way that computer science has always developed, theoreticians are both behind and ahead of the game. This paper looks at some classic problems – the unbounded buffer, readers and writers and Simpson's 4-slot algorithm – which are beginning to yield to pressure.

7.2 Background

The difficulties of concurrent programming are easily illustrated. Suppose that there are two programs executing simultaneously on two processors, making two concurrent processes, and we intend them to communicate. One way to do so is via a shared store. Process A might write a message into a shared area of the store, for example, for process B to read. If this is the means of communication, B cannot know when A has communicated – notification is communication! – and so must read the store every now and then to see if A has said anything new. Then if A writes at about the same time that B reads, they may trip over each other: depending on the order that the various parts of the message are written and read, B may read part of what A is currently writing, and part of what it wrote previously, thereby retrieving a garbled message. Miscommunication is clearly possible and really happens: you can read the wrong time, for example, if the hardware clock writes a two-word value into the store and you are not careful about the way you read it.

The problem is just as acute if the processes are running in a multiprogramming system, not actually simultaneously, but with parts of their execution interleaved: A runs for a while, then B, then A and so on. Unless we carefully control communications, A can be interrupted part way through sending a message or B part way through sending one, and miscommunication can as easily result as before. And we can get tied into just the same knots if we consider threads within a single process.

Once, back in the prehistory of concurrent programming, people thought that they would have to reason about the speed of concurrent processes and prove that communication collisions could not happen. That is effectively impossible in the case of multiprogramming pseudo-concurrency, and – not for the last time – Dijkstra had to put us right:

> "We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other". [12]

He then provided us with the first effective mechanism for concurrent programming, using semaphores and critical sections, but he did not provide us with a formalism in which we could reason about concurrent programs. His injunction provides us with the basic plank of a programming framework, though we have to separate consideration of communication from consideration of what happens between communications.

Once we recognise that we need some special mechanism to communicate reliably, all kinds of new difficulties arise: deadlock (everybody waiting for somebody to move first), livelock (everybody rushing about, constantly missing each other) and problems of efficiency. Those are interesting and difficult questions, but my focus will be on safety, showing that if communication does happen, it happens properly.

The problem is eased, but not solved, by going for more tractable means of communication, like message-sending. Message-sending has to be implemented in terms of things that the hardware can do, which means worrying about who does what, when and with which physical communications medium. And the other issues, of deadlock, livelock and efficiency, somehow manage to keep poking through.

7.2.1 Related Work

Concurrent separation logic, as we shall see, is about partial-correctness specifications of threads executed concurrently. The seminal work in this area was Susan Owicki's. In her thesis [23], she tackled head-on the problems of "interference", where two processes compete for access to the same resource, and she dealt with some significant examples, including the bounded buffer, a version of readers-and-writers, dining philosophers and several semaphore programs. Sometimes referred to as the Owicki-Gries method, following the papers she wrote with her supervisor [24, 25], it proved capable of verifying really difficult algorithms [13]. But it was not without its drawbacks: it did not deal well with pointers, and in general, each instruction in every thread had to be shown not to interfere with any of the assertions expressed in any other thread. Her RPL language, using conditional critical regions (CCR) (described in Sect. 7.5), was more tractable, but restricted.

Jones's rely-guarantee [18] is the best-known of the later developments. Jones introduced thread-global "rely" and "guarantee" conditions, in addition to pre and postconditions. Rely conditions state what a thread can assume about the state of shared data, essentially constraints on the action of other threads. Guarantee conditions state what a thread must maintain in the state of shared data. Whilst more modular than Owicki's method, it is still the case that a proof must check the maintenance of guarantee conditions by each instruction of a thread.

There is also temporal logic, for example [1, 26, 29]. Temporal logic can deal with liveness; separation logic, as we shall see, can deal only with safety.

(Owicki characterised the difference as “safety is that bad things do not happen; liveness is that good things do happen”.) But temporal logic approaches so far are not as modular as separation logic.

7.3 Hoare Logic

Separation logic uses Hoare triples, a mechanism of Hoare logic [15]. In Hoare logic, you make classical-logic assertions about the state of the computer's store, and you reason about programs in terms of their effect on that store, the triple $\{Q\}\,C\,\{R\}$ is a specification of the command (program) C: if you execute C in a store which satisfies precondition Q, then it is certain to terminate, leaving a store which satisfies R. Hoare logic works with a programming language which is rather like Pascal or Algol 60: assignments to variables and arrays, sequences of commands, choices (if-then-else), loops (while-do) and procedure calls. Its greatest glory is the variable-assignment axiom $\{R[E/x]\}\,x := E\,\{R\}$, which converts difficult semantic arguments about stores into simple substitution of expression E for occurrences of variable x in the postcondition formula R, thus reducing large parts of formal program verifications to syntactic calculation. Its most evident complexity is the use of invariants to describe loops but, as we have learned to accept, that is not a fault of the logic – loops really are that difficult to understand.

One difficulty in Hoare logic is *aliasing*. It arises in a minor way with the assignment axiom – if you have more than one name for a program variable, then you cannot use simple substitution for the single name used in the assignment. That difficulty could perhaps be overcome, or variable aliasing could be outlawed as it was in Ada, but there is a worse problem in array-element aliasing. To make array-element assignment work in the logic, arrays are treated as single variables containing an indexed sequence. The array-element assignment $a[I] := E$ is treated as if it writes a new sequence value into the variable a, which is the same as the sequence before the assignment, except that at position I, it contains E. That is all very well, but if, for example, $i + 2 = j$, then elements $a[i+1]$ and $a[j-1]$ are aliases, so that assignment to one affects the other and vice-versa. There is no possibility of outlawing this problem, and as a consequence program verifications turn, more often than not, on arithmetical arguments about array indices.

Array variables are not the end of the difficulty. The program heap, in which dynamically-allocated data structures live, is really a very large array indexed by addresses – “pointers” or “references” – and the practical problems of aliasing in the heap have bedevilled Hoare-logic reasoners from the start. Following the success of structured programming in abolishing the goto, some people would have us abandon use of pointers. Hoare himself opined:

> “References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover”. [17, 19]

Gotos have almost disappeared nowadays, but pointers/references have proved more durable. One of the most popular programming languages, Java, depends almost entirely on references. The heap, like concurrency, will not go away just because we cannot reason very well about it. To recover we shall have to tame pointers, rather than banish them.

7.4 A Resource Logic for the Heap

One possible solution is to exploit separation. If, for example, there are two lists in the heap which do not share any cells, then changes to one do not affect the other and we can reason about operations on one independently of operations on the other. This was first noted by Burstall [9]. I managed to prove some graph algorithms using a Hoare logic augmented with an operation that asserted separation of data structures [3], but the proofs were very clumsy. Separation logic [21, 30], which began by building on Burstall's insight, does it much more elegantly.

Notice, first, that each variable-assignment instruction in a program affects only one store location, and Hoare's variable-assignment axiom is beautiful just because it captures that simplicity in a purely formal mechanism. The same is true of heap-cell assignment: we can only assign one cell at a time, and if that cell is separate from the cells used by other parts of the program, then we can have a single-cell heap assignment axiom. In separation logic, we write $E \mapsto F$ (E points to F) to assert exclusive ownership of the heap cell at address E with contents F. E is a positive integer expression and, for simplicity, F is an integer expression. To avoid getting tangled up in aliasing, E and F must be "pure" expressions not depending on the heap – that is, involving only variables and constants. We write $E \mapsto \_$ to express ownership without knowledge of contents (it is equivalent to $\exists\, k \cdot (E \mapsto k)$), we write $[E]$ (pronounced "contents of E") to describe the value stored at address E, and we have a heap assignment axiom

$$\{E \mapsto \_\}\, [E] := F\, \{E \mapsto F\} \tag{7.1}$$

– provided we exclusively own the cell at address E before the assignment, we can put a new value into it and be confident that we would find that value in the cell if we looked, and we still own that cell afterwards. We only need to own one cell to be able to assign it.

There is a similar heap-access assignment axiom

$$\{F = N \wedge E \mapsto F\}\, x := [E]\, \{x = N \wedge E \mapsto N\} \tag{7.2}$$

These two forms of assignment are the only way to deal with the heap in the logic and it is easy to see how more complicated patterns of access and assignment can be compiled into sequences of variable assignment, heap assignment and heap access. There is a weakest-precondition version of these axioms, but I am going to stick to forward reasoning in simple examples and will not need it.

In non-concurrent separation logic, $\{Q\}\,C\,\{R\}$ is a no-fault total-correctness assertion: if you execute C in a situation whose resources are described by Q, then it will terminate leaving resources described by R and its execution will not go outside the resource footprint described (that is, use a cell that it does not own). As we shall see, C can acquire or release resources during its operation, thereby increasing or reducing its footprint, but the specification says that it will never exceed the footprint it has at any time.

We will not always be dealing with single heap cells, so we need a logical mechanism to handle combination of resources. Separation logic's "star" does that: $A \star B$ says that we own resources described by A and at the same time but separately, resources described by B. It follows that

$$E \mapsto F \star E' \to F' \Rightarrow E \neq E' \tag{7.3}$$

– separation in the heap is separation of addresses. We allow address arithmetic: $E \mapsto F1, F2$ is shorthand for $E \mapsto F1 \star E+1 \mapsto F2$. The normal conjunction $A \wedge B$ has a resource interpretation too: we own resources which are described by A and at the same time described by B. There is a resource implication $-\!\!\star$ as well, but I am not going to use it. And that is it so far as logical assertions go: separation logic uses classical logic with a separating conjunction and resource implication.

With no more than that logical machinery it is possible to describe how, for example, a list is represented in the heap. Lists are non-cyclic and end in nil (a special non-pointing pointer value); ever since the invention of LISP they have been a sequence of two-cell records, the first cell of which contains a head value and the second a pointer to the rest of the list. The heap predicate $\mathrm{list}\, E\ \beta s$ says that E points to a list in the heap containing the sequence of head-values βs (it is true just exactly when the heap contains such a list and nothing else: combination of this heap with other heaps using $\star$ then expresses separation of the list from other data structures).

$$\begin{array}{ll} \mathrm{list}\, x\ \langle\,\rangle & \overset{\mathrm{def}}{=} x = \mathrm{nil} \wedge \mathbf{emp} \\ \mathrm{list}\, x\ (\langle\alpha\rangle +\!\!+ \beta s) & \overset{\mathrm{def}}{=} \exists x' \cdot (x \mapsto \alpha, x' \star \mathrm{list}\, x'\ \beta s) \end{array} \tag{7.4}$$

The first line says that an empty sequence is represented by nil and does not take any space: **emp** asserts ownership of nothing and is a zero, i.e., $A \star \mathbf{emp} \Leftrightarrow A$. The second line says that to represent a sequence which starts with α, we have to start with a record that contains α and points to the rest of the list, which is contained in an entirely separate area of the heap. This captures precisely the separation of the head record from the rest of the list – and, recursively, separation of cells within the rest of the list. Notice also that the assertion $x \mapsto \alpha, x'$ contains a "dangling pointer" x': we may think we know it is a pointer but we do not know, within that assertion, what it points to. Separation logic embraces the dangling pointer and that is the source of some of its power. In the conjunction $x \mapsto \alpha, x' \star \mathrm{list}\, x'\ \beta s$, we can see what x' points to, but in $x \mapsto \alpha, x'$ alone we do not know that it points to anything at all.

We can use a similar definition to describe how to represent a binary tree, with values at the nodes.

$$\begin{array}{ll} \mathrm{tree}\, x\ (\,) & \overset{\mathrm{def}}{=}\ x = \mathrm{nil} \wedge \mathbf{emp} \\ \mathrm{tree}\, x\ (\alpha, \lambda, \rho) & \overset{\mathrm{def}}{=}\ \exists\, l, r \cdot (x \mapsto l, \alpha, r \star \mathrm{tree}\, l\ \lambda \star \mathrm{tree}\, r\ \rho) \end{array} \tag{7.5}$$

An empty tree is nil; the root is separate from the left and right subtrees.

The most important inference rule of separation logic is the frame rule.

$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}}\ (\mathrm{modifies}\, C\ \cap \mathrm{vars}\, P = \varnothing) \tag{7.6}$$

If we can prove $\{Q\}\ C\ \{R\}$, and if P is separate from Q, then execution of C cannot affect it: so if we start C in a state described by $P \star Q$, it must terminate in $P \star R$. There is a side condition on the use of variables: C must not alter variables so as to alter the meaning of P. (A treatment of variables as resource – see Sect. 7.7 – eliminates that side condition, but for the time being it is easiest to take it on board.)

Reading upwards, the frame rule enables us to focus on the minimum resources required by a program, and in particular, it allows elegant axioms for new and dispose (similar to C's `malloc` and `free`).

$$\{\mathbf{emp}\}\ x := \mathrm{new}()\ \{x \mapsto \_\} \tag{7.7}$$

$$\{E \mapsto \_\}\ \mathrm{dispose}\, E\ \{\mathbf{emp}\} \tag{7.8}$$

In order to work with the frame rule, new must give us a pointer to a cell that is not in use anywhere else in the program. That is easy to implement: new has a heap of cells, initially we have none, and each call to new merely transfers one from its heap to our own. Dispose pulls the reverse trick, and crucially, just as in real life, it leaves a dangling pointer – or maybe hundreds of dangling pointers – to the cell which we used to own. Crucially again, because of the frame rule, that cell cannot be referred to in any assertion in the program. And, because new is not constrained to give us brand-new cells, it may, just as in real life, give us a pointer to a cell which we used to own and to which there are dangling pointers. Heap programmers must learn to deal with that, just as they always have.

Now it is possible to show a first example. Disposetree disposes of a tree, guarantees to leave nothing behind and does not put a foot outside the tree you give it – i.e., $\{\mathrm{tree}\, E\ \tau\}\ \mathrm{disposetree}\, E\ \{\mathbf{emp}\}$. Figure 7.1 shows the procedure, and Fig. 7.2 its proof. The proof is a tedious calculation for such a small

```
disposetree $x$ =
    if $x = \text{nil}$ then skip else
        local $l, r$; $l := [x]$; $r := [x+2]$;
        disposetree $l$; disposetree $r$;
        dispose $x$; dispose$(x+1)$; dispose$(x+2)$
    fi
```

Fig. 7.1. Tree disposal

```
disposetree $x$ =
    $\{\text{tree}\, x\ \tau\}$
    if $x = \text{nil}$ then
        $\{x = nil \wedge \mathbf{emp}\}$
        $\{\mathbf{emp}\}$
        skip
        $\{\mathbf{emp}\}$
    else
        $\{\exists\, l, r \cdot (x \mapsto l, \alpha, r \star \text{tree}\, l\ \lambda \star \text{tree}\, r\ \rho)\}$
        local $l, r$; $l := [x]$; $r := [x+2]$;
        $\{x \mapsto l, \alpha, r \star \text{tree}\, l\ \lambda \star \text{tree}\, r\ \rho\}$
        Framed: $\langle\ \{\text{tree}\, l\ \lambda\}$ disposetree $l$ $\{\mathbf{emp}\}\ \rangle$;
        $\{x \mapsto l, \alpha, r \star \mathbf{emp} \star \text{tree}\, r\ \rho\}$
        Framed: $\langle\ \{\text{tree}\, r\ \rho\}$ disposetree $r$ $\{\mathbf{emp}\}\ \rangle$;
        $\{x \mapsto l, \alpha, r \star \mathbf{emp} \star \mathbf{emp}\}$
        Framed: $\langle\ \{x \mapsto l\}$ dispose $x$ $\{\mathbf{emp}\}\ \rangle$;
        $\{\mathbf{emp} \star x+1 \mapsto \alpha, r \star \mathbf{emp} \star \mathbf{emp}\}$
        Framed: $\langle\ \{x+1 \mapsto \alpha\}$ dispose$(x+1)$ $\{\mathbf{emp}\}\ \rangle$;
        $\{\mathbf{emp} \star \mathbf{emp} \star x+2 \mapsto r \star \mathbf{emp} \star \mathbf{emp}\}$
        Framed: $\langle\ \{x+2 \mapsto r\}$ dispose$(x+2)$ $\{\mathbf{emp}\}\ \rangle$;
        $\{\mathbf{emp} \star \mathbf{emp} \star \mathbf{emp} \star \mathbf{emp} \star \mathbf{emp}\}$
        $\{\mathbf{emp}\}$
    fi
```

Fig. 7.2. Tree disposal is safe

result, but it is something that a compiler could follow, and indeed Berdine and Calcagno's Smallfoot tool [2] can guess this proof, given the definition of the tree predicate and the pre and postconditions of disposetree.

This approach to pointers gets us quite a long way, but as the examples in [4] show, it is not an immediate panacea. There is perhaps more to be wrung out of pointer problems, but the truth is that concurrency is so much more exciting that the separation-logic research community has for the time being turned its attention there.[2]

[2] In fact, as I write, attention has turned back to heap-and-pointer problems. An automatic safety verification of a version of `malloc` has been completed, and the same techniques are being applied to finding bugs in Windows device drivers.

7.5 A Resource Logic for Concurrency

Since binary trees consist of three separate parts, there is no reason in principle why we should not dispose of those three parts in parallel. I can show that the procedure in Fig. 7.3, which does just that, is safe. The disjoint-concurrency rule is

$$\frac{\{Q1\}\ C1\ \{R1\} \quad \ldots \quad \{Qn\}\ Cn\ \{Rn\}}{\{Q1 \star \cdots \star Qn\}\ \left(\ C1 \,\|\, \ldots \,\|\, Cn\ \right)\ \{R1 \star \cdots \star Rn\}} \left(\left(\begin{smallmatrix} \text{modifies } Cj \,\cap \\ (\text{vars } Qi \cup \text{vars } Ri) \end{smallmatrix} \right) = \varnothing \right) \tag{7.9}$$

– essentially, if you start concurrent threads with separated resources, they stay separated and produce separated results. The side condition on the use of variables says that threads cannot modify variables free in each other's pre and postconditions: for the time being, it is simplest to imagine that $C1 \ldots Cn$ each have their own variables.

The relevant part of the parallel disposetree proof is in Fig. 7.4. It is not an industrial-strength example, but the point is in the simplicity of the reasoning. Parallel mergesort, and even parallel quicksort, can be proved safe in the same way (see [22]).

Concurrency without communication would be pretty boring. Separation logic uses a version of Hoare's CCR mechanism [16], which is easy to reason about but hard to implement, and then uses that to describe atomic actions that are easier to implement, like semaphores and CAS instructions. A program contains a number of resource names,[3] each of which owns some heap. Each resource name r has an invariant I_r. The rule which sets up the resource is

$$\frac{\{Q\}\ C\ \{R\}}{\{I_r \star Q\}\ \textsf{let resource}\ r : I_r\ \textsf{in}\ C\ \textsf{ni}\ \{R \star I_r\}} \tag{7.10}$$

```
disposetree $x$ =
    if $x$ = nil then skip else
        local $l, r$;  $l := [x]$;  $r := [x+2]$;
        ( disposetree $l$ ‖ disposetree $r$ ‖ dispose$(x, x+1, x+2)$ )
    fi
```

Fig. 7.3. Tree disposal in parallel

$$\begin{array}{l}
\{x \mapsto l, \alpha, r \star \text{tree}\, l\ \lambda \star \text{tree}\, r\ \rho\} \\
\left(\begin{array}{l} \{\text{tree}\, l\ \lambda\} \\ \text{disposetree}\, l \\ \{\mathbf{emp}\} \end{array} \middle\| \begin{array}{l} \{\text{tree}\, r\ \rho\} \\ \text{disposetree}\, r \\ \{\mathbf{emp}\} \end{array} \middle\| \begin{array}{l} \{x \mapsto l, \alpha, r\} \\ \text{dispose}(x, x+1, x+2) \\ \{\mathbf{emp}\} \end{array} \right) \\
\{\mathbf{emp} \star \mathbf{emp} \star \mathbf{emp}\}
\end{array}$$

Fig. 7.4. Safety of the parallel section of parallel disposetree

[3] I think they should really be called resource *bundles*, because separation logic is about resource and resources in various forms, but history is hard to shake off.

(The formula I_r has to be "precise" (i.e., pick out at most a single subheap in any heap: see [8]), but that is a minor technical point – program specifications are naturally precise for the most point.) Then the CCR command $\mathsf{with}\ r\ \mathsf{when}\ G\ \mathsf{do}\ C\ \mathsf{od}$ waits for exclusive access to resource name r: when it gets it, it evaluates the guard G; if the guard is false, it releases r and tries again; if the guard is true, it executes C using the resources of the program plus the resources of I_r, finally reestablishing I_r and releasing r. The rule says just that, implicitly depending on the condition about exclusive access:

$$\frac{\{(Q \star I_r) \wedge G\}\ C\ \{R \star I_r\}}{\{Q\}\ \mathsf{with}\ r\ \mathsf{when}\ G\ \mathsf{do}\ C\ \mathsf{od}\ \{R\}} \tag{7.11}$$

The variable-use side-condition on this rule is so horrid that it is normally hidden, and I will follow that tradition because I shall show in Sect. 7.7 how to eliminate it. Much more important is to note that now we are into partial correctness: a proof in the antecedent that command C executes properly does not establish that the CCR command as a whole is ever successfully executed, because it might never be possible to get exclusive access to r when G holds. It only proves that *if* the CCR command terminates, *then* it transforms Q into R. So in concurrent separation logic including this rule, $\{Q\}\ C\ \{R\}$ is a no-fault partial-correctness assertion: C, given resources Q, will not go wrong (exceed its resource footprint, use a cell it does not own) and *if* it terminates, it will deliver resources R. Partial correctness restricts the range of the logic, as we shall see, but it leaves plenty of interesting questions.

There is a nice example which shows how it all works. Suppose for the moment that we can implement a CCR command. Then the program in Fig. 7.5 can be shown to be safe. It allocates a heap cell in the left-hand thread, which it passes into the resource *buf*, from where the right-hand thread takes it out and disposes it. Of course the heap cell does not move, what moves is *ownership* of that cell. Once it has put it in the resource, the left-hand thread cannot use that cell because it does not own it anymore (see (7.2)), it cannot modify it for just the same reason (see (7.1)) and it cannot dispose it (see (7.8)). We have, in effect, zero-cost execution barriers between threads so far as heap space is concerned – or we shall have, once we produce compilers that can carry out the necessary static checks.

$$\begin{array}{l}
\mathsf{local}\ \mathit{full}, b;\ \mathit{full} := \text{false};\\
\mathsf{let\ resource}\ \mathit{buf} : (\mathit{full} \wedge b \mapsto \_) \vee (\neg\mathit{full} \wedge \mathbf{emp})\ \mathsf{in}\\
\left(\begin{array}{l|l}
\mathsf{local}\ x; & \mathsf{local}\ y;\\
x := \text{new}(); & \mathsf{with}\ \mathit{buf}\ \mathsf{when}\ \mathit{full}\ \mathsf{do}\\
\mathsf{with}\ \mathit{buf}\ \mathsf{when}\ \neg\mathit{full}\ \mathsf{do} & \quad y := b;\ \mathit{full} := \text{false}\\
\quad b := x;\ \mathit{full} := \text{true} & \ \mathsf{od};\\
\ \mathsf{od} & \text{dispose}\ y
\end{array}\right)\\
\mathsf{ni}
\end{array}$$

Fig. 7.5. A pointer-transferring buffer (adapted from [22])

```
$\{\mathbf{emp}\}$
$x := \mathrm{new}();$
$\{x \mapsto \_\}$
with $buf$ when $\neg full$ do
    $\{x \mapsto \_ \star (\neg full \wedge \mathbf{emp})\}$
    $\{x \mapsto \_ \wedge \neg full\}$
    $b := x;$
    $\{x \mapsto \_ \wedge \neg full \wedge b = x\}$
    $full := \mathrm{true}$
    $\{x \mapsto \_ \wedge full \wedge b = x\}$
    $\{(b \mapsto \_ \wedge full) \star \mathbf{emp}\}$
od
$\{\mathbf{emp}\}$
```

```
$\{\mathbf{emp}\}$
with $buf$ when $full$ do
    $\{\mathbf{emp} \star (full \wedge b \mapsto \_)\}$
    $\{full \wedge b \mapsto \_\}$
    $y := b;$
    $\{full \wedge b \mapsto \_ \wedge y = b\}$
    $full := \mathrm{false}$
    $\{\neg full \wedge b \mapsto \_ \wedge y = b\}$
    $\{(\neg full \wedge \mathbf{emp}) \star y \mapsto \_\}$
od;
$\{y \mapsto \_\}$
dispose $y$
$\{\mathbf{emp}\}$
```

Fig. 7.6. Safety of pointer-transferring buffer

The pointer-transferring program is unremarkable, no more than a single-place buffer transferring a pointer value. It is the separation-logic treatment that makes it special. Figure 7.6 gives the proof and shows how the left thread is left with nothing once ownership is transferred into the buffer, and conversely, the right thread gets a cell ownership from the buffer. Each side starts with **emp** and finishes with **emp**; this proof does not show that the buffer resource finishes with **emp** too, but it does (see [22] for a proof). The trickery is all in the lines that deal with attachment and detachment of I_{buf} in the CCR commands. On the left we know on entry $\neg full$, so we know that the invariant reduces to $\neg full \wedge \mathbf{emp}$; on exit we know $full$, so the invariant reduces to $b \mapsto \_ \wedge full$, which we can provide – but there is not any heap left over, so the thread is left with **emp**. Proof in the right thread is similar, but in the other direction.

7.5.1 Dealing with Semaphores

Semaphores are hard to reason with, so the more tractable CCRs and monitors [7] were invented. But separation logic has opened up the possibility of *modular* reasoning about semaphores. Crucially, we treat semaphores as a kind of CCR resource, reversing history, and in an inversion of the usual view, treat them as come-in stores rather than keep-out locks.

Semaphores were invented to be guards of "critical sections" of code, like railway signals at the ends of a block of track, with hardware mutual exclusion of execution guaranteed on P and V instructions. P(m), with a binary semaphore m, blocks – i.e., the executing thread is suspended – when $m = 0$; when $m \neq 0$, it executes $m := 0$ and proceeds. V(m) simply executes $m := 1$. Then P(m) can be used to guard the entrance to a critical section, and V(m) will release it on exit. The effect, with careful use of semaphores, is to produce software mutual exclusion of execution of separate critical sections each guarded by the same semaphore – i.e., bracketed with P(m) and V(m) for

$$\begin{array}{l}
\mathsf{local}\ x, m, c;\ x := \mathrm{new}();\ [x] := 0;\ m := 0;\\
\mathsf{let\ semaphore}\ m : (m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)\ \mathsf{in}\\
\left(\begin{array}{l}
\mathrm{P}(m);\\
\quad c := [x];\ c\,{+}{+};\ [x] := c;\\
\mathrm{V}(m);\\
\ldots\\
\mathrm{P}(m);\\
\quad c := [x];\ c\,{-}{-};\ [x] := c;\\
\mathrm{V}(m)
\end{array}
\ \middle\|\ \ldots\ \middle\|\
\begin{array}{l}
\mathrm{P}(m);\\
\quad c := [x];\ c\,{+}{+};\ [x] := c;\\
\mathrm{V}(m);\\
\ldots\\
\mathrm{P}(m);\\
\quad c := [x];\ c\,{-}{-};\ [x] := c;\\
\mathrm{V}(m)
\end{array}\right)\\
\mathsf{ni}
\end{array}$$

Fig. 7.7. Example critical sections

$$\begin{array}{l}
\{\mathbf{emp}\}\\
\mathrm{P}(m) : \left\{\begin{array}{l}
\{(\mathbf{emp} \star ((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_))) \wedge m = 1\}\\
\{\mathbf{emp} \star (m = 1 \wedge x \mapsto \_)\}\\
\{m = 1 \wedge x \mapsto \_\}\\
\quad m := 0\\
\{m = 0 \wedge x \mapsto \_\}\\
\{(m = 0 \wedge \mathbf{emp}) \star x \mapsto \_\}\\
\{((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)) \star x \mapsto \_\}
\end{array}\right\}\\
\{x \mapsto \_\}
\end{array}$$

Fig. 7.8. P retrieves heap cell x from semaphore m in Fig. 7.7

the same m. Consider, for example, the example in Fig. 7.7, which contains a parallel composition of lots of copies of a couple of critical sections, the first of which increments a heap cell, the second decreasing it. If all the uses of semaphore m are those shown, then there can be no interference – no "races" – between the instructions which access the heap cell. But it is one thing to be sure that this is so and quite another to prove it in a way that a formal tool such as a compiler can check your proof – or, better still, find a proof for itself.

P(m) for a binary semaphore m is in effect with m when $m = 1$ do $m := 0$ od, and V(m) is with m when true do $m := 1$ od – note the pun in each case on the resource-name m and the semaphore-variable name m. Then in separation logic, the semaphore must have an invariant: in Fig. 7.7, I have shown it as $(m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)$. It is possible then to prove, using the CCR rule, that P(m) retrieves ownership of the heap cell x from the semaphore (Fig. 7.8) and V(m) restores it (Fig. 7.9 – the simplification between the first and second lines of the braced proof is possible since $x \mapsto \_ \star x \mapsto \_$ is false because of (7.3)).

Then the reasoning which lets us conclude that threads do not collide in critical sections is based on separation logic's *separation principle*: if threads start out with separated resources, if resource-names have separate invariants and if all threads play by the rules, staying within their own separate resource

$$
\begin{array}{l}
\{x \mapsto \_\} \\
\mathrm{V}(m) : \left\{ \begin{array}{l}
\{(x \mapsto \_ \star ((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_))) \wedge \mathrm{true}\} \\
\{x \mapsto \_ \star (m = 0 \wedge \mathbf{emp})\} \\
\{m = 0 \wedge x \mapsto \_\} \\
\quad m := 1 \\
\{m = 1 \wedge x \mapsto \_\} \\
\{(m = 1 \wedge x \mapsto \_) \star \mathbf{emp}\} \\
\{((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)) \star \mathbf{emp}\}
\end{array} \right\} \\
\{\mathbf{emp}\}
\end{array}
$$

Fig. 7.9. V replaces heap cell x in semaphore m in Fig. 7.7

$$
\left(\begin{array}{l}
\text{CONSUMER} \\
\textsf{while true do} \\
\quad tc := \mathit{front}; \\
\quad \mathrm{P}(n); \\
\quad \mathit{front} := [\mathit{front} + 1]; \\
\quad \mathrm{consume}[tc]; \\
\quad \mathrm{uncons}\ tc \\
\textsf{od}
\end{array}
\;\middle\|\;
\begin{array}{l}
\text{BUFFER} \\
\textbf{semaphore}\ n := 0
\end{array}
\;\middle\|\;
\begin{array}{l}
\text{PRODUCER} \\
\textsf{while true do} \\
\quad [\mathit{back}] := \mathrm{produce}(); \\
\quad tp := \mathrm{cons}(); \\
\quad [\mathit{back} + 1] := tp; \\
\quad \mathrm{V}(n); \\
\quad \mathit{back} := tp \\
\textsf{od}
\end{array} \right)
$$

Fig. 7.10. An unbounded buffer (adapted from [22])

footprints and communicating through CCR commands, then their resources are separate at every instant. After successfully executing P(m), a thread has $x \mapsto \_$, and it keeps hold of that cell until it executes V(m). No other thread can own or access that cell, by the separation principle. So, throughout the critical section, a thread has exclusive ownership of the cell pointed to by x, and there cannot be a race. And that is *local* reasoning: we do not have to worry about what other threads are doing, merely that they are obeying the rules (that is, they are staying within their own local resource footprint). We have provable software mutual exclusion.

Before I show any more of the mechanisms of separation logic, I want to deal with an example of the sort of program that got semaphores a bad-reasoning name. O'Hearn's version of the unbounded buffer is shown in Fig. 7.10. A sample state of the buffer is shown in Fig. 7.11: producer and consumer share a counting semaphore n; the buffer always holds a list of exactly n cells; the consumer has a (dangling) pointer to the front of the buffer list and the back of the list always (danglingly) points to a cell owned by the producer. Because the algorithm works in terms of two-cell records, I have used cons and uncons rather than new and dispose, and have supposed there is a produce function and a consume procedure to deal with the values communicated.

The most startling thing about this algorithm is that there are no critical sections: only the consumer Ps and only the producer Vs. Nevertheless, we shall see that because producer and consumer work at opposite ends of the

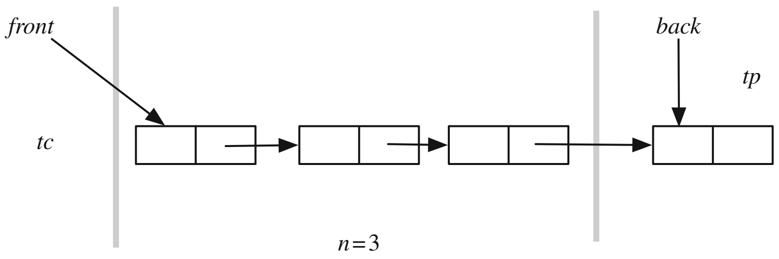


Fig. 7.11. A sample state of the unbounded buffer

buffer list, there is effective separation – even though, when the buffer list is empty, the consumer's dangling pointer points to the producer's private cell.

The details of the proof require some more information. First, because n is a counting semaphore, $\mathrm{P}(n)$ is $\mathsf{with}\ n\ \mathsf{when}\ n \neq 0\ \mathsf{do}\ n := n-1\ \mathsf{od}$ and $\mathrm{V}(n)$ is $\mathsf{with}\ n\ \mathsf{when}\ \text{true}\ \mathsf{do}\ n := n+1\ \mathsf{od}$ – only a little harder to implement in mutual exclusion than a binary semaphore. Second, in order to describe the resource held by the semaphore, I have to define a list segment – a straight-line section of a list that does not necessarily end in nil and does not loop back on itself:

$$\begin{array}{l}\text{listseg}\, n\ x\ y \ \overset{\text{def}}{=} \\ \quad (n = 0 \wedge x = y \wedge \mathbf{emp}) \ \vee \\ \qquad (n > 0 \wedge x \neq y \wedge \exists x' \cdot (x \mapsto \_, x' \star \text{listseg}\ (n-1)\ x'\ y)))\end{array} \tag{7.12}$$

(note that in listseg $n\ E\ F$, E always points to the front of the segment, unless it is empty, but F is the pointer at the end of the segment, not the address of its last cell). Finally, I have to use two "auxiliary variables" – variables put in for the purposes of the proof, whose values do not affect the operation of the program. I need a variable f that always points to the front of the buffer-list-segment, and a variable b that always tracks the pointer at the end of that segment. From now on I will highlight auxiliary variables like this – f, b – as a reminder that they do not really exist.

Now the consumer's loop-invariant is $front = f \wedge \mathbf{emp}$, the producer's is $back = b \wedge back \mapsto \_, \_$ and the counting-semaphore invariant is listseg $n\ f\ b$. $\mathrm{P}(n)$ has to move f forward to allow the consumer to take possession of the first cell in the list – it can do so instantaneously, because f does not really exist – and becomes $\mathsf{with}\ n\ \mathsf{when}\ n \neq 0\ \mathsf{do}\ n := n-1;\ f := [f+1]\ \mathsf{od}$; similarly $\mathrm{V}(n)$ becomes $\mathsf{with}\ n\ \mathsf{when}\ \text{true}\ \mathsf{do}\ n := n+1;\ b := [b+1]\ \mathsf{od}$.

With those preparations the proof is straightforward, shown in Figs. 7.12 and 7.13. By reducing n and moving f one place forward, the $\mathrm{P}(n)$ operation leaves an orphaned cell, which is pointed to by *front* in the producer. By increasing n and moving b one place forward, the $\mathrm{V}(n)$ operation absorbs the producer's private cell into the buffer list (it takes an induction, which I have omitted, to show that a list can be extended at its tail in that way).

$$
\begin{array}{l}
\{\mathit{front} = \boxed{f} \wedge \mathbf{emp}\} \\
\quad \mathit{tc} := \mathit{front} \\
\{\mathit{tc} = \mathit{front} \wedge \mathit{front} = \boxed{f} \wedge \mathbf{emp}\} \\
\mathrm{P}(n) : \left\{ \begin{array}{l}
\{((\mathit{tc} = \mathit{front} \wedge \mathit{front} = \boxed{f} \wedge \mathbf{emp}) \star \text{listseg}\, n\, \boxed{f}\, \boxed{b}) \wedge n \neq 0\} \\
\{\mathit{tc} = \mathit{front} \wedge \mathit{front} = \boxed{f} \wedge (\exists\, x \cdot (\boxed{f} \mapsto \_, x \star \text{listseg}\,(n-1)\; x\; \boxed{b}))\} \\
\quad n := n - 1 \\
\{\mathit{tc} = \mathit{front} \wedge \mathit{front} = \boxed{f} \wedge (\exists\, x \cdot (\boxed{f} \mapsto \_, x \star \text{listseg}\, n\; x\; \boxed{b}))\} \\
\quad \boxed{f} := [\boxed{f} + 1] \\
\{\mathit{tc} = \mathit{front} \wedge (\mathit{front} \mapsto \_, \boxed{f} \star \text{listseg}\, n\, \boxed{f}\, \boxed{b})\} \\
\{(\mathit{tc} = \mathit{front} \wedge \mathit{front} \mapsto \_, \boxed{f}) \star \text{listseg}\, n\, \boxed{f}\, \boxed{b})\}
\end{array} \right\} \\
\{\mathit{tc} = \mathit{front} \wedge \mathit{front} \mapsto \_, \boxed{f}\} \\
\quad \mathit{front} := [\mathit{front} + 1] \\
\{\mathit{tc} \mapsto \_, \boxed{f} \wedge \mathit{front} = \boxed{f}\} \\
\quad \text{consume}[\mathit{tc}] \\
\{\mathit{tc} \mapsto \_, \boxed{f} \wedge \mathit{front} = \boxed{f}\} \\
\quad \text{uncons}\, \mathit{tc} \\
\{\mathit{front} = \boxed{f} \wedge \mathbf{emp}\}
\end{array}
$$

Fig. 7.12. Consumer loop invariant is safely preserved

$$
\begin{array}{l}
\{\mathit{back} = \boxed{b} \wedge \mathit{back} \mapsto \_, \_\} \\
\quad [\mathit{back}] := \text{produce}() \\
\{\mathit{back} = \boxed{b} \wedge \mathit{back} \mapsto \_, \_\} \\
\quad \mathit{tp} := \text{cons}() \\
\{\mathit{back} = \boxed{b} \wedge (\mathit{back} \mapsto \_, \_ \star \mathit{tp} \mapsto \_, \_)\} \\
\quad [\mathit{back} + 1] := \mathit{tp} \\
\{\mathit{back} = \boxed{b} \wedge (\mathit{back} \mapsto \_, \mathit{tp} \star \mathit{tp} \mapsto \_, \_)\} \\
\mathrm{V}(n) : \left\{ \begin{array}{l}
\{(\mathit{back} = \boxed{b} \wedge (\mathit{back} \mapsto \_, \mathit{tp} \star \mathit{tp} \mapsto \_, \_)) \star \text{listseg}\, n\, \boxed{f}\, \boxed{b}\} \\
\quad n := n + 1 \\
\{(\mathit{back} = \boxed{b} \wedge (\mathit{back} \mapsto \_, \mathit{tp} \star \mathit{tp} \mapsto \_, \_)) \star \text{listseg}\,(n-1)\, \boxed{f}\, \boxed{b}\} \\
\quad \boxed{b} := [\boxed{b} + 1] \\
\{(\text{listseg}\,(n-1)\, \boxed{f}\; \mathit{back} \star \mathit{back} \mapsto \_, \mathit{tp} \star \mathit{tp} \mapsto \_, \_) \wedge \boxed{b} = \mathit{tp}\} \\
\{(\text{listseg}\;\, n\, \boxed{f}\; \mathit{tp} \star \mathit{tp} \mapsto \_, \_) \wedge \boxed{b} = \mathit{tp}\} \\
\{\text{listseg}\,(n-1)\, \boxed{f}\, \boxed{b} \star (\mathit{tp} \mapsto \_, \_ \wedge \boxed{b} = \mathit{tp})\}
\end{array} \right\} \\
\{\mathit{tp} \mapsto \_, \_ \wedge \boxed{b} = \mathit{tp}\} \\
\quad \mathit{back} := \mathit{tp} \\
\{\mathit{back} = \boxed{b} \wedge \mathit{back} \mapsto \_, \_\}
\end{array}
$$

Fig. 7.13. Producer loop invariant is safely preserved

If you have been paying really close attention, you will have noticed something fishy. In the last example, the auxiliary variables $\boxed{f}$ and $\boxed{b}$ were shared between the producer/consumer threads and the buffer, and the buffer operations altered them even though they appear also in the loop invariants of the threads. It is more than a little unclear that the variable-use side conditions on the CCR rule are being obeyed. Actually they are, but it takes a little more logical machinery to show it.

7.6 Permissions

Dijkstra's methodology for concurrency was to make processes/threads loosely coupled, apart from moments of synchronisation. To achieve that, we are supposed to divide the variables used by a thread into three groups:

- Those used exclusively by the thread, which can be read or written.
- Those shared with other threads, which can only be read.
- Those read and written in mutual exclusion (e.g. in critical sections, or in CCR commands).

I have been describing a logic that deals with concurrent use of heap cells rather than variables, but the principle is the same. $E \mapsto F$ (exclusive ownership) corresponds to the first case and the use of resource-name invariants allows me to deal with the third. But so far there is nothing in heap cells that corresponds to read-only shared variables.

The solution, as Boyland pointed out in [6], is to realise that ownership is licence to do anything we like with a cell (read, write, dispose), but that this can be cut up into a number of fractional *permissions*, each of which allows only read access. Some of the consequences of this idea are explored in [4]: since that paper we have gone beyond the initial idea of fractional permissions and some of our newer ideas will emerge in later examples.

With simple ownership, $E \mapsto \_ \star E \mapsto \_$ is simply false: you cannot split a heap so that both halves contain the same cell, so you cannot give exclusive ownership of a cell in the disjoint-concurrency rule (7.9) to two threads. But you can give each thread a fractional permission, via which it can read but not write or dispose. We write $E \underset{z}{\mapsto} F$ for a fractional permission, and we have

$$E \underset{1}{\mapsto} F \Leftrightarrow E \mapsto F \tag{7.13}$$

$$E \underset{z}{\mapsto} F \Rightarrow 0 < z \leq 1 \tag{7.14}$$

$$E \underset{z}{\mapsto} F \star E' \underset{z'}{\mapsto} F' \Rightarrow E = E' \Rightarrow (F = F' \wedge z + z' \leq 1) \tag{7.15}$$

$$E \underset{z}{\mapsto} F \wedge 0 < z' < z \Leftrightarrow z \leq 1 \wedge (E \underset{z-z'}{\longmapsto} F \star E \underset{z'}{\mapsto} F) \tag{7.16}$$

– $\underset{1}{\mapsto}$ means total ownership; we can only use non-zero fractions; we can combine permissions provided we do not exceed a $\underset{1}{\mapsto}$ permission; we can split permissions indefinitely. Only total ownership allows writing – i.e., we still use the axiom of (7.1) – whereas reading is allowed with fractional ownership of any size (including, of course, total ownership):

$$\{F = N \wedge E \underset{z}{\mapsto} F\}\, x := [E]\, \{x = N \wedge E \underset{z}{\mapsto} N\} \tag{7.17}$$

Permissions are no more "real" than ownership: they are artefacts of a proof, ignored by the machine but clear to the prover. In practice, apart from the restrictions applied by inter-process hardware address barriers, any instruction in a program can access any part of the heap. Like types, permissions are something we can check to be sure that our instructions only access the parts of the heap that we want them to.[4] Permissions are not types, though: they alter dynamically as a program executes and can be connected to the values of program variables.

All this means that we can deal with shared heap cells. But that is by no means all, because we can apply the same treatment to variables.

7.7 A Resource Logic for Variables

Concurrent separation logic, as I have described it so far, deals with heap cells as resource and accounts for their ownership and transfer during the operation of a program. But in some programs, ownership of variables has to be accounted for as well. At a deeper level, the frame rule, the disjoint-concurrency rule and the CCR rule have variable-use side conditions. If we treat variables as resource, those side conditions go away, and the entire load is carried by the separating conjunction ($\star$).

In conventional non-concurrent programs, use of variables is controlled by scoping rules, and these work well except when you get aliasing between reference parameters and global variables in procedure calls. (Separation logic can deal with that problem too, though I am not going to address it here.) But in concurrent programs ownership of variables gets transferred into and out of resource-holders – semaphores or CCRs or whatever – and that complicates things. Consider, for example, the unbounded buffer of Fig. 7.10. The auxiliary variable f is referred to in the invariants of the semaphore n and of the consumer. If each could have a half permission for that variable, then sharing could be formally legitimated – each partner can read f; neither has a total permission so neither can write. But f is written by the consumer when it executes P(n), which looks like a contradiction. There is in fact no contradiction: when the CCR command is executed, the permissions of the consumer and the invariant of the semaphore are combined ($Q \star I_n$ in the CCR rule), two halves make a whole, and it is possible both to read and write that variable for the duration of that instruction.

In order to preserve the simplicity of Hoare logic's treatment of variables as far as possible, it is necessary to split ownership assertions from assertions about the values of variables, rather like lvalues and rvalues, but unlike the $\mapsto$ connective used to talk about the heap. Ownership of a variable can be described by a predicate Own, which can be subscripted like $\mapsto$ to describe fractional ownership. To simplify the notation, we gather all the ownership predicates together, separate them from the rest of the assertion with a turnstile ($\Vdash$) and then do not bother to write Own at all.

[4] There is an analogy in the real world: in canon law an unfrocked priest is capable of celebrating marriage, but forbidden to do so (Marek Sergot, talk on logics of permission and belief; see also *Daily Telegraph*, 21/vi/93).

An assertion is then "well-scoped" if on the left of the turnstile it claims ownership of all the variables mentioned to the right of the turnstile. So, for example, the consumer's loop invariant in the program of Fig. 7.10 is $\mathit{front}, \mathit{tc}, \boxed{f}_{0.5} \Vdash \mathit{front} = \boxed{f} \wedge \mathbf{emp}$: it owns *front* and *tc* exclusively and owns half of $\boxed{f}$, so it can state the relationship between *front* and $\boxed{f}$ legitimately. The semaphore invariant is $\boxed{f}_{0.5}, \boxed{b}_{0.5} \Vdash \mathrm{listseg}\, n\ \boxed{f}\ \boxed{b}$: it owns half each of $\boxed{f}$ and $\boxed{b}$, so it can use their values to delimit the list. The producer's loop invariant is $\mathit{back}, \mathit{tp}, \boxed{b}_{0.5} \Vdash \mathit{back} = \boxed{b} \wedge \mathit{back} \mapsto \_, \_$, claiming ownership of *back* and *tp* and half (i.e., read) permission for $\boxed{b}$.

To assign to a variable, you need total permission:

$$\{\Gamma, x \Vdash R[E/x]\}\ x := E\ \{\Gamma, x \Vdash R\} \tag{7.18}$$

This axiom needs the side-condition that both assertions are well-scoped. An initial version of the logic is in [5]. A later version, which eliminates even that side condition, is in [27].

It is possible to deal with access to semaphores using permissions – the semaphore keeps a permission to read and write its own value, clients get permission to P and V it – but in this paper, I will spare you the details and ignore the fact that semaphores are really variables.

Treating variables as resource is very fiddly, but it is essential to explain the operation of some very famous algorithms like readers and writers.

7.7.1 Readers and Writers

The readers-and-writers algorithm of Fig. 7.14 shows how a shared variable (or any shared data structure) may be given either many simultaneous readers or a single writer (we do not need to avoid races between readers, but we must not race readers and writers, or writers and writers). It is a very succinct

```
READERS                                                  WRITER

prologue: ( P($m$);
            $c$++;
            if $c = 1$ then P($w$) else skip fi;
            V($m$);                          )
                                                         $P(w)$;
      .....                                                  .....
reading happens                                          writing happens
      .....                                                  .....
                                                         $V(w)$;
epilogue: ( P($m$);
            $c$−−;
            $k$if $c = 0$ then V($w$) else skip fi;
            V($m$)                           )
```

Fig. 7.14. Readers and writers (adapted from [11])

algorithm, but has quite a tricky proof. It has three visible critical sections, but the "reading happens" part is not one of them, because there can be many readers in it at the same time. Nevertheless, that part is clearly executed in software mutual exclusion with the "writing happens" critical section. The problem is to explain how this is so.

The easiest part of the solution is to recognise that the m semaphore owns the c variable, which it uses to count the readers. It is then impossible to read or write c without first going through $\mathrm{P}(m)$ – not at all a scoping restriction. Then clearly c counts the number of read permissions that have been handed out, and that is where the fun begins. It might be possible to track fractional permissions, but not without unreasonable arithmetic agility (I certainly cannot work out how to do it!), so instead I use a "counting permission". The idea is that you have a permission that is like a flint block, from which you chip away read permissions rather like flake arrowheads. Each flake can, in principle, be glued back into the block, and each time you chip away a flake you still have a block in your hand.

$E \overset{N}{\longmapsto} F$, with N above the arrow to distinguish it from a fractional permission, is a block from which you have chipped away exactly N counted read permissions; $E \rightarrowtail F$ is a counted read permission. For simplicity I suppose that the counted read permissions cannot be fractionally subdivided.

$$E \overset{0}{\longmapsto} F \Leftrightarrow E \mapsto F \tag{7.19}$$

$$E \overset{N}{\longmapsto} F \Rightarrow N \geq 0 \tag{7.20}$$

$$E \overset{N}{\longmapsto} F \Leftrightarrow E \overset{N+1}{\longmapsto} F \star E \rightarrowtail \_ \tag{7.21}$$

– $\overset{0}{\longmapsto}$ is a total permission; you cannot have a negative counting permission; you can always chip away a new counted permission or recombine one with its source.

We can do counting permissions for variables too and superscript them by analogy with heap counting permissions: x^N corresponds to $\overset{N}{\longmapsto}$ and $x^>$ to $\rightarrowtail$. For simplicity I suppose that the readers and writers are competing for access to a single shared variable y. In the algorithm, we start with a total permission for y in the w semaphore, which has the invariant

$$(\Vdash w = 0) \vee (y \Vdash w = 1) \tag{7.22}$$

– when $w = 0$, the y-permission has been lent out; when $w = 1$, it is safely locked away. That is enough to explain how any thread can $\mathrm{P}(w)$ to gain exclusive access to the "writing happens" critical section, and then do what it likes with y until it executes $V(w)$.

A first guess at the invariant for semaphore m might be

$$(\Vdash m = 0) \vee (c \Vdash m = 1 \wedge c = 0) \vee (c, y^c \Vdash m = 1 \wedge c > 0)$$

– when $m = 0$, everything has been lent out; when $m = 1$, either there are no readers, in which case the y permission is either with a writer or in the

w semaphore, or there are c readers and it has permission for y from which it has chipped off exactly c read permissions. That is plausible, but it is not quite right. It allows me to prove that the prologue gives a read permission for variable y – $\{\Vdash \text{true}\}$ *prologue* $\{y^{>} \Vdash \text{true}\}$ – but it does not let me prove that the epilogue takes it back – $\{y^{>} \Vdash \text{true}\}$ *epilogue* $\{\Vdash \text{true}\}$. The reason for the failure is quite subtle: we cannot deduce from ownership of a read permission $y^{>}$ that $c > 0$. In practical terms, a rogue writer could check out total y ownership from the w semaphore, chip away a read permission and then present that to the readers' epilogue: nothing in the algorithm stops it from doing so! The net effect would be to produce $c = -1$, and the m semaphore's invariant would be destroyed.

The solution is ingenious. We use an auxiliary variable t to make tickets that are handed out by the reader prologue and which have to be handed in at the epilogue. The variable does not appear in the program at all, so it is the most splendidly auxiliary addition, just like f and b in the unbounded buffer. The invariant is then

$$(\Vdash m = 0) \vee (c, t \Vdash m = 1 \wedge c = 0) \vee (c, t^{c}, y^{c} \Vdash m = 1 \wedge c > 0) \qquad (7.23)$$

Now it is possible – see Fig. 7.15 – to prove $\{y^{>}, t^{>} \Vdash \text{true}\}$ *epilogue* $\{\Vdash \text{true}\}$. With the ticket, the epilogue works perfectly, and without the ticket, a rogue writer cannot bust in (or rather, if it does, we cannot prove that the epilogue will work properly). In combining the semaphore invariant with the precondition, we can eliminate the first alternative because $m = 1$, we can eliminate the second because $t \star t^{>}$ implies t^{-1}, which is false, and we are left with the last alternative, which tells us that $c > 0$. Reasoning is then straightforward until the conclusion of the if-then-else, which generates a disjunction: either we have y^{c} or we do not. That disjunction is just what is needed when splitting off the semaphore invariant at the end of $\mathrm{V}(m)$. Proof of the prologue is similar but more straightforward, since the ticket is generated but plays no significant role.

It is possible to do more complicated examples than Fig. 7.15 – in [5], for example, there is a proof of a version of readers-and-writers which avoids starvation of either group – but they do not give any extra illumination. What is happening is very close *resource accounting*, showing that separate threads cannot possibly interfere with each other in their use of variables or the heap. This is useful, and to complain that it is tediously intricate is to miss the point. It is conceivable that this sort of approach may lead to programming tools that can do this kind of resource accounting, either automatically or with minimal hints from the programmer. Tools are good at tedious intricacy. Who would not say, for example, that the derivation of the type of *map* in the Hindley-Milner type system is not tediously intricate? It is, but we do not care because the compiler does the calculation for us. Maybe, one day, a compiler will do our resource accounting too. "Resourcing" may, perhaps, be the new typing.

```
$\{y^{>}, t^{>} \Vdash \text{true}\}$
    P$(m)$ : (
        $\left\{\begin{array}{l} (y^{>}, t^{>} \Vdash \text{true} \wedge m = 0 \wedge m = 1)\ \vee \\ (y^{>}, t^{>}, c, t \Vdash \text{true} \wedge m = 1 \wedge c = 0 \wedge m = 1)\ \vee \\ (y^{>}, t^{>}, c, t^{c}, y^{c} \Vdash \text{true} \wedge m = 1 \wedge c > 0 \wedge m = 1) \end{array}\right\}$
        $\{y^{>}, t^{>}, c, t^{c}, y^{c} \Vdash m = 1 \wedge c > 0\}$
        $\{c, t^{c-1}, y^{c-1} \Vdash m = 1 \wedge c > 0\}$
            $m := 0$
        $\{c, t^{c-1}, y^{c-1} \Vdash m = 0 \wedge c > 0\}$
        $\{(c, t^{c-1}, y^{c-1} \Vdash c > 0) \star (\Vdash m = 0)\}$
    )
$\{c, t^{c-1}, y^{c-1} \Vdash c > 0\}$
    $c--$
$\{c, t^{c}, y^{c} \Vdash c \geq 0\}$
    if $c = 0$ then
        $\{c, t^{c}, y^{c} \Vdash c \geq 0 \wedge c = 0\}$
        $\{c, t, y \Vdash c = 0\}$
        V$(w)$ : (
            $\left\{\begin{array}{l} (c, t, y, y \Vdash c = 0 \wedge w = 1 \wedge \text{true})\ \vee \\ (c, t, y \Vdash c = 0 \wedge w = 0 \wedge \text{true}) \end{array}\right\}$
            $\{c, t, y \Vdash c = 0 \wedge w = 0\}$
                $w := 1$
            $\{c, t, y \Vdash c = 0 \wedge w = 1\}$
            $\{(c, t \Vdash c = 0) \star (y \Vdash w = 1)\}$
        )
        $\{c, t \Vdash c = 0\}$
        $\{c, t^{c} \Vdash c = 0\}$
    else
        $\{c, t^{c}, y^{c} \Vdash c \geq 0 \wedge c \neq 0\}$
        $\{c, t^{c}, y^{c} \Vdash c > 0\}$
            skip
        $\{c, t^{c}, y^{c} \Vdash c > 0\}$
    fi
$\{(c, t \Vdash c = 0) \vee (c, t^{c}, y^{c} \Vdash c > 0)\}$
    V$(m)$ : (
        $\left\{\begin{array}{l} (c, t \Vdash c = 0 \wedge m = 0 \wedge \text{true})\ \vee \\ (c, t, c, t \Vdash c = 0 \wedge m = 1 \wedge c = 0 \wedge \text{true})\ \vee \\ (c, t, c, t^{c}, y^{c} \Vdash c = 0 \wedge m = 1 \wedge c > 0 \wedge \text{true})\ \vee \\ (c, t^{c}, y^{c} \Vdash c > 0 \wedge m = 0 \wedge \text{true})\ \vee \\ (c, t^{c}, y^{c}, c, t \Vdash c > 0 \wedge m = 1 \wedge c = 0 \wedge \text{true})\ \vee \\ (c, t^{c}, y^{c}, c, t^{c}, y^{c} \Vdash c > 0 \wedge m = 1 \wedge c > 0 \wedge \text{true}) \end{array}\right\}$
        $\left\{\begin{array}{l} (c, t \Vdash c = 0 \wedge m = 0 \wedge \text{true})\ \vee \\ (c, t^{c}, y^{c} \Vdash c > 0 \wedge m = 0 \wedge \text{true}) \end{array}\right\}$
            $m := 1$
        $\left\{\begin{array}{l} (c, t \Vdash c = 0 \wedge m = 1 \wedge \text{true})\ \vee \\ (c, t^{c}, y^{c} \Vdash c > 0 \wedge m = 1 \wedge \text{true}) \end{array}\right\}$
        $\left\{ (\Vdash \text{true}) \star \left(\begin{array}{l} (c, t \Vdash c = 0 \wedge m = 1)\ \vee \\ (c, t^{c}, y^{c} \Vdash c > 0 \wedge m = 1) \end{array}\right) \right\}$
    )
$\{\Vdash \text{true}\}$
```

Fig. 7.15. Proof of safety of the reader epilogue

7.8 Nonblocking Concurrency

I would be dishonest if I were to pretend that everything in the separation logic pond is plain sailing. We have reached a point where we can make a good fist of the proofs of some race-free concurrent algorithms, but there are always more challenges. People nowadays, for example, are getting very excited about so-called "nonblocking" concurrency [20], in which concurrent threads attempt to make updates to a shared data structure at exactly the same time, backing off and trying again if their attempt fails, rather in the way that database transactions have been dealt with for some time now. This is claimed to have efficiency advantages – modern processors hate to block, and if you have more than one processor concurrently acting, it makes sense to let it try to make progress – and programming advantages too, in the shape of the "atomic" construct of the software-transactional-memory approach [14]. It is beyond the scope of this paper to consider whether transactional memory and/or non-blocking concurrency is the wave of the future. It is enough to say that it is a hot research topic and one that separation logic might hope to pick up.

In joint work with Matthew Parkinson, I have helped to find a proof for a particularly small and simple non-blocking algorithm (Parkinson has a neater invariant, but in the interests of intellectual honesty, I give my own here). We have also been able to show that proofs like this can be made *modular*, in the sense that the racy interference necessary to make inter-thread communication can entirely be confined to the internals of the operations which implement that communication. That is, communications can be treated as black-box operations with conventional pre and postconditions, and the rest of the code of a thread can be handled independently, as Dijkstra recommended [28]. This is in contrast to other approaches to racy concurrency, such as rely-guarantee [18], in which the whole of a thread's proof is affected by interference. But on the other hand, again in the interests of intellectual honesty, I have to say that our proofs do not go far enough, and we hope to learn from rely-guarantee and temporal logic so as to be able to extend our logic and make modular proofs which are more readable and understandable and so more convincing.

Despite those caveats, we do have some proofs and it is not really necessary to apologise if work at the frontier of science is at first a little scrappy. New approaches, new avenues, new thoughts: all are likely to be more than a little messy at first. I press on.

Simpson's 4-slot algorithm [31] is shown in Fig. 7.16. It is a remarkable mechanism: it has *absolutely no* synchronisation instructions at all and, unlike most non-blocking algorithms, it does not have a loop-and-try-again structure. It implements a kind of shared variable: a single writer thread puts values into one of the slots of the shared array *data*, and a single reader thread later retrieves it. If they act at exactly the same time, then the reader gets a value that the writer put there earlier. The reader always sees a value which is no earlier than the latest value written at the instant it starts the *read* procedure,

```
var   $readcol := 0, latest := 0 : bit$
      $slot :$ array $bit$ of $bit := \{0, 0\}$
      $data :$ array $bit$ of array $bit$ of $datatype$

procedure  $write$  $(item : datatype)$;
           var      $col, row : bit$;
           begin
                    $col :=$ not$(readcol)$;
                    $row :=$ not$(slot[col])$;
                    $\boxed{data[col, row] := item}$;
                    $slot[col] := row$;
                    $latest := col$
           end;

procedure  $read :$  $datatype$;
           var      $col, row : bit$;
           begin
                    $col := latest$;
                    $readcol := col$;
                    $row := slot[col]$;
                    $\boxed{read := data[col, row]}$
           end;
```

Fig. 7.16. Simpson's 4-slot algorithm (adapted from [31])

but it can be overtaken by the writer as often as maybe, so the value it reads can be somewhat out-of-date.

Simpson's algorithm relies for its correctness, as do all non-blocking algorithms, on serialisation of store operations. A shared memory will be able to read or write but not both at the same time, nor even to overlap two operations of the same kind in the same area. "Word"-sized operations are serialised and synchronisation problems arise because we want to communicate using more than a single word. Simpson makes certain that most of his variables will have serialised access by making them single-bit; only the elements of the *data* array can be larger and cannot, therefore, be assumed to be hardware-serialised. In Fig. 7.16, the operations that we need to worry about are boxed: every other assignment involves only a single-bit access or a single-bit update to shared variables. In effect, the hardware makes those accesses and updates mutually exclusive and we can treat them as if they were CCR commands for a single resource-name which owns all the shared store, both variables and array.

Essentially that is the way the proof works. Writing "atomic$\{C\}$" as shorthand for "with *shareddata* when true do C od", and adding auxiliary variables *writecol* to indicate the column the writer is working in and *readrow* for the row the reader is looking at, the code of the algorithm is shown in Fig. 7.17. It is easy to see that each atomic instruction reads or writes exactly one one-bit global variable (note that each procedure has its own local variables

```
var       $readcol := 0, latest := 0 : bit$
          $slot :$ array $bit$ of $bit := \{0, 0\}$
          $data :$ array $bit$ of array $bit$ of $datatype$
auxvar    $writecol := -1, readrow := -1 : twobit$

procedure $write$  $(item : datatype)$;
          var      $col, row : bit$;
          begin
                   atomic$\{col := \mathrm{not}(readcol);\ writecol := col\}$;
                   $row := \mathrm{not}(slot[col])$;
                   $\boxed{data[col, row] := item}$;
                   atomic$\{slot[col] := row;\ writecol := -1\}$;
                   atomic$\{latest := col\}$
          end;

procedure $read :$  $datatype$;
          var      $col, row : bit$;
          begin
                   atomic$\{col := latest\}$;
                   atomic$\{readcol := col\}$;
                   atomic$\{row := slot[col];\ readrow := row\}$;
                   $\boxed{read := data[col, row]}$;
                   atomic$\{readrow := -1\}$
          end;
```

Fig. 7.17. Simpson's 4-slot algorithm with "atomic" annotations and auxiliary variables

col and *row*) or one element from the bit-array *slot*. (In separation logic *slot* is a pointer to two heap cells and $slot[col]$ is really $[slot + col]$, and similarly $data[col, row]$ can be translated into a pointer operation, but I have left the array notation in the program and the proof, for simplicity's sake.)

Just as in the unbounded buffer, ownership of a variable or a heap cell can be part in a thread, part in the shared resource. Some variables never need to be written – the *slot* and *data* pointers, for example – but all three components need to refer to them, so ownership can be split three ways. Ownership of those which are written in an atomic operation – *latest*, for example – can be split between a thread and the shared data invariant: then the other thread can read that variable in an atomic instruction, but not write it.

The proof depends on the fact that the writer uses a slot in the *writecol* column and the reader uses the *readrow* row of the *readcol* column. These slots, we shall see, do not overlap. The pre and postcondition of the *write* procedure is

$$latest_{0.5}, slot_{0.33}, data_{0.33}, writecol_{0.5}, col, row \Vdash writecol = -1 \land slot \xmapsto[0.5]{} \_, \_ \tag{7.24}$$

– it shares *latest* and *writecol* with the shared invariant, *slot* and *data* with the shared invariant and the reader; it shares the content of the *slot* array with the shared invariant. When *writecol* is 0 or 1 inside the procedure, the writer also owns $data[writecol, \mathrm{not}(slot[writecol])]$.

The reader is slightly simpler:

$$readcol_{0.5}, readrow_{0.5}, slot_{0.33}, data_{0.33}, col, row \Vdash readrow = -1 \tag{7.25}$$

– it shares *readcol* and *readrow* with the shared invariant and *slot* and *data* with the shared invariant and the reader. When *readrow* is 0 or 1, the reader also owns $data[readcol, readrow]$. By the separation principle, this cannot be the same as the $data[writecol, \mathrm{not}(slot[writecol])]$ slot, which the writer owns. To show that everything works, it is only necessary to show that the atomic operations which allocate slots to the writer and the reader can always do their job and allocate separate slots.

The invariant of the shared data resource is shown in Fig. 7.18. It is possible to express this invariant more succinctly, as Parkinson does, but I have chosen to do it as an enumeration of states because that illuminates the way the algorithm works. The resource owns matching permissions for the variables that the threads part own and shares the *slot* array with the writer. When the writer is communicating (lines 2 and 3 of the *write* procedure) *writecol* is 0 or 1, and similarly *readrow* is 0 or 1 when the reader is communicating (lines 3 and 4 of the *read* procedure). Either both threads are communicating, in which case the shared invariant owns the slots the threads do not own;

```
$latest_{0.5}, readcol_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, readrow_{0.5}$
    ( $slot \xmapsto[0.5]{} \_, \_ \star$
      if $writecol \geq 0 \wedge readrow \geq 0$ then
         if $writecol = readcol$ then
            $data[\mathrm{not}(writecol), slot[\mathrm{not}(writecol)]] \mapsto \_, \_ \star$
            $data[\mathrm{not}(writecol), \mathrm{not}(slot[\mathrm{not}(writecol)])] \mapsto \_, \_$
         else
            $data[readcol, \mathrm{not}(readrow)] \mapsto \_, \_ \star data[writecol, slot[writecol]] \mapsto \_, \_$
         fi
      elsf $writecol \geq 0$ then
$\Vdash$   $data[writecol, slot[writecol]] \mapsto \_, \_ \star$
         $data[\mathrm{not}(writecol), slot[\mathrm{not}(writecol)]] \mapsto \_, \_ \star$
         $data[\mathrm{not}(writecol), \mathrm{not}(slot[\mathrm{not}(writecol)])] \mapsto \_, \_$
      elsf $readrow \geq 0$ then
         $data[readcol, \mathrm{not}(readrow)] \mapsto \_, \_ \star$
         $data[\mathrm{not}(readcol), slot[\mathrm{not}(readcol)]] \mapsto \_, \_ \star$
         $data[\mathrm{not}(readcol), \mathrm{not}(slot[\mathrm{not}(readcol)])] \mapsto \_, \_)$
      else
         $data \mapsto \_, \_, \ \_, \_, \ \_, \_, \ \_, \_$
      fi )
```

Fig. 7.18. The shared data invariant

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = -1 \wedge (slot[0] \underset{0.5}{\mapsto} \_ \star slot[1] \underset{0.5}{\mapsto} \_) \end{array}\right\}$$

$$\mathsf{atomic}\{col := \mathrm{not}(readcol);\ writecol := col\};$$

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = col \wedge \left(\begin{array}{l} slot[0] \underset{0.5}{\mapsto} \_ \star slot[1] \underset{0.5}{\mapsto} \_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto \_, \_ \end{array}\right) \end{array}\right\}$$

$$row := \mathrm{not}(slot[col]);$$

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \qquad (writecol = col \wedge row = \mathrm{not}(slot[col])) \star \\ \quad \Vdash \left(\begin{array}{l} slot[0] \underset{0.5}{\mapsto} \_ \star slot[1] \underset{0.5}{\mapsto} \_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto \_, \_ \end{array}\right) \end{array}\right\}$$

$$data[col, row] := item;$$

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = col \wedge \left(\begin{array}{l} slot[0] \underset{0.5}{\mapsto} \_ \star slot[1] \underset{0.5}{\mapsto} \_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto item \end{array}\right) \end{array}\right\}$$

$$\mathsf{atomic}\{slot[col] := row;\ writecol := -1\};$$

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = -1 \wedge (slot[0] \underset{0.5}{\mapsto} \_ \star slot[1] \underset{0.5}{\mapsto} \_) \end{array}\right\}$$

$$\mathsf{atomic}\{latest := col\}$$

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = -1 \wedge (slot[0] \underset{0.5}{\mapsto} \_ \star slot[1] \underset{0.5}{\mapsto} \_) \end{array}\right\}$$

Fig. 7.19. The writer proof summarised

or one or the other is communicating, in which case it owns the slots the communicating thread does not use or neither is communicating, where it owns the whole array. In the case that both threads are communicating, there are two possibilities: either they are working in separate rows of the same column (this can happen if the reader is overtaken by the writer) or they are working in separate columns. To emphasise that the slots of the *data* array cannot be written atomically, I have made them two heap cells each.

Proof of separation for the writer is summarised in Fig. 7.19. In its first step, it extracts ownership of the slot it is going to assign, and this step is shown in detail in Fig. 7.20. Entering with $writecol = -1$, the *data* array splits into those parts which the reader might own (the *readcol* column) and those it does not (the not(*readcol*) column). The writer picks one of those that the reader does not at the moment seem to require, splits the invariant off and exits with the slot it wants. The $writecol := col$ action can be crammed into the same atomic action since it only affects an auxiliary variable, and therefore, does not really happen.

Nothing else the writer does in Fig. 7.19 is so complicated. The second step is not even atomic (the writer already has half permission for the contents of the *slot* array, and *row* and *col* are local); the third step writes into the slot of the *data* array owned since the first step; the fourth step alters the *slot* array and returns the borrowed *data* slot and the final step, as we shall see, is irrelevant to this proof.

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \end{array}\right\}$$

atomic{

$$\left\{\begin{array}{l} latest, readcol_{0.5}, slot, data_{0.66}, writecol, col, row \\ \quad \Vdash \left(\begin{array}{l} writecol = -1 \wedge slot \mapsto \_,\_ \star \\ data[\mathrm{not}(readcol), slot[\mathrm{not}(readcol)]] \mapsto \_,\_ \star \\ data[\mathrm{not}(readcol), \mathrm{not}(slot[\mathrm{not}(readcol)])] \mapsto \_,\_ \star \\ \textsf{if } readrow \geq 0 \textsf{ then } data[readcol, \mathrm{not}(readrow)] \mapsto \_,\_ \\ \qquad\qquad \textsf{else} \left(\begin{array}{l} data[readcol, slot[readcol]] \mapsto \_,\_ \star \\ data[readcol, \mathrm{not}(slot[readcol])] \mapsto \_,\_ \end{array}\right) \\ \textsf{fi} \end{array}\right) \end{array}\right\}$$

$col := \mathrm{not}(readcol)$;

$$\left\{\begin{array}{l} latest, readcol_{0.5}, slot, data_{0.66}, writecol, col, row \\ \quad \Vdash \left(\begin{array}{l} writecol = -1 \wedge col = \mathrm{not}(readcol) \wedge slot \mapsto \_,\_ \star \\ data[col, slot[col]] \mapsto \_,\_ \star data[col, \mathrm{not}(slot[col])] \mapsto \_,\_ \star \\ \textsf{if } readrow \geq 0 \textsf{ then } data[readcol, \mathrm{not}(readrow)] \mapsto \_,\_ \\ \qquad\qquad \textsf{else} \left(\begin{array}{l} data[readcol, slot[readcol]] \mapsto \_,\_ \star \\ data[readcol, \mathrm{not}(slot[readcol])] \mapsto \_,\_ \end{array}\right) \\ \textsf{fi} \end{array}\right) \end{array}\right\}$$

$writecol := col$

$$\left\{\begin{array}{l} latest, readcol_{0.5}, slot, data_{0.66}, writecol, col, row \\ \quad \Vdash \left(\begin{array}{l} writecol = col \wedge col = \mathrm{not}(readcol) \wedge slot \mapsto \_,\_ \star \\ data[writecol, slot[writecol]] \mapsto \_,\_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto \_,\_ \star \\ \textsf{if } readrow \geq 0 \textsf{ then } data[readcol, \mathrm{not}(readrow)] \mapsto \_,\_ \\ \qquad\qquad \textsf{else} \left(\begin{array}{l} data[readcol, slot[readcol]] \mapsto \_,\_ \star \\ data[readcol, \mathrm{not}(slot[readcol])] \mapsto \_,\_ \end{array}\right) \\ \textsf{fi} \end{array}\right) \end{array}\right\}$$

$$\left\{\begin{array}{l} latest, readcol_{0.5}, slot, data_{0.66}, writecol, col, row \\ \quad \Vdash \left(\begin{array}{l} writecol = col \wedge col = \mathrm{not}(readcol) \wedge slot \mapsto \_,\_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto \_,\_ \star \\ \left(\begin{array}{l} \textsf{if } writecol \geq 0 \textsf{ then } data[writecol, slot[writecol]] \mapsto \_,\_ \\ \textsf{else} \left(\begin{array}{l} data[writecol, slot[writecol]] \mapsto \_,\_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto \_,\_ \end{array}\right) \\ \textsf{fi} \end{array}\right) \star \\ \left(\begin{array}{l} \textsf{if } readrow \geq 0 \textsf{ then } data[readcol, \mathrm{not}(readrow)] \mapsto \_,\_ \\ \textsf{else} \left(\begin{array}{l} data[readcol, slot[readcol]] \mapsto \_,\_ \star \\ data[readcol, \mathrm{not}(slot[readcol])] \mapsto \_,\_ \end{array}\right) \\ \textsf{fi} \end{array}\right) \end{array}\right) \end{array}\right\}$$

};

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \\ \quad \Vdash writecol = col \wedge \left(\begin{array}{l} slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \star \\ data[writecol, \mathrm{not}(slot[writecol])] \mapsto \_,\_ \end{array}\right) \end{array}\right\}$$

Fig. 7.20. The writer's first step in detail

$$\left\{ \mathit{readcol}_{0.5}, \mathit{readrow}_{0.5}, \mathit{data}_{0.33}, \mathit{col}, \mathit{row} \Vdash \mathit{readrow} = -1 \land \mathbf{emp} \right\}$$
$$\quad \mathsf{atomic}\{\mathit{col} := \mathit{latest}\};$$
$$\left\{ \mathit{readcol}_{0.5}, \mathit{readrow}_{0.5}, \mathit{data}_{0.33}, \mathit{col}, \mathit{row} \Vdash \mathit{readrow} = -1 \land \mathbf{emp} \right\}$$
$$\quad \mathsf{atomic}\{\mathit{readcol} := \mathit{col}\};$$
$$\left\{ \mathit{readcol}_{0.5}, \mathit{readrow}_{0.5}, \mathit{data}_{0.33}, \mathit{col}, \mathit{readrow} = -1 \land \mathit{readcol} = \mathit{col} \land \mathbf{emp} \right\}$$
$$\quad \mathsf{atomic}\{\mathit{row} := \mathit{slot}[\mathit{col}];\ \mathit{readrow} := \mathit{row}\};$$
$$\left\{ \begin{array}{l} \mathit{readcol}_{0.5}, \mathit{readrow}_{0.5}, \mathit{data}_{0.33}, \mathit{col}, \mathit{row} \\ \quad \Vdash \mathit{readrow} = \mathit{row} \land \mathit{readcol} = \mathit{col} \land \mathit{data}[\mathit{col}, \mathit{row}] \mapsto \_, \_ \end{array} \right\}$$
$$\quad \mathit{read} := \mathit{data}[\mathit{col}, \mathit{row}];$$
$$\left\{ \begin{array}{l} \mathit{readcol}_{0.5}, \mathit{readrow}_{0.5}, \mathit{data}_{0.33}, \mathit{col}, \mathit{row} \\ \quad \Vdash \mathit{readrow} = \mathit{row} \land \mathit{readcol} = \mathit{col} \land \exists i \cdot (\mathit{data}[\mathit{col}, \mathit{row}] \mapsto i \land \mathit{read} = i) \end{array} \right\}$$
$$\quad \mathsf{atomic}\{\mathit{readrow} := -1\}$$
$$\left\{ \mathit{readcol}_{0.5}, \mathit{readrow}_{0.5}, \mathit{data}_{0.33}, \mathit{col}, \mathit{row} \Vdash \mathit{readrow} = -1 \land \mathbf{emp} \right\}$$

Fig. 7.21. The reader proof summarized

Proving separation in the reader is a little more intricate, but hardly more complicated. Figure 7.21 shows the summary. The first two steps pick a value for *readcol* and the third step claims a slot. Figure 7.22 shows that step in detail.

There is an uncertainty at the beginning of the reader's third step, because nothing provides that *readcol* and *writecol* are different at that instant. But whichever is true, there is no problem: if the variables are the same then the reader claims the slot in *writecol* that the writer is not using, and if they are different then there is no possibility of a clash. In both cases, separation is preserved.

7.8.1 What has Been Proved?

The four-slot algorithm does not really need a formal proof because it can be completely model-checked – Simpson himself did so in [32]. But not all difficult concurrent algorithms have such a small state space, and if we are ever to have a "verifying compiler" for concurrent programs, we shall have to deal with problems like this one. The techniques employed here – shared-data invariant, atomic operations containing a single serialised shared-store operation – have already been employed to deal with a shared-stack problem too large to model-check [28].

This proof is also rather small compared with the hundred-page refinement development in [10], which uses rely-guarantee techniques. One of the reasons for that is that the separation logic proof needs only to deal with interference inside the *read* and *write* procedures, using logical techniques that I do not have space to include here but which are dealt with in [28]. The aim of separation logic developments is to increase modularity in proofs, and in this case, we have managed that rather well.

All the ingenuity in this proof is in the choice of the auxiliary variables *readrow* and *writecol*, and in the shared-data invariant. After that it really is

```
$\{\ readcol_{0.5}, readrow_{0.5}, data_{0.33}\ readrow = -1 \wedge readcol = col \wedge \mathbf{emp}\ \}$
atomic{
    $\left\{ \begin{array}{l} latest_{0.5}, readcol, slot_{0.5}, data_{0.66}, readrow, writecol_{0.5}, col, row \\ \quad \Vdash \left( \begin{array}{l} (readrow = -1 \wedge readcol = col) \wedge slot \xmapsto[0.5]{} \_,\_ \star \\ \textsf{if } writecol \geq 0 \textsf{ then} \\ \quad data[writecol, slot[writecol]] \mapsto \_,\_ \star \\ \quad data[\mathrm{not}(writecol), slot[\mathrm{not}(writecol)]] \mapsto \_,\_ \star \\ \quad data[\mathrm{not}(writecol), \mathrm{not}(slot[\mathrm{not}(writecol)])] \mapsto \_,\_ \\ \textsf{else} \\ \quad data \mapsto \_,\_,\ \_,\_,\ \_,\_,\ \_,\_ \\ \textsf{fi} \end{array} \right) \end{array} \right\}$
    $row := slot[col];$
    $\left\{ \begin{array}{l} latest_{0.5}, readcol, slot_{0.5}, data_{0.66}, readrow, writecol_{0.5}, col, row \\ \quad \Vdash \left( \begin{array}{l} (readrow = -1 \wedge readcol = col \wedge row = slot[col]) \ \wedge \\ \left( \begin{array}{l} slot \xmapsto[0.5]{} \_,\_ \star \\ \textsf{if } writecol \geq 0 \textsf{ then} \\ \quad data[writecol, slot[writecol]] \mapsto \_,\_ \star \\ \quad data[\mathrm{not}(writecol), slot[\mathrm{not}(writecol)]] \mapsto \_,\_ \star \\ \quad data[\mathrm{not}(writecol), \mathrm{not}(slot[\mathrm{not}(writecol)])] \mapsto \_,\_ \\ \textsf{else} \\ \quad data \mapsto \_,\_,\ \_,\_,\ \_,\_,\ \_,\_ \\ \textsf{fi} \end{array} \right) \end{array} \right) \end{array} \right\}$
    $readrow := row$
    $\left\{ \begin{array}{l} latest_{0.5}, readcol, slot_{0.5}, data_{0.66}, readrow, writecol_{0.5}, col, row \\ \quad \Vdash \left( \begin{array}{l} (readcol = col \wedge row = slot[col] \wedge readrow = row) \ \wedge \\ \left( \begin{array}{l} slot \xmapsto[0.5]{} \_,\_ \star \\ \textsf{if } writecol \geq 0 \textsf{ then} \\ \quad data[writecol, slot[writecol]] \mapsto \_,\_ \star \\ \quad data[\mathrm{not}(writecol), slot[\mathrm{not}(writecol)]] \mapsto \_,\_ \star \\ \quad data[\mathrm{not}(writecol), \mathrm{not}(slot[\mathrm{not}(writecol)])] \mapsto \_,\_ \\ \textsf{else} \\ \quad data \mapsto \_,\_,\ \_,\_,\ \_,\_,\ \_,\_ \\ \textsf{fi} \end{array} \right) \end{array} \right) \end{array} \right\}$
}
$\left\{ \begin{array}{l} readcol_{0.5}, readrow_{0.5}, data_{0.33}, col, row \\ \quad \Vdash readrow = row \wedge readcol = col \wedge data[col, row] \mapsto \_,\_ \end{array} \right\}$
```

Fig. 7.22. The reader's third step

a matter of tedious calculation, not all of which I have detailed. I think that is a strength: if "resourcing" is to be as successful as typing, we must make algorithmic the steps which connect user-supplied assertions to the program, so that a compiler or the like can carry them out. I neither pretend that we have reached that stage, nor that the hints that a user would have to give to a formal programming tool can yet be simple enough to be popularly acceptable.

But there is a more important sense in which this proof might be considered unsatisfactory: it does not show that the reader and writer actually communicate. The *slot* array and the *latest* variable are used by the writer to point the reader to the most recent value written. Indeed it is possible to see that the reader will not read a value written earlier than the last one written before it begins to execute a *read* operation. All we have been able to show so far is that reader and writer do not collide, not that effective communication occurs. It ought to be possible, in this particular example which only uses true in the guards of its CCR commands, to do a little better.

7.9 Conclusion

Separation logic is a step forward in the search for modular proofs of safety assertions. As a partial-correctness logic, it cannot deal with questions of liveness and termination. But since it can deal easily with pointers, safety properties of a wider range of programs than before are now practically specifiable and verifiable. The examples in this paper show that the frontier is now much farther forward than was envisaged when work on the logic began. Active work is going on to incorporate this advance into practical programming tools, and to extend the logic's range even further by dealing with some properties that at present need global proof, by incorporating some of the insights of rely-guarantee and/or temporal logic. Watch this space!

Acknowledgements

Some of the proofs in this paper are my own and I have contributed some of the logic. But almost all the logic is due to O'Hearn and Reynolds, and the early examples are theirs too. In addition, I owe an enormous debt to all the other members – Josh Berdine, Cristiano Calcagno, Dino DiStefano, Ivana Mijhailovic, Matthew Parkinson – of the group we call the East London Massive, in whose meetings these ideas were kicked around and collectively refined. The mistakes, I need hardly add, are my own.

References

1. H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *STOC '84: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pp. 51–63, New York, 1984. ACM Press.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of FMCO'05*, Lecture Notes in Computer Science, vol. 4111, pp. 115–137, Berlin, 2006. Springer.
3. R. Bornat. Proving pointer programs in Hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference*, Lecture Notes in Computer Science, pp. 102–126, New York, 2000. Springer.

4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 259–270, New York, January 2005. ACM Press.
5. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in Separation Logic. In *Proceedings of MFPS XXI*. Elsevier ENTCS, May 2005.
6. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, Lecture Notes in Computer Science, vol. 2694, pp. 55–72, Berlin, Heidelberg, New York, 2003. Springer.
7. P. Brinch Hansen, editor. *The Origin of Concurrent Programming*. New York, 2002. Springer.
8. S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007.
9. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
10. J. Burton. *The Theory and Practice of Refinement-After-Hiding*. Ph.D. thesis, University of Newcastle upon Tyne, 2005. Available as technical report CS-TR-904.
11. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communication of the ACM*, 14(10):667–668, 1971.
12. E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pp. 43–112. New York, 1968. Academic Press.
13. D. Gries. An exercise in proving parallel programs correct. *Communication of the ACM*, 20(12):921–930, 1977.
14. T. Harris, S. Marlowe, S. Peyton-Jones, and M. P. Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–60, New York, 2005. ACM Press.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12(10):576–580, 1969.
16. C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating System Techniques*, pp. 61–71, New York, 1972. Academic Press.
17. C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, Computer Science Department, 1973. Keynote address to ACM SIGPLAN Conference, pp. 193–216.
18. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pp. 321–332, North Holland, 1983. Amsterdam.
19. C. B. Jones, editor. *Essays in Computing Science*. Upper Saddle River, NJ, 1989. Prentice-Hall.
20. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 267–275, New York, 1996. ACM Press.
21. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, Lecture Notes in Computer Science, vol. 2142, pp. 1–19, Berlin, 2001. Springer.
22. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

23. S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* Ph.D. thesis, Cornell, 1975. Technical report TR75-251.
24. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 19:319–340, 1976.
25. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communication of the ACM*, 19(5):279–285, 1976.
26. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.
27. M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Proceedings of LICS*, pp. 137–146, Seattle, WA, 2006. IEEE.
28. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 297–302, New York, 2007. ACM.
29. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.
30. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, Washington, DC, 2002. IEEE Computer Society.
31. H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137(1):17–30, 1990.
32. H. R. Simpson. Role model analysis of an asynchronous communication mechanism. *IEE Proceedings of Computer Digital Technology*, 144:232–240, 1997.

8

Programming Language Description Languages

From Christopher Strachey to Semantics Online

Peter D. Mosses

Department of Computer Science, Swansea University, Swansea, SA2 8PP, UK
p.d.mosses@swan.ac.uk

Abstract Since the middle of the twentieth century, hundreds of programming languages have been designed and implemented – and new ones are continually emerging. The syntax of a programming language can usually be described quite precisely and efficiently using formal grammars. However, the formal description of its semantics is much more challenging. Language designers, implementers and programmers commonly regard formal semantic descriptions as impractical. Research in semantics has allowed us to reason about software and has provided valuable insight into the design of programming languages, but few semantic descriptions of full languages have been published, and hardly any of these are currently available online.

One of the major approaches to formal semantics is denotational semantics, which originated from Strachey's pioneering studies in the 1960s. Why is such a theoretically attractive approach generally regarded as impractical for describing full-scale programming languages? Does it help much to use monads in denotational descriptions, or is a more radical change needed? How might efficient online access to a repository of semantic descriptions be provided? Could it ever become as easy to generate efficient compilers and interpreters from semantic descriptions as it already is to generate parsers from grammars? This chapter addresses such questions and gives some grounds for optimism about the development of highly practical, online semantic descriptions.

8.1 Introduction

About 500 notable programming languages (and about 2,000 lesser ones) have been designed and implemented; new programming languages are continually emerging. In almost all cases, the documentation of the language design consists mainly of a reference manual – for instance, the reference manual for the Java language is the published Java Language Specification [11]. A reference manual is intended to be sufficiently precise and complete for language implementers and programmers to grasp the form (i.e., syntax) and intended behaviour (i.e., semantics) of all legal programs in the language. In general, however, the reference manual consists mostly of ordinary (English) text, which is inherently imprecise and difficult to check for completeness.

A description of a programming language is called *formal* when it is expressed entirely in a mathematically well-defined notation: in a *programming language description language*. Formal descriptions are highly desirable, particularly since they can be checked for both precision and completeness; see Sect. 8.2 for a discussion of some further advantages over informal descriptions. The context-free syntax of a programming language can usually be described quite precisely and efficiently using a formal grammar. However, to give a formal description of the semantics of legal programs is much more challenging. Currently, language designers, implementers and programmers regard formal semantic descriptions as impractical, requiring too much effort (both for the writer and the reader) in relation to the claimed benefits.

Denotational semantics is one of the major approaches to formal description of program behaviour. Many of the main ideas of denotational semantics were expounded in a paper by Christopher Strachey [42] at the IFIP working conference on *Formal Language Description Languages* in 1964, although it was not until 1969 that Dana Scott established proper mathematical foundations [40]. Section 8.3 recalls the main features of denotational semantics. For theoreticians, it is a particularly attractive approach. However, it does have some significant pragmatic drawbacks, and only a few full-scale programming languages have been given denotational semantic descriptions.

Several researchers have addressed the perceived drawbacks of the denotational approach to semantics by developing alternative semantic description languages. In particular, Gordon Plotkin introduced a *structural* form of *operational semantics*, having substantially simpler foundations than denotational semantics while retaining some of its benefits; Eugenio Moggi proposed the use of mathematical structures called *monads* to encapsulate information in denotational semantics; and the present author developed a hybrid of denotational and operational semantics (together with David Watt) called *action semantics*, as well as a modular variant of structural operational semantics. Section 8.4 gives an overview of these alternative approaches.

The last three of the alternative approaches mentioned above all allow the semantic descriptions of individual language constructs to be formulated *independently*. This supports large-scale reuse of parts of language descriptions, which could be a crucial factor for encouraging practical use of formal semantics. Section 8.5 presents a general framework called *constructive semantics* for organising language descriptions based on semantic descriptions of individual constructs.

It is currently not easy to find semantic descriptions online. An easily accessible electronic archive of all existing significant language descriptions is badly needed. Commonly-occurring constructs could play a dual role here: for indexing existing language descriptions and as the basis for composing constructive language descriptions. Section 8.6 discusses some preliminary considerations regarding a proposed online repository for semantic descriptions.

8.2 Programming Language Descriptions

Before introducing any particular approaches to formal semantics, let us consider the main motivation for giving formal descriptions of programming languages, and contrast the successful adoption of formal grammars for describing syntax with the so far largely unfulfilled aspirations of formal semantics.

Who might benefit from precise and complete formal descriptions of programming languages? The designers of a new language need to document their completed design to inform implementers and programmers of the form and intended behaviour of legal programs in the language. Formal descriptions can be significantly more concise than the usual informal reference manuals, and should be well-suited for this purpose – provided that the description language is sufficiently clear and intelligible to all concerned – as well as for setting standards for implementations.

However, it seems unlikely that language designers would make the effort to produce a formal description of a completed design unless (1) they can use the same kind of description to record their tentative decisions during the design process and (2) the description can easily be adjusted when a new version or extension of the language is developed.

A major advantage of formal language descriptions over informal ones is the possibility of developing tools for checking their precision and completeness. Moreover, when a formal language description not only constrains but also determines the form and behaviour of legal programs, it should be possible to generate a prototype implementation of the language from it (as shown e.g. in [14, 26]) as well as other useful tools [13].

Finally, only formal language descriptions can provide a basis for sound reasoning about program behaviour. Such reasoning is relevant in the implementation and certification of safety-critical systems, and in connection with the proposed Verified Software Repository [49].

When a complete and precise formal description of a programming language is available, informal descriptions of the language can focus on giving a broad overview, explaining the underlying concepts and highlighting novel features. The vain attempt to achieve completeness and precision using natural language in reference manuals generally leads to tedious and repetitive text, where the interesting points are not easy to locate.

Complete language descriptions have to specify both the form of programs, called their *syntax* and their behaviour or *semantics*. The syntax of a language can be described without regard to the intended semantics of programs, whereas semantics cannot be described without reference to the syntax of the language. Both syntax and semantics can be further subdivided:

- Lexical syntax, concerned with how sequences of characters are grouped into tokens (words and symbols), and where layout and comments can be inserted.

- Context-free syntax, regarding how sequences of tokens are parsed, i.e., grouped into phrases.
- Context-sensitive syntax, also known as static semantics, dealing with legality constraints on parsed programs (such as compile-time type-checking).
- Dynamic semantics, modelling the relevant (observable) aspects of running legal programs.

The rest of this chapter is mainly about formal description of dynamic semantics, but let us first recall the use of formal grammars for the description of context-free syntax.

8.2.1 Syntax

BNF was introduced by John Backus [1] in 1959, for giving a formal description of the context-free syntax of the programming language Algol 60 [2]. Noam Chomsky had previously used formal grammars for describing fragments of natural languages [7], but it seems that this was the first time a formal grammar was used to describe the syntax of a programming language.

Figure 8.1 shows a fragment of the BNF of Algol 60. Each rule of the form "$\langle \mathrm{A} \rangle ::= x_1 \dots x_n$" specifies that each sequence of tokens generated by $x_1 \dots x_n$ can be generated by the non-terminal symbol $\langle \mathrm{A} \rangle$, and thereby grouped into a phrase of the corresponding sort. The sequences generated by non-terminal symbols in $x_1 \dots x_n$ have to be determined inductively, since $\langle \mathrm{A} \rangle$ might occur among the x_i, or mutual dependence might occur between several non-terminal symbols. A rule of the form "$\langle \mathrm{A} \rangle ::= w_1 \mid \dots \mid w_n$" abbreviates the set of rules "$\langle \mathrm{A} \rangle ::= w_1 \;\dots\; \langle \mathrm{A} \rangle ::= w_n$".

BNF turned out to be equivalent to Chomsky's Type-2 grammars, although Backus was apparently unaware of this relationship at the time. BNF (and minor notational variants of it) have been widely adopted for describing the context-free syntax of programming languages in reference manuals. Tools are available for generating parsers from arbitrary context-free grammars, allowing the grammars to be checked and validated. In general, when a new version or extension of a language is required, it is straightforward to adjust the

⟨adding operator⟩ ::= + | −
⟨multiplying operator⟩ ::= ∗ | / | div
⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ |
    ⟨function designator⟩ | (⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩ | ⟨factor⟩⟨primary⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩⟨multiplying operator⟩⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩⟨term⟩ |
    ⟨simple arithmetic expression⟩⟨adding operator⟩⟨term⟩

Fig. 8.1. A fragment of the BNF of Algol 60

grammar accordingly. The difficulty of producing an *unambiguous* grammar for a full-scale programming language should not be underestimated, and in practice language reference manuals tend to use ambiguous grammars supplemented by disambiguation rules; but in general, BNF has excellent pragmatic properties, as well as solid and well-studied foundations.

8.2.2 Semantics

Although the syntax of Algol 60 was described formally using BNF, the description of the intended semantics of Algol 60 programs given in the Revised Algol 60 Report [2] in 1963 was completely informal. Could one formally describe the semantics of all phrases in Algol 60, composing the semantics of each phrase from the semantics of its sub-phrases? If so, it would correspond closely to the use of BNF for syntax – and might enjoy similarly good pragmatic properties. This was the aim of the approach that later came to be known as denotational semantics. As we shall see in the next section, it is indeed possible to define semantics compositionally, although the pragmatic properties of the approach fall considerably short of those of BNF.

8.3 Denotational Semantics

Let us here look back at the development of denotational semantics and recall some of its applications in the description of full-scale programming languages. In Sect. 8.4, we shall consider some alternative semantic description languages that address pragmatic drawbacks of denotational semantics and perhaps come closer to satisfying the aim of "BNF for semantics".

Caveat: The following account is highly selective and based, to some extent, on the author's recollections from the early 1970s as a graduate student under Strachey's supervision at the Programming Research Group, Oxford.

8.3.1 Towards a Formal Semantics

In 1964, IFIP held a working conference on *Formal Language Description Languages*. The proceedings were published in 1966, including transcripts of the extensive discussions following the presented papers, many of which were of seminal importance. Strachey's paper *Towards a Formal Semantics* [42] shows that many of the basic analytic and descriptive techniques of denotational semantics had already been developed, although there was still considerable uncertainty about the details. The definitions of semantic functions given in Strachey's paper (see also his 1967 summer school lecture notes on *Fundamental Concepts in Programming Languages* [44]) are closely related to those found in many modern texts on denotational semantics.

However, all was not well regarding the mathematical foundations of the approach, as we shall see.

8.3.2 Mathematical Foundations

The major problem with the foundations of Strachey's work was that the semantics of a program was intended to be a mathematical function (from initial stores to final stores), defined by mapping programs and their parts (inductively) to the *untyped λ-calculus*. At the time, however, the untyped λ-calculus itself had no known model, such that its "applicative expressions" [42] could be interpreted as functions.[1]

A grammar for the untyped λ-calculus is given in Fig. 8.2. The notation "$\lambda x. \cdots x \cdots$" is intended to express the function of x determined by evaluating the body "$\cdots x \cdots$" for particular values of x. The λ-calculus is equipped with rules (called α-, β- and η-conversion) that are consistent with this interpretation. For example, the following λ-expression:

$$(\lambda f.\lambda g.\lambda x.((f\ x)(g\ x)))(\lambda a.\lambda b.b)(\lambda c.c) \tag{8.1}$$

can be converted to $\lambda x.x$ using the rules, which shows that it is really just another way of expressing the function that always returns its argument. Remarkably, the λ-calculus is Turing-complete: all computable functions (on the natural numbers) can be expressed in it.

The problem was that some other λ-expressions could not be interpreted as ordinary set-theoretic functions at all. For instance, consider the following expression, named Y:

$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)). \tag{8.2}$$

Y has the property that $Y(f)$ is convertible by the rules of the λ-calculus to $f(Y(f))$, so $Y(f)$ is a fixed point of f, i.e., a value x such that $f(x)$ returns x. Strachey needed some way of computing fixed points for expressing the semantics of iterative statements and recursive procedures.

However, the definition of Y involves application of the argument x to itself: this is only possible if any element of the set of arguments can be regarded as a function on that set, which requires a set D isomorphic to a set of functions from D to D. But by a variant of Cantor's Theorem, the cardinality of the set of all functions from D to D is strictly greater than that of D (provided that D contains at least two values). For this reason, the λ-expression

$$\begin{array}{lcll}
\langle\text{exp}\rangle & ::= & \langle\text{id}\rangle & \text{– reference to an identifier} \\
 & \mid & \lambda\langle\text{id}\rangle.\langle\text{exp}\rangle & \text{– abstraction on an identifer} \\
 & \mid & \langle\text{exp}\rangle\langle\text{exp}\rangle & \text{– application to an argument} \\
 & \mid & (\langle\text{exp}\rangle) & \text{– grouping} \\
\langle\text{id}\rangle & ::= & a \mid b \mid c \mid \ldots & \text{– identifiers}
\end{array}$$

Fig. 8.2. A grammar for the λ-calculus

[1] In the known models, expressions were interpreted as equivalence classes of terms.

Y was called "paradoxical", and for decades, it was thought impossible to find a model for any λ-calculus allowing such self-applicative expressions when λ-abstractions are to be interpreted as mathematical functions.

In the autumn of 1969, Scott visited Strachey in Oxford while on sabbatical. He had apparently managed to persuade Strachey to abandon the untyped λ-calculus and start using a typed version, which was known to have a model. Scott gave some seminars about the typed λ-calculus but then suddenly realised that the constructions involved could be extended to provide a model for the untyped λ-calculus too!

The basic idea of Scott's model is quite easy to grasp: he constructed a set D isomorphic to the set of all *continuous* functions from D to D. This set was partially ordered by a notion of approximation and each function was the limit of an infinite series of finite approximations. All λ-expressions could be interpreted as continuous functions on D, giving the desired model. (The cardinality of this set of continuous functions on D is the same as that of D, and Cantor's theorem is not applicable.)

Before long, Scott had founded his theory of *domains* [40], which provides solutions (up to isomorphism) not only of the equation $D = (D \to D)$, but also of any set of mutually-recursive domain definitions. Furthermore, he showed that any continuous function f from any domain to itself has a least fixed point, given by the limit of the series $f^n(\bot)$ of finite iterations of f, where $\bot$ is the least element of the domain.

Now that domains could be defined, Scott and Strachey finally dropped the untyped λ-calculus in favour of a typed λ-*notation*: the domain over which each identifier ranges is given and each term is interpreted directly in the appropriate domain, without reference to any conversion rules [41]. Moreover, the definition of the relevant domains was now regarded as an important part of the semantic description of a programming language [43].

The least fixed point operation is provided on every domain, making the paradoxical λ-expression Y redundant. This provided satisfactory mathematical foundations for the kind of semantic description that Strachey had originally proposed, and for a while, the Scott–Strachey approach was referred to simply as "mathematical semantics". The name was later changed to *denotational* semantics, reflecting that each phrase of a program is mapped to a mathematical entity – its *denotation* – modelling the contribution of the phrase to the overall behaviour of any program in which it might occur.

8.3.3 Development

In Strachey's Programming Research Group in Oxford in the early 1970s, there was a strong belief that the denotational approach was indeed "BNF for semantics". Surely every major programming language would soon be given its denotational semantics, and the semantics of a new language would be worked out before the language had been implemented – perhaps leading to better language design?

Some encouraging progress was indeed being made. For instance, it had been realised how to give a denotational semantics to unstructured jumps to labels using continuations (i.e., functions representing the behaviour of a program starting from particular points), and a so-called continuation-passing style had been adopted: denotations were now usually functions of continuations [45]. The denotational semantics of significant languages, such as Algol 60 and Algol 68, had been worked out [21, 24]. A group at the IBM Laboratory in Vienna had dropped VDL (an operational approach they had previously used) in favour of VDM, a variant of denotational semantics, and had used VDM to give a denotational definition of (sequential) PL/I [3, 17]. Significantly, VDM avoided the use of continuations and introduced special notation for dealing with abrupt termination due to "exits" [16,36]. A system for generating prototype implementations from (grammars and) denotational descriptions was already being implemented in 1975 [25, 26].

It seemed that denotational semantics was becoming quite well established, when, tragically, Strachey died (at the age of only 58, after a brief illness). Suddenly, without its charismatic leader, the Programming Research Group had less optimism about new developments. It was left to other groups to pursue further practical application of denotational semantics to the description of programming languages.

8.3.4 Applications

Towards the end of the 1970s, the development of the language Ada had been initiated by the Department of Defense (USA).The development contract, won by a European consortium, stipulated that the documentation of the language was to include a formal semantics. In bidding for the contract, the consortium had illustrated how the semantics of a mini-language could be described in the denotational style, and checked and validated using the Semantics Implementation System (SIS) [28]. In the initial phase of the project, the denotational semantics of a sublanguage of Ada was given in continuation-passing style. However, at some stage, it was decided to switch to using a variant of Ada itself to express the semantics of Ada. The variant was a higher-order extension of a purely functional sublanguage of Ada, but did not include λ-expressions, so each continuation had to be defined by a series of named function declarations (which made the description very lengthy). Anyway, at least a formal semantics of Ada was being developed at the same time as its implementation, which was encouraging, although the published semantics covered only the sequential part of the language, omitting all constructs concerned with tasking [10].

An alternative semantics of Ada was given using VDM by Dines Bjørner et al. at the Technical University of Denmark [5]. This description was subsequently used by Danish Datamatics Center (DDC) as the basis for the development of an Ada compiler. The project was a notable success: the compiler was not only one of the first to market, but also of very high quality. Halfway through the project, however, the design of Ada was significantly revised. Ideally, the denotational description should have been updated to reflect the changes, but this was regarded as too tedious and time-consuming (partly due to the lack of tool support for VDM at the time): comments were merely pencilled-in at the places where changes should be made, and these comments were taken into account during the subsequent development of the compiler.

8.4 Pragmatic Aspects

The projects to give a denotational semantics of Ada (see above) revealed that although denotational semantics had solid theoretical foundations, it also had some pragmatic shortcomings. In particular, it was an immense effort to give a complete denotational description of even the sequential sublanguage of Ada, and difficult to modify the semantic description to accommodate the evolution of the language design. The λ-notation and domain equations used in denotational descriptions were also unfamiliar to most programmers.

In this section, we shall take a brief look at some of the alternative approaches that were developed directly in response to the perceived pragmatic problems of denotational semantics; an overview including further approaches is available elsewhere [34]. For illustration, we shall consider the semantics of arithmetic expressions for addition: Exp_1+Exp_2. Let the semantic function *sem* map expressions *Exp* to their denotations.

Strictly speaking, semantic functions are defined on *abstract syntax trees* that reflect the intended compositional phrase structure of program texts. This is easy to overlook, since sets of abstract syntax trees are usually specified by BNF-like grammars using concrete symbols from program texts as terminal symbols. For instance, the grammar specifying the abstract syntax trees of our illustrative arithmetic expressions would include:

$$\mathit{Exp} ::= \cdots \mid \mathit{Exp}\texttt{+}\mathit{Exp} \mid \cdots$$

The terminal symbols distinguish between different kinds of nodes and make the relationship between program texts and the corresponding trees rather obvious (up to disambiguation of grouping).

8.4.1 Scott–Strachey Semantics

A conventional denotational description of a pure functional language might take the domain of denotations for expressions to be the domain of (continuous) functions from Env to Val, written Env $\to$ Val, where an element ρ of the auxiliary domain Env is an *environment* representing the current bindings of identifiers. The denotation of the expression Exp_1+Exp_2 would then be defined thus:

$$\mathit{sem}[\![\mathit{Exp}_1\texttt{+}\mathit{Exp}_2]\!] = \lambda\rho{:}\mathrm{Env}.\ (\mathit{sem}[\![\mathit{Exp}_1]\!]\rho + \mathit{sem}[\![\mathit{Exp}_2]\!]\rho) \qquad (8.3)$$

(The occurrence of "+" on the left of the equation is a symbol in a program, whereas the "+" on the right is the usual mathematical operation on numbers.)

However, if the described language includes assignment statements and expressions include variable identifiers, the above domain of expression denotations is inadequate (assuming that bindings are unaffected by statements). We might change it to Env $\to$ (S $\to$ Val), where an element σ of the auxiliary domain S is a *store* representing the values stored at *locations* corresponding to variable identifiers. The above equation would need a minor reformulation to match the new domain of denotations:

$$sem[\![Exp_1\texttt{+}Exp_2]\!] = \lambda\rho{:}\mathrm{Env}.\ \lambda\sigma{:}\mathrm{S}.\ (sem[\![Exp_1]\!]\rho\,\sigma + sem[\![Exp_2]\!]\rho\,\sigma) \tag{8.4}$$

If expression evaluation can itself cause assignment, the above domains would still be inadequate. We might then take the domain of denotations to be Env $\to$ (S $\to$ (Val $\times$ S)), but a bigger reformulation of the equation would be needed (let us, henceforth, leave types such as $\rho{:}\mathrm{Env}, \sigma{:}\mathrm{S}$ implicit):

$$\begin{aligned} sem[\![Exp_1\texttt{+}Exp_2]\!] = \lambda\rho.\lambda\sigma.\ &(\lambda(v_1,\sigma_1).\\ &\quad(\lambda(v_2,\sigma_2).\ (v_1+v_2,\sigma_2))\\ &\quad(sem[\![Exp_2]\!]\rho\,\sigma_1))\\ &(sem[\![Exp_1]\!]\rho\,\sigma) \end{aligned} \tag{8.5}$$

Writing $f * g$ for $(\lambda(v,\sigma).f\ v\ \sigma)\circ g$ and P for $\lambda v.\lambda\sigma.(v,\sigma)$ following Scott and Strachey [41], the above can be formulated without explicit reference to the store σ:

$$\begin{aligned} sem[\![Exp_1\texttt{+}Exp_2]\!] = \lambda\rho.\ &(\lambda v_1.(\lambda v_2.\ P(v_1+v_2))\\ &\qquad * sem[\![Exp_2]\!]\rho)\\ &* sem[\![Exp_1]\!]\rho \end{aligned} \tag{8.6}$$

A perhaps more attractive alternative in conventional denotational semantics would be to use *continuation-passing style*, letting the domain of denotations be Env $\to$ (K $\to$ C) where K = Val $\to$ C and C = S $\to$ S. Then the denotation would be defined by:

$$\begin{aligned} sem[\![Exp_1\texttt{+}Exp_2]\!] = \lambda\rho.\lambda\kappa.\ &sem[\![Exp_1]\!]\ \rho\ (\lambda v_1.\\ &sem[\![Exp_2]\!]\ \rho\ (\lambda v_2.\\ &\kappa(v_1+v_2))) \end{aligned} \tag{8.7}$$

Although continuation-passing style is sometimes regarded as a standard style to use for denotational semantics, it is inadequate for describing languages that involve non-determinism or concurrent processes.

Observe how the pattern of notation required for defining the denotation of each construct depends strongly on the details of the domain definitions, which, in turn, depend on the features of the entire language. Extending a language with a new construct may require changing some of the domain definitions, and then reformulating *all* the definitions of denotations that involve those domains [27].

8.4.2 Monadic Semantics

In 1989, Moggi proposed the use of mathematical structures called *monads* to encapsulate the definitions of domains of denotations [23]. A monad M defines, for all domains D, D' of computed values:

- A domain $M(D)$ whose elements represent computations that (when they terminate normally) compute values in D; for example, expressions normally compute values in a domain Val, so $M(\mathrm{Val})$ would be a domain of expression denotations, and $M(D)$ could be defined by:

$$M(D) = \mathrm{Env} \to (\mathrm{S} \to (D \times \mathrm{S})) \tag{8.8}$$

- A function "return" in $D \to M(D)$ that maps each value v in D to an element in $M(D)$ representing the trivial computation, which immediately terminates with v as the result, e.g.:

$$\mathrm{return} = \lambda v.\lambda\rho.\lambda\sigma.(v, \sigma) \tag{8.9}$$

- A binary function "$\gg\!=$" in $M(D) \times (D \to M(D')) \to M(D')$, such that $m \gg\!= f$ represents the computation that starts with m, then continues with the computation obtained by applying f to the value computed by m, e.g:

$$m \gg\!= f = \lambda\rho.(\lambda(v, \sigma).f\ v\ \rho\,\sigma) \circ m\ \rho \tag{8.10}$$

Regardless of the structure of $M(D)$, the denotation of Exp_1+Exp_2 can be expressed as follows using the functions defined by the monad M:

$$\begin{aligned} sem[\![Exp_1\texttt{+}Exp_2]\!] = \ & sem[\![Exp_1]\!] \gg\!= \lambda v_1. \\ & sem[\![Exp_2]\!] \gg\!= \lambda v_2. \\ & \mathrm{return}(v_1 + v_2) \end{aligned} \tag{8.11}$$

Notice, however, that the order in which the two expressions are evaluated has to be specified, even when computations have no side-effects.

Moggi also showed how to define monads incrementally, using *monad transformers* that add new features (e.g., environments, stores, exceptions, continuations) to a previously-defined monad [23]. A monad transformer T maps any monad M to a monad TM, defining $TM(D)$ in terms of $M(D)$ and redefining the functions for return of values and composition accordingly. In general, the transformer also defines some auxiliary functions that are associated with the new feature. For example, the monad transformer that adds a store, with $TM(D) = \mathrm{S} \to (M(D) \times \mathrm{S})$, also defines functions for inspecting and updating particular locations.

The use of monads in denotational descriptions addresses many of their pragmatic problems, and largely eliminates the need for reformulation of the equations defining the denotations when the described language is extended. However, the need to understand the use of monads is a further complication of the denotational approach, and appears to be a hindrance for programmers (except perhaps for those familiar with functional programming languages, where monads are sometimes used). See also [36].

8.4.3 Action Semantics

The *action semantics* framework was developed by the present author, in collaboration with Watt, in the second half of the 1980s [30, 38, 51]. As in denotational semantics, inductively-defined semantic functions map phrases to their denotations, only here the denotations are so-called *actions*. The notation for actions is itself defined operationally [30].

The universe of actions supports specification not only of control and data flow, but also of scopes of bindings, effects on storage and interactive processes, allowing a simple and direct representation of many programming concepts.

Regardless of what features expressions might have (apart from computing values when their evaluations terminate normally) such as side-effects or concurrent processing, the action semantics of Exp_1+Exp_2 can be expressed as follows using standard action notation:

$$sem[\![Exp_1\texttt{+}Exp_2]\!] = (sem[\![Exp_1]\!] \text{ and } sem[\![Exp_2]\!]) \text{ then give}(+) \tag{8.12}$$

The combinator "A_1 and A_2" expresses unspecified (possibly interleaved) order of execution of the actions A_1, A_2. The combined action concatenates the values (sequences) given by its subactions. In contrast, the combinator "A_1 then A_2" expresses sequential composition of the actions, with the values given by A_1 being passed to A_2. The primitive action "give(+)" gives the result of applying the data operation "+" to the two values passed to it. In contrast to monadic semantics, it is just as easy to leave the order in which the two expressions are evaluated unspecified (using "and") as it is to specify it (using an action combinator "and then"). If A_1 or A_2 has side-effects, the unspecified order of interleaving in "A_1 and A_2" may give rise to different results, i.e., non-determinism; both "A_1 then A_2" and "A_1 and then A_2" preserve determinism.

The use of action notation in denotational descriptions addresses many of their pragmatic problems and like monads, eliminates the need for reformulation of the equations defining the denotations when the described language is extended. One obvious drawback is the need to become familiar with action notation; moreover, the original version of action semantics was based on an unorthodox algebraic specification framework called Unified Algebras [30], and the operational definition of action notation was difficult to work with, although both these drawbacks have subsequently been eliminated [19].

8.4.4 Structural Operational Semantics

In 1981, Plotkin introduced *SOS*, a structural form of operational semantics for programming languages, having substantially simpler foundations than denotational semantics while retaining some of its benefits [39]. This was a direct reaction to pragmatic concerns regarding accessibility of denotational semantics to programmers. The main aim was to provide a simple and direct approach, allowing concise and comprehensible semantic descriptions based on elementary mathematics.

Computations are represented in SOS as sequences of transitions in *labelled transition systems*. An SOS defines transition systems for all parts of programs, as well as for complete programs. The possible transitions between configurations are specified by giving *axioms* and *inference rules*.

Assuming that the value V of an expression Exp depends only on an environment ρ representing the current bindings of identifiers, the SOS rules for sequential expression evaluation are written thus:[2]

$$\frac{\rho \vdash Exp_1 \longrightarrow Exp_1'}{\rho \vdash Exp_1 + Exp_2 \longrightarrow Exp_1' + Exp_2} \tag{8.13}$$

$$\frac{\rho \vdash Exp_2 \longrightarrow Exp_2'}{\rho \vdash V_1 + Exp_2 \longrightarrow V_1 + Exp_2'} \tag{8.14}$$

$$\rho \vdash V_1 + V_2 \longrightarrow V \quad (\text{if } V = V_1 + V_2) \tag{8.15}$$

Transitions derived using (8.13) above allow Exp_1 to be gradually evaluated to a value V_1, whereupon transitions derived using (8.14) allow Exp_2 to be gradually evaluated to a value V_2; finally the transition derived using (8.15) replaces the addition expression $V_1 + V_2$ by the value V given by adding V_1 and V_2.[3] The order of evaluation can be left unspecified by generalising (8.14), allowing expressions Exp_1 instead of only values V_1.

As with conventional denotational semantics, reformulation of the semantic description of addition expressions is needed if the described language is extended, so that expressions may inspect the values assigned to variables or cause assignments themselves. For example, (8.13) above becomes:

$$\frac{\rho \vdash \langle Exp_1, \sigma \rangle \longrightarrow \langle Exp_1', \sigma' \rangle}{\rho \vdash \langle Exp_1 + Exp_2, \sigma \rangle \longrightarrow \langle Exp_1' + Exp_2, \sigma' \rangle} \tag{8.16}$$

Notice that there are no explicit labels on the transitions in the above rules: labels are normally used explicitly in SOS only in connection with communication and synchronization between concurrent processes, and do not occur at all with transitions for purely sequential programming constructs.

An alternative "big-step" style of SOS, also called "natural" semantics [18] and used for the definition of Standard ML [22], represents computations by (unlabelled) transitions directly from initial to final configurations. The following rule specifies the evaluation of the same expressions as the "small-step" SOS rules given above:

$$\frac{\rho \vdash \langle Exp_1, \sigma \rangle \longrightarrow \langle V_1, \sigma' \rangle,\ \rho \vdash \langle Exp_2, \sigma' \rangle \longrightarrow \langle V_2, \sigma'' \rangle,\ V = V_1 + V_2}{\rho \vdash \langle Exp_1 + Exp_2, \sigma \rangle \longrightarrow \langle V, \sigma'' \rangle} \tag{8.17}$$

Note that only terminating computations are represented by transitions in the big-step style of SOS.

[2] $\rho \vdash Exp \longrightarrow Exp'$ abbreviates $\langle \rho, Exp \rangle \longrightarrow \langle \rho, Exp' \rangle$.

[3] Side-conditions on rules are often written as premises, for notational convenience.

8.4.5 Modular SOS

SOS, as illustrated above, is significantly simpler than denotational semantics regarding foundations. The transition rules are somewhat less concise than the corresponding equations that define denotations, but they are quite straightforward to read (and write). Unfortunately, however, SOS is no better than conventional denotational semantics with respect to the reformulation required when extending the described language.

In 1997, Keith Wansbrough gave a monadic semantics for the action notation used in action semantics [50]. In contrast to the original SOS description of action notation, this monadic semantics was quite modular and could be extended without reformulation. This drew attention to the question of whether the modularity of SOS descriptions could be significantly improved or not – a question that had been noted by Plotkin in 1981 [39], but apparently not pursued in the meantime.

How might rules in SOS be formulated in such a way that they would never need changing when the described language is extended? Suppose that configurations are restricted to syntax (as when using SOS for describing concurrency calculi such as CCS) and we generally make the labels on transitions explicit. Then the rules for expression evaluation would look like this:

$$\frac{\mathit{Exp}_1 \xrightarrow{\alpha} \mathit{Exp}'_1}{\mathit{Exp}_1\texttt{+}\mathit{Exp}_2 \xrightarrow{\alpha} \mathit{Exp}'_1\texttt{+}\mathit{Exp}_2} \tag{8.18}$$

$$\frac{\mathit{Exp}_2 \xrightarrow{\alpha} \mathit{Exp}'_2}{V_1\texttt{+}\mathit{Exp}_2 \xrightarrow{\alpha} V_1\texttt{+}\mathit{Exp}'_2} \tag{8.19}$$

$$V_1\texttt{+}V_2 \longrightarrow V \quad (\text{if } V = V_1 + V_2) \tag{8.20}$$

This variant of SOS is called *Modular SOS* [31, 32]. The key insight is to exploit the labels α on transitions to propagate processed information, such as environments ρ and stores σ. For instance, α might be simply an environment ρ (representing the current bindings), or a tuple of the form (ρ, σ, σ'), (where σ represents the store before the transition and σ' the store after it), or have further components. Rules (8.18) and (8.19) above specify that all the information processed when one of the subexpressions makes a transition is exactly the same as when the enclosing expression makes the corresponding transition. Thus, if the subexpression and the enclosing expression refer to the environment ρ, they both refer to the same environment, and similarly for the store σ; and if one of the subexpressions causes a change from σ to σ', so does the enclosing expression.

Labels on adjacent transitions in computations are required to be *composable*. Suppose that $\mathit{Exp} \xrightarrow{\alpha_1} \mathit{Exp}'$ and $\mathit{Exp}' \xrightarrow{\alpha_2} \mathit{Exp}''$, with $\alpha_1 = (\rho_1, \sigma_1, \sigma'_1)$ and $\alpha_2 = (\rho_2, \sigma_2, \sigma'_2)$. Composability implies that $\rho_1 = \rho_2$ and $\sigma'_1 = \sigma_2$.

The reader may have noticed that the label on the transition specified by (8.20) above is omitted. This specifies that the label is constrained to be *unobservable.* When labels contain pairs of stores σ, σ', representing the possibility of assignment changing the store, unobservability requires $\sigma = \sigma'$.

The difference between the foundations of Modular SOS and those of conventional SOS is slight: the set of labels is now equipped with a composability relation and an unobservability predicate, and labels on adjacent transitions are required to be composable. In their full generality, the labels are the morphisms of a category, but in practice, it is sufficient for them to be ordinary record values (as found in several programming languages). Many rules (including almost all rules for expression evaluation) are completely independent of the structure of labels; rules which need to refer to particular components can do so without mentioning irrelevant components, using formal notation for elision [32].

An SOS can easily be converted to Modular SOS, simply by moving components representing processed information (such as ρ, σ, σ') from configurations to labels. The result is a huge improvement in pragmatics: the rules for each individual language construct can be given as a separate, independent module and never need reformulating when the constructs are combined in the same language.

8.5 Constructive Semantics

The previous section considered some variants of denotational and operational semantics: monadic semantics, action semantics and Modular SOS. These variants all allow the semantics of individual language constructs to be specified independently, which implies that the semantic descriptions never need reformulation when the described language is extended.

Constructive semantics is a way of organising language descriptions to support explicit, large-scale reuse of their parts [33, 37]. It was initially developed for use with action semantics, in collaboration with Kyung-Goo Doh [9], but it can be used with any kind of semantics that allows *independent* specification of the semantics of individual constructs: the formulation of the semantic description of a construct, such as Exp_1+Exp_2, must not depend on whether expressions might have side-effects, be non-deterministic, spawn concurrent processes, etc., nor on whether other sorts of constructs (statements, declarations, etc.) occur in the same language. The main differences between constructive semantics and conventional semantic descriptions concern the overall modular structure and the notation used for the syntax of constructs.

8.5.1 Modular Structure

The modular structure of a conventional semantic description is to specify the entire *syntax* of the described language in one module, *auxiliary entities* for use in the entire semantics in a second module and the *semantics* of all constructs together in a third module. Each of these modules may have submodules: for example, the syntax and semantics modules may be divided into submodules covering expressions, statements, declarations, etc.; the auxiliary entities may be divided into separate definitions of data types for environments, stores, arrays, records, etc. In practice, however, there is never any *explicit* reuse of submodules between semantic descriptions of different languages – even when one language is an extension or a new version of another. There may still be plenty of *implicit* reuse due to copying parts of specifications, but no indication is given of where the copied version originated from and whether it has subsequently been edited or not.

The modular structure of a *constructive* semantics is radically different from that of conventional semantic descriptions: it insists on a separate, independent, named module for each *sort* of construct (e.g., expressions) and for each *individual construct* (e.g., a conditional expression). The module for a sort of construct must make no assumptions at all about which constructs are included in that sort. The module for an individual construct (implicitly) imports the module for the sort of the construct and for the sort of each component of the construct. Both kinds of modules may import auxiliary modules for any required data types. The semantics of a language can then be given by adding a single module that imports the modules for all the constructs to be included in the language.

The main panel in Fig. 8.3 shows the modular structure of a constructive action semantics of a (very) small illustrative sublanguage of Standard ML. The top module (SmallML) imports the modules for a couple of declaration constructs and four expression constructs; each of these construct modules imports the modules for the relevant sorts of constructs (declarations, identifiers and expressions) together with modules for data (representing functions

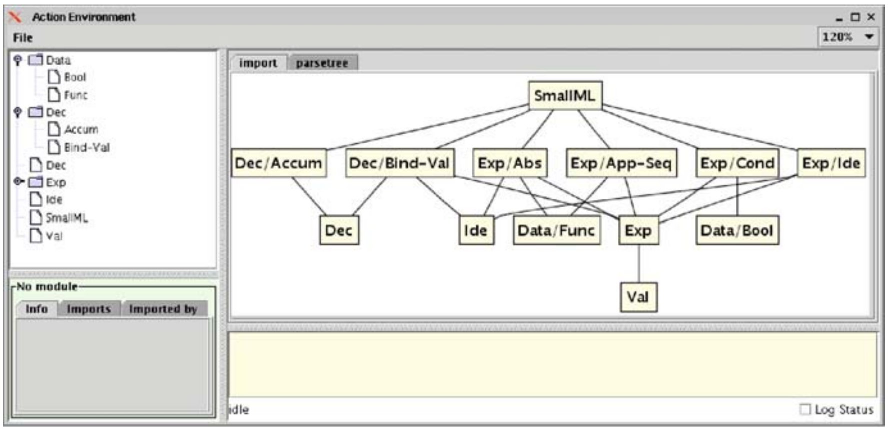


Fig. 8.3. Modular structure in constructive semantics

and boolean values). Notice that the modular structure for the individual constructs is completely flat and does *not* involve any mutual dependence.

The constructive action semantics module for the sort *Exp* of expressions is shown in Fig. 8.4. It is written in a meta-notation called Action Semantics Description Formalism (ASDF) (see Sect. 8.5.3). The requirement *E* : *Exp* specifies that the variable *E* (and all variables formed by adding digits and primes to *E*) ranges over the sort *Exp*. The sort *Datum* is provided by action notation and includes all individual data items that can be computed by actions; *Datum* ::= *Val* specifies that *Datum* includes *Val* as a subsort. The module also specifies that evaluate is a semantic function, mapping expressions of sort *Exp* to actions which (when they terminate normally) give elements of sort *Val*.

The module for an individual conditional expression construct is shown in Fig. 8.5. It specifies both the abstract syntax of the construct (the choice of notation for abstract syntax will be explained in Sect. 8.5.2) and its action semantics. Requiring *Val* to include *Boolean* as a subsort ensures the *possibility* of evaluate *E1* giving a boolean value; the reference to "the boolean" has the effect of a dynamic type check on the value, being undefined when the value is not a boolean. The action check *Y* terminates normally when its argument *Y* yields true and fails when it yields false. (The action maybe is not so important: it merely fails when the value yielded by the boolean is undefined, thereby eliminating the possibility of abnormal termination due to evaluating conditional expressions with non-boolean conditions.)

```
Module Exp
requires
  E : Exp
  Datum ::= Val
semantics  evaluate: Exp → Action & giving Val
```

Fig. 8.4. A module for the action semantics of arbitrary expressions

```
Module Exp/Cond
syntax     Exp ::= cond(Exp, Exp, Exp)
requires   Val ::= Boolean
semantics  evaluate cond(E1, E2, E3) =
        evaluate E1 then
        maybe check the boolean
        then evaluate E2
        else evaluate E3
```

Fig. 8.5. A module for the action semantics of a conditional expression

8.5.2 Notation for Syntax

To maximise the reusability of the modules describing the semantics of individual constructs, constructive semantics uses a neutral, language-independent notation for syntax, and avoids using the reserved words and symbols of particular languages. For example, the construct for a conditional expression should have a notation such as $\mathrm{cond}(Exp, Exp, Exp)$, rather than the usual language-dependent syntax "**if** *Exp* **then** *Exp* **else** *Exp*" or "*Exp* ? *Exp* : *Exp*". When John McCarthy introduced the concept of abstract syntax [20], his notation was independent of concrete syntax; however, he was using so-called *analytic* abstract syntax based on predicates and selector functions, avoiding constructor functions altogether – in fact the terms used as arguments to predicates and selectors were strings, rather than trees. Abstract notation for syntactic constructor functions was introduced in VDM [3].[4]

Constructs such as the conditional expression occur in many different languages; introducing a neutral name for a language-independent construct emphasises its commonality. Variants of constructs having different semantics need distinct names. For instance, the conditional expression with a boolean condition may be regarded as standard and named "cond", and that with a condition testing for nonzero values could be named "cond-nonzero". (The semantic distinction between these two constructs shows up not only in their static semantics, where their conditions have different required types, but also in their dynamic semantics, when the value computed by the condition is tested.)

The conventional use of language-dependent notation for syntax in semantic descriptions has the advantage that the semantics can be regarded as applying directly to program texts. With constructive semantics, the mapping from (parsed) program texts to language-independent constructs has to be specified explicitly. In practice, however, a conventional semantic description often involves mapping a full programming language to a kernel sublanguage, so that only the semantics of the kernel language has to be specified directly. It is just as easy to map the full language to (combinations of) language-independent constructs. In fact the specification of the mapping resembles a conventional denotational semantics, except that the denotations are simply combinations of language-independent constructs. Provided that the semantics of these constructs are already familiar to the reader, inspection of the specification of the mapping is sufficient to understand the semantics of the programming language. For instance, the following fragment of such a mapping confirms that the if-then-else expression of the described programming language corresponds directly to the standard language-independent conditional expression:

$$[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3]\!] = \mathrm{cond}([\![E_1]\!], [\![E_2]\!], [\![E_3]\!]) \tag{8.21}$$

[4] In contrast to VDM, VDL used unlabelled, unordered abstract syntax trees with branches labelled by selector functions.

8.5.3 Tool Support

A constructive semantics of a programming language may involve hundreds of (tiny) modules. Tool support is essential to ensure consistency of notation and well-formedness of specifications.

Rudimentary tool support for constructive action semantics was initially provided [9] by writing action semantic descriptions directly in *ASF+SDF* and using the ASF+SDF *Meta-Environment* [47]. ASF+SDF is a general-purpose, executable, algebraic specification language. As the name indicates, it is a combination of two previous formalisms: the Algebraic Specification Formalism (ASF) [4, 48] and the Syntax Definition Formalism (SDF) [12]. SDF is used to define the syntax of a language, whereas ASF is used to define conditional rewrite rules; the combination ASF+SDF allows the syntax defined in the SDF part of a specification to be used in the ASF part, thus supporting the use of so-called "mixfix notation" in algebraic specifications. ASF+SDF allows specifications to be divided into named modules, facilitating reuse and sharing (as in SDF).

A more concise notation called *ASDF* was subsequently supported by the *Action Environment*, developed in collaboration with Mark van den Brand and Jørgen Iversen [46]. Figure 8.3 (in Sect. 8.5.1) shows the Action Environment in use on a (very) small example; it scaled up without difficulty to support a much larger case study of constructive action semantics: a description of Core ML [15]. Iversen has also implemented a compiler generator based on constructive action semantics and used it to generate a Core ML compiler that produces reasonably efficient code [14].

Some tool support for constructive Modular SOS descriptions has been implemented in Prolog, for use in teaching formal semantics [35]. Parsing concrete programs and translating them to abstract constructs are specified together using Definite Clause Grammars (DCGs), which are built into Prolog. For example, the parsing and translation of ML conditional expressions can be specified by the following DCG rule:

```
exp(cond(E1,E2,E3)) -->
    "if", i, exp(E1), i, "then", i, exp(E2), i,
    "else", i, exp(E3)
```

(The non-terminal "`i`" stands for ignored text, such as layout and comments.) The main drawback of DCGs for parsing programming languages is that left-recursive rules give rise to non-termination.

In the initial version of the tool, the Modular SOS rules for each abstract construct had to be translated manually to Prolog; in later versions, the Modular SOS rules were written in *MSDF* (a concise notation similar to ASDF, see Fig. 8.5), then parsed and translated to Prolog by the tool. The following module shows how Modular SOS rules for evaluating conditional expressions can be specified in MSDF:

```
Exp ::= cond(Exp,Exp,Exp)

Value ::= Boolean

                E --X-> E'
-------------------------------------
cond(E,E1,E2):Exp --X-> cond(E',E1,E2)

cond(true,E1,E2):Exp  ---> E1

cond(false,E1,E2):Exp ---> E2
```

Loading the resulting Prolog files provided an interpreter for the described language.

Another tool for MSOS called *MMT* [6] has been implemented in the Rewriting Logic system Maude [8] by Christiano Braga and Fabricio Chalub. The notation used for specifying MSOS rules is similar to MSDF. The MMT web site[5] provides several case studies of constructive MSOS, including descriptions of sublanguages of ML and Java, and of the pure functional programming language Mini-Freja.

8.6 Future Directions: Semantics Online

A graduate student (who had been reading a survey of descriptive techniques in denotational semantics [29]) emailed me:

> Lately I've been interested in finding a denotational semantics (using the CPS style employed in the Scheme standard) for pattern matching. Do you happen to know where such a development exists?

I did not know and I realised that I could not even refer him to a good website where he might be able to find one! So I sent him a rough sketch of how I would formulate such a semantics myself.

The ease of sketching a semantic description of a particular language construct could be regarded as a sign of the success and maturity of formal semantics. However, the difficulty of locating existing semantic descriptions online is quite embarrassing: as computer scientists, we should surely be leaders in archiving and presenting our research results online, rather than lagging behind.

A new initiative is called for: to establish a digital library of formal descriptions of programming languages, i.e., an appropriately indexed, searchable and easily accessed *online repository*. The main aims are:

[5] `http://maude-msos-tool.sourceforge.net/`

- To collect and enable access to older language descriptions that are presently unavailable electronically.
- To collect and facilitate browsing and searching more recent language descriptions that are already available electronically.
- To encourage and support the development of new language descriptions.

Although the initial emphasis is to be on providing easy access to (static and dynamic) semantic descriptions, a subsidiary goal is to augment them by grammars for parsing the described languages and producing the corresponding abstract syntax trees.

A semantic description of a particular language in any framework is naturally indexed by the individual syntactic constructs of the language; the names of the underlying language-independent constructs could provide appropriate indexing terms for the entire repository. Modular semantic frameworks, moreover, allow the description of each construct to be formulated independently, and provided as a separate item in a repository.

Because modular semantic descriptions encourage large-scale reuse, they could significantly lower the effort required to develop complete language descriptions: the descriptions of common constructs are already available and can simply be cited, and only those novel or unusual constructs that are not already included in the repository remain to be described. This, together with the inherent support for evolution provided by truly modular semantic frameworks, might encourage the designers of a new language to provide a formal description of its semantics together with that of its syntax.

A long-term aim is to be able to generate useful reliable language prototyping and verification tools (such as those needed in connection with the UKCRC Grand Challenge on Dependable Systems Evolution) from language descriptions in the repository [13]; this may ultimately attract the interest and involvement not only of expert semanticists and language designers, but also of all computer scientists concerned with programming languages. Generation of compilers producing reasonably efficient code has already been shown feasible, based on constructive action semantics [14].

The production of a comprehensive online repository of formal semantic descriptions should dramatically increase the visibility of present research in this area of computer science, as well as preserve older semantic descriptions which might otherwise disappear altogether. The author has recently surveyed the published semantic descriptions of major programming languages in VDM [36]. These descriptions have been scanned to pdf and are to be used in the development of a proof-of-concept prototype of the proposed repository.

Properly-documented validation of the descriptions would give confidence in their accuracy and completeness, and could be included in the repository, along with theorems about semantic equivalences based on the descriptions. Contributions would need to be strictly reviewed; both contributors and reviewers could be duly credited for their efforts. With proper quality control and a well-designed web interface for searching and browsing the repository, we would be able to tell our students and colleagues where they can find semantic descriptions for themselves.

8.7 Conclusion

Let us summarise the main points of this chapter:

- One of the major approaches to formal semantics is denotational semantics. We have shown why such a theoretically attractive approach is generally regarded as impractical for describing full-scale programming languages. Use of monads in denotational descriptions allows individual constructs to be described independently, but the unfamiliarity of the approach to ordinary programmers is still a drawback.
- Action semantics is a hybrid of denotational and operational semantics and shares some of the pragmatic advantages of monadic semantics. Here, the unfamiliarity of action notation appears to be the main drawback.
- Structural operational semantics was developed to provide a simple and accessible approach, but it does not support independent description of individual constructs. Modular SOS removes this drawback, without significant complications to the foundations.
- Constructive semantics is a general framework for organising semantic descriptions, so as to allow large-scale reuse. It can be used with any approach that allows independent description of individual constructs. Some prototype tool support for constructive action semantics has been developed, including a compiler generator.
- It is currently not easy to find semantic descriptions online. An easily accessible electronic archive of all existing significant language descriptions is badly needed. Online descriptions of individual constructs would support development of constructive semantic descriptions.

The recent development of constructive action semantics and Modular SOS, together with the emerging tool support for constructive semantic descriptions, give grounds for optimism for the provision of accessible online semantic descriptions of many major programming languages, as well as generation of efficient compilers and interpreters from these descriptions.

Acknowledgements

The author thanks the anonymous referees and Paul Boca for helpful comments on drafts of this paper, and the organisers of the BCS-FACS Seminar Series for the invitation to give the presentation on which this chapter is based.

References

1. J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GRAMM conference. In *Proceedings of the International Conference on Information Processing*, Paris, pp. 125–132. UNESCO, 1959.

2. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Commun. ACM*, 6(1):1–17, 1963. Edited by P. Naur.
3. H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. On the formal definition of a PL/I subset (selected parts). In *Programming Languages and Their Definition – Hans Bekič (1936–1982)*, Lecture Notes in Computer Science, vol. 177, pp. 107–155. Springer, Berlin, 1984 (Full version published as Technical Report 25.139, IBM Laboratory, Vienna, December 1974).
4. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. Addison-Wesley, Reading, MA, 1989.
5. D. Bjørner and O. N. Oest. *Towards a Formal Description of Ada*, Lecture Notes in Computer Science, vol. 98. Springer, Berlin, 1980.
6. F. Chalub and C. Braga. Maude MSOS tool. In *WRLA 2006*, ENTCS, vol. 176.4, pp. 133–146. Elsevier, Amsterdam, 2007.
7. N. Chomsky. On certain formal properties of grammars. *Inform. Control*, 2:137–167, 1959.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework*, Lecture Notes in Computer Science, vol. 4350. Springer, Berlin, 2007.
9. K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Programming*, 47(1):3–36, 2003.
10. V. Donzeau-Gouge, G. Kahn, and B. Lang. On the formal definition of Ada. In *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science, vol. 94, pp. 475–489. Springer, Berlin, 1980.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*, Third Edition. Prentice-Hall, Englewood Cliffs, NJ, 2005.
12. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: Reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.
13. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Not.*, 35(3):39–48, 2000.
14. J. Iversen. An action compiler targeting Standard ML. In *LDTA 2005*, ENTCS, vol. 141.4, pp. 167–188. Elsevier, Amsterdam, 2005.
15. J. Iversen and P. D. Mosses. Constructive action semantics for Core ML. *IEE Proc.-Software*, 152:79–98, 2005 (Special issue on Language Definitions and Tool Generation).
16. C. B. Jones. More on exception mechanisms. In *Formal Specification and Software Development*, chap. 5, pp. 125–140. Prentice-Hall, Englewood Cliffs, NJ, 1982.
17. C. B. Jones. The transition from VDL to VDM. *J. Universal Comput. Sci.*, 7(8):631–640, 2001.
18. G. Kahn. Natural semantics. In *STACS 87*, Lecture Notes in Computer Science, vol. 247, pp. 22–39. Springer, Berlin, 1987.
19. S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of action notation. In *AS 2000, 3rd International Workshop on Action Semantics*, Recife, Brazil, Proceedings, number 00-6, BRICS NS, pp. 19–36. Department of Computer Science, University of Aarhus, 2000.
20. J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, North-Holland, Amsterdam, pp. 21–28, 1962.

21. R. E. Milne. The mathematical semantics of Algol 68. Unpublished manuscript, Programming Research Group, University of Oxford, 1972.
22. R. Milner, M. Tofte, R. Harper, and D. Macqueen. *The Definition of Standard ML – Revised*. MIT Press, Cambridge, MA, 1997.
23. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1989.
24. P. D. Mosses. The mathematical semantics of Algol60. Technical Monograph PRG-12, Oxford University Computing Laboratory, 1974.
25. P. D. Mosses. *Mathematical Semantics and Compiler Generation*. D.Phil. dissertation, University of Oxford, 1975.
26. P. D. Mosses. Compiler generation using denotational semantics. In *MFCS 76*, Lecture Notes in Computer Science, vol. 45, pp. 436–441. Springer, Berlin, 1976.
27. P. D. Mosses. Making denotational semantics less concrete. In *Proceedings of International Workshop on Semantics of Programming Languages*, Bad Honnef, Bericht 41, pp. 102–109. Abteilung Informatik, Universität Dortmund, 1977.
28. P. D. Mosses. SIS, Semantics Implementation System: Reference manual and user guide. DAIMI MD 30, Department of Computer Science, University of Aarhus, 1979.
29. P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, vol. B, chap. 11. Elsevier, Amsterdam/MIT Press, Cambridge, MA, 1990.
30. P. D. Mosses. Action semantics. In *Cambridge Tracts in Theoretical Computer Science*, vol. 26. Cambridge University Press, Cambridge, MA, 1992.
31. P. D. Mosses. Exploiting labels in structural operational semantics. *Fund. Informaticae*, 60:17–31, 2004.
32. P. D. Mosses. Modular structural operational semantics. *J. Logic Algebr. Program.*, 60–61:195–228, 2004 (Special issue on SOS).
33. P. D. Mosses. A constructive approach to language definition. *J. Universal Comput. Sci.*, 11(7):1117–1134, 2005.
34. P. D. Mosses. Formal semantics of programming languages: An overview. In *FoVMT 2004*, *ENTCS*, vol. 148.1, pp. 41–73. Elsevier, Amsterdam, 2006.
35. P. D. Mosses. Teaching semantics of programming languages with Modular SOS. In *Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing. British Computer Society, 2006.
36. P. D. Mosses. VDM semantics of programming languages: Combinators and monads. In *Formal Methods and Hybrid Real-Time Systems*, Lecture Notes in Computer Science, vol. 4700, pp. 483–503. Springer, Berlin, 2007.
37. P. D. Mosses. Component-based description of programming languages. In *Visions of Computer Science*, Electronic Workshops in Computing, pp. 275–286. BCS, Swindon, 2008.
38. P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proceedings of IFIP TC2 Working Conference*, Gl. Avernæs, 1986, pp. 135–166. North-Holland, Amsterdam, 1987.
39. G. D. Plotkin. A structural approach to operational semantics. *J. Logic Algebr. Program.*, 60–61:17–139, 2004 (Originally published as DAIMI FN-19, Department of Computer Science, University of Aarhus, 1981).
40. D. S. Scott. Outline of a mathematical theory of computation. In *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems*, pp. 169–176. Princeton University, 1970. Superseded by Technical Monograph PRG-2, Oxford University Computing Laboratory, November 1970.

41. D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, *Microwave Research Institute Symposia*, vol. 21, pp. 19–46. Polytechnic Institute of Brooklyn, 1971. Also Technical Monograph PRG-6, Oxford University Computing Laboratory, August 1971.
42. C. Strachey. Towards a formal semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming, Proceedings of IFIP Working Conference*, Vienna, 1964. North-Holland, Amsterdam, 1966.
43. C. Strachey. The varieties of programming language. In *Proceedings of the International Computing Symposium*, Cini Foundation, Venice, pp. 222–233, 1972 (A revised and slightly expanded version is Technical Monograph PRG-10, Oxford University Computing Laboratory, March 1973).
44. C. Strachey. Fundamental concepts in programming languages. *Higher-Order Symbolic Comput.*, 13(1–2):11–49, 2000 (Originally lecture notes, NATO Copenhagen Summer School, 1967).
45. C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order Symbolic Comput.*, 13(1–2):135–152, 2000 (Originally published as Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974).
46. M. van den Brand, J. Iversen, and P. D. Mosses. An action environment. *Sci. Comput. Program.*, 61(3):245–264, 2006.
47. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A component-based language development environment. In *CC 2001*, Lecture Notes in Computer Science, vol. 2027, pp. 365–370. Springer, Berlin, 2001.
48. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, *AMAST Series in Computing*, vol. 5. World Scientific, Singapore, 1996.
49. VSR-net: A network for the verified software repository. `http://vsr.sourceforge.net/`.
50. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pp. 157–170. The USENIX Association, Berkeley, CA, 1997.
51. D. A. Watt and M. Thomas. *Programming Language Syntax and Semantics.* Prentice-Hall, Englewood Cliffs, NJ, 1991.