Preface

1 Specification Languages

By a specification language we understand a formal system of syntax, semantics and proof rules. The syntax and semantics define a language; the proof rules a proof system. Specifications are expressions in the language — and reasoning over properties of these specifications is done within the proof system.

This book [2] will present nine of the current specification languages (ASM [40], B [5], CafeOBJ [8], CASL [33], Duration Calculus [14], RAISE (RSL) [12], TLA$^+$ [32], VDM (VDM-SL) [11] and Z [22]) and their logics of reasoning.

1.1 Specifications

Using a specification language we can formally describe a domain, some universe of discourse "out there, in reality", or we can prescribe requirements to computing systems that support activities of the domain; or we can specify designs of computing systems (i.e., machines: hardware + software).

A specification has a meaning. Meanings can be expressed in a property-oriented style, as in ASM, CafeOBJ, CASL and Duration Calculus, or can be expressed in a model-oriented style, as in B, RAISE/RSL, TLA, VDM or Z. RAISE/RSL provides a means for "slanting" a specification either way, or some "compromise" in-between. In the property-oriented style specifications emphasise properties of entities and functions. In the model-oriented style specifications emphasise mathematical values like sets, Cartesians, sequences, and maps and functions over these. (The above "compartmentalisation" is a very rough one. The nine language chapters of this book will provide more definitive delineations.)

Descriptions

Descriptions specify an area of, say, human activity, a domain, as it is, with no reference to requirements to computing systems that support activities

of the domain. Usually the domain is "loose", entities, functions, events and behaviours of the domain are not fully understood and hence need be loosely described, that is, allow for multiple interpretations. Or phenomena of the domain are non-deterministic: the value of an entity, the outcome of a function application (i.e., an action), the elaboration of an event, or the course of a behaviour is not unique: could be any one of several. We take behaviours to be sets of sequences of actions and events — or of behaviours, that is multiple, possibly intertwined behaviours.

Hence we find that some specification languages allow for expressions of looseness, underspecification, non-determinism and/or concurrency. Since phenomena of domains are usually not computable the specification language must allow for the expression of non-computable properties, values and functions.

Prescriptions

Prescriptions are also specifications, but now of computable properties, values, functions and behaviours. Prescriptions express requirements to a computing system, i.e., a machine, that is to support activities (phenomena: entities, functions, events and behaviours) of a domain. Thus prescription languages usually emphasise computability, but not necessarily efficiency of computations or of representation of entities (i.e., data).

Designs

On the basis of a requirements prescription one can develop the design of a computing system. The computing system design is likewise expressed in a specification language and specifies a machine: the hardware and software that supposedly implement the requirements and support desired activities of the domain. The machine, once implemented, resides in the (previously described) domain and constitutes with that prior domain a new domain. (Usually we think of requirements being implemented in software on given hardware. We shall, accordingly, just use the term software design where computing systems is the more general term.)

1.2 Reasoning

In describing domains, in prescribing requirements and in designing software we may need to argue that the specification possess certain not immediately obvious (i.e., not explicitly expressed) properties. And in relating requirements prescriptions to the "background" domain, and in relating software designs to the "background" requirements and domain, one may need to argue that the requirements prescription stands in a certain relation to a domain description or that the software design is correct with respect to "its" requirements under the assumptions expressed by a domain description.

For this we need resort to the proof system of the specification language — as well as to other means. We consider in this prelude three such means.

Verification

Verification, in general terms, is a wide and inclusive term covering all approaches which have the aim of establishing that a system meets certain properties. Even a simple *test case* demonstrates a, perhaps limited, fact: that in *this* case (though maybe no others) a given system achieves (or does not) a desirable outcome.

More specifically and usually, we use the term *verification* for more elaborate and systematic mathematical techniques for establishing that systems possess certain properties. Here, the *system* might be a more-or-less abstract description (a specification) or a concrete realisation in hardware or software. The *properties* may be specific emergent properties of abstract specifications; they include general statements of, say, *liveness*, *safety* and/or *termination*; and they cover the *correctness* of realisations or implementations of given system specifications. In all the cases of interest to us, the system description and the properties to be determined will be couched in a precise formal mathematical language. As a consequence, the results of such a verification will be correspondingly precise and formal.

There are three forms of formal verification that are relevant to the material covered in this book and that are, therefore, worth describing in just a little more detail.

Inferential Verification

This approach is often simply referred to as *verification* despite the fact that other approaches, such as model checking, are also such methods. Here, we have at our disposal logical principles, a logic or proof system, which correctly captures the framework within which the system is described. This framework might be a programming or specification language with a semantics which lays down, normatively, its meaning. The logical principles will (at the very least) be *sound* with respect to that semantics; thus ensuring that any conclusions drawn will be correct judgements of the language in question.

The logical principles, or fully-fledged logic, will provide means that are appropriate for reasoning about the techniques and mechanisms that are available in the language of description. For example, many frameworks provide a means for describing recursive systems, and appropriate induction principles are then available for reasoning about such systems.

Inference-based methods of verification allow us to make and support general claims about a system. These may demonstrate that an implementation is *always* guaranteed to meet its specification; that it *always* possesses certain characteristic properties (for example, that it is *deadlock-free* or maybe that it

terminates); or that an abstract specification will always possess certain implicit properties (which will, in turn, be inherited properties of *any* (correct) implementation).

Model Checking

This approach to verification (see, for example, [6]) aims to automatically establish (or provide a counterexample for) a property by direct inspection of a model of the system in question. The model may be represented (explicitly or implicitly) by a directed graph whose nodes are states and whose edges are legitimate state transitions; properties may be expressed in some form of temporal logic.

Two key issues are *finiteness* and the potential *combinatorial explosion* of the state space. Many techniques have been developed to minimise the search. In many cases it is not necessary to build the state graph but simply to represent it symbolically, for example by propositional formulae, and then, using techniques such as SAT-solvers, to mimic the graph search. Partial order reductions, which remove redundancies (in explicit graphs) arising from independent interleavings of concurrent events can also be employed to significantly reduce the size of the search space. It is also possible to simplify the system, through abstraction, and to investigate the simpler model as a surrogate for the original system. This, of course, requires that the original and abstracted systems are related (by refinement) and that the abstracted system is at least *sound* (if not *complete*) with respect to the original: that properties true of the abstracted system are also true of the original, even if the abstracted system does not capture *all* properties of the original.

Model checking has been a spectacularly successful technology by any measure; the model checker SPIN [23], for example, detected several crucial errors in the controller for a spacecraft [21]. Other important model checkers are SMV [31] and FDR, based on the standard *failures-divergencies* model of CSP [42].

Formal Testing

Dijkstra, in his ACM Turing Lecture in 1972, famously said: "*... program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence*" [9]. A correct contrast between informal testing (which might demonstrate a flaw in a system) and a formal verification (which might make a general correctness claim) was established by this remark. More recently, however, it has become clear that there is something to be gained by combining variations on the general theme of testing with formal specifications and verifications. Indeed, the failure of a *formal* test is a *counterexample*, which is as standard a mathematical result as could be wished for (and potentially as valuable too); the problem is that when testing

without a theoretical basis (informal testing), it is often simply unclear what conclusion can and should be drawn from such a methodology.

A portfolio approach, in which a variety of verification methods are used, brings benefits. In the case of *formal* testing, there is an interplay between test (creation, application and analysis) and system specification: a formal description of a system is an excellent basis for the generation (possibly automatically) of test cases which, themselves, have precise properties regarding coverage, correctness and so on. In addition, the creation of adequate test suites is expensive and time-consuming, not to say repetitious if requirements and specifications evolve; exploiting the precision implicit in formal specification to aid the creation of test suites is a major benefit of formal testing technologies.

1.3 Integration of Specification Languages

Domains, requirements or software being described, prescribed or designed, respectively, usually possess properties that cannot be suitably specified in one language only. Typically a variety, a composition, a "mix" of specification notations need be deployed. In addition to, for example, either of ASM, B, CafeOBJ, CASL, RAISE/RSL, VDM or Z, the specifier may resort to additionally using one or more (sometimes diagrammatic) notations such as Petri nets [27, 35, 37–39], message sequence charts [24–26], live sequence charts [7, 19, 28], statecharts [15–18, 20], and/or some textual notations such as temporal logics (Duration Calculus, TLA+, or LTL — for linear temporal logic [10, 29, 30, 34, 36]).

Using two or more notations, that is, two or more semantics, requires their integration: that an identifier a in one specification (expressed in one language) and "the same" identifier (a) in another specification (in another language) can be semantically related (i.e., that there is a 'satisfaction relation').

This issue of integrating formal tools and techniques is currently receiving high attention as witnessed by many papers and a series of conferences: [1, 3, 4, 13, 41]. The present book will basically not cover integration.[1]

2 Structure of Book

The book is structured as follows: In the main part, Part II, we introduce, in alphabetic order, nine chapters on ASM, event-B, CafeOBJ, CASL, DC, RAISE, TLA$^+$, VDM and Z. Each chapter is freestanding: It has its own list of references and its own pair of symbol and concept indexes. Part III introduces just one chapter, Review, in which eight "originators" of respective specification languages will comment briefly on the chapter on "that language".

[1] TLA$^+$ can be said to be an integration of a temporal logic of actions, TLA, with set-theoretical specification. The RAISE specification language has been "integrated" with both Duration Calculus and concrete timing.

3 Acknowledgements

Many different kinds of institutions and people must be gratefully acknowledged.

CoLogNET: Dines Bjørner thanks the 5th EU/IST Framework Programme (`http://www.cordis.lu/fp5/home.html`) of the European Commission, Contract Reference IST-2001-33123: CoLogNET: Network of Excellence in Computational Logic: `http://www.eurice.de/colognet` for support.

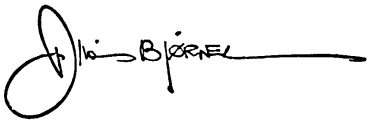
CAI: Dines Bjørner thanks the editorial board of the Slovak Academy Journal for giving us the opportunity to publish the papers mentioned on Pages 4–5.

Stara Lesna: We both thank Dr. Martin Pěnička of the Czech Technical University in Prague and Prof. Branislav Rovan and Dr. Dusan Guller of Comenius University in Bratislava, Slovakia for their support in organising the Summer School mentioned on Pages 5–6.

Book Preparation: We both thank all the contributing authors for their willingness to provide their contributions and their endurance also during the latter parts of the editing phase.

Springer: We both thank the editorial board of the EATCS Monographs in Theoretical Computer Science Series and the Springer editor, Ronan Nugent, for their support in furthering the aims of this book.

Our Universities: Last, but not least, we gratefully acknowledge our universities for providing the basis for this work: the University of Essex, UK and the Technical University of Denmark (DTU).



Martin Henson
University of Essex
Colchester, UK
April 4, 2007

Dines Bjørner
Technical University of Denmark
Kgs. Lyngby, Denmark
April 4, 2007

References

1. K. Araki, A. Galloway, K. Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.
2. Edited by D. Bjørner, M.C. Henson: *Logics of Specification Languages* (Springer, 2007)

3. E.A. Boiten, J. Derrick, G. Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, UK, April 4–7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM.
4. M.J. Butler, L. Petre, K. Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15–18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM.
5. D. Cansell, D. Méry. *The event-B Modelling Method: Concepts and Case Studies*, pages 33–138. Springer, 2007. See [2].
6. E.M. Clarke, O. Grumberg, D.A. Peled: *Model Checking* (MIT Press, 2000)
7. W. Damm, D. Harel: *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design **19** (2001) pages 45–80
8. R. Diaconescu. *A Methodological Guide to CafeOBJ Logic*, pages 139–218. Springer, 2007. See [2].
9. E.W. Dijkstra: *The Humble Programmer.* Communications of the ACM **15**, 10 (1972) pages 859–866
10. B. Dutertre: Complete Proof System for First-Order Interval Temporal Logic. In: *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science* (IEEE CS, 1995) pages 36–43
11. J.S. Fitzgerald. *The Typed Logic of Partial Functions and the Vienna Development Method*, pages 427–461. Springer, 2007. See [2].
12. C. George, A.E. Haxthausen. *The Logic of the RAISE Specification Language*, pages 325–375. Springer, 2007. See [2].
13. W. Grieskamp, T. Santen, B. Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1–3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.
14. M.R. Hansen. *Duration Calculus*, pages 277–324. Springer, 2007. See [2].
15. D. Harel: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming **8**, 3 (1987) pages 231–274
16. D. Harel: *On Visual Formalisms.* Communications of the ACM **33**, 5 (1988)
17. D. Harel, E. Gery: *Executable Object Modeling with Statecharts.* IEEE Computer **30**, 7 (1997) pages 31–42
18. D. Harel, H. Lachover, A. Naamad et al.: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems.* Software Engineering **16**, 4 (1990) pages 403–414
19. D. Harel, R. Marelly: *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine* (Springer, 2003)
20. D. Harel, A. Naamad: *The STATEMATE Semantics of Statecharts.* ACM Transactions on Software Engineering and Methodology (TOSEM) **5**, 4 (1996) pages 293–333
21. K. Havelund, M.R. Lowry, J. Penix: *Formal Analysis of a Space Craft Controller Using SPIN.* Software Engineering **27**, 8 (2001) pages 1000–9999
22. M.C. Henson, M. Deutsch, S. Reeves. *Z Logic and Its Applications*, pages 463–565. Springer, 2007. See [2].
23. G.J. Holzmann: *The SPIN Model Checker: Primer and Reference Manual* (Addison-Wesley Professional, 2003)
24. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.
25. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.
26. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.

27. 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–Verlag, Heidelberg 1985, revised and corrected second version: 1997)
28. J. Klose, H. Wittke: An Automata Based Interpretation of Live Sequence Charts. In: *TACAS 2001*, ed by T. Margaria, W. Yi (Springer-Verlag, 2001) pages 512–527
29. Z. Manna, A. Pnueli: *The Temporal Logic of Reactive Systems: Specifications* (Addison-Wesley, 1991)
30. Z. Manna, A. Pnueli: *The Temporal Logic of Reactive Systems: Safety* (Addison-Wesley, 1995)
31. K. McMillan: *Symbolic Model Checking* (Kluwer, Amsterdam 1993)
32. S. Merz. *The Specification Language TLA$^+$*, pages 377–426. Springer, 2007. See [2].
33. T. Mossakowski, A.E. Haxthausen, D. Sannella, A. Tarlecki. *CASL – The Common Algebraic Specification Language*, pages 219–276. Springer, 2007. See [2].
34. B.C. Moszkowski: *Executing Temporal Logic Programs* (Cambridge University Press, UK 1986)
35. C.A. Petri: *Kommunikation mit Automaten* (Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962)
36. A. Pnueli: The Temporal Logic of Programs. In: *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science* (IEEE CS, 1977) pp 46–57
37. W. Reisig: *Petri Nets: An Introduction*, vol 4 of *EATCS Monographs in Theoretical Computer Science* (Springer, 1985)
38. W. Reisig: *A Primer in Petri Net Design* (Springer, 1992)
39. W. Reisig: *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets* (Springer, 1998)
40. W. Reisig. *Abstract State Machines for the Classroom*, pages 1–32. Springer, 2007. See [2].
41. J.M. Romijn, G.P. Smith, J.C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM.
42. A.W. Roscoe: *The Theory and Practice of Concurrency* (Prentice Hall, 1999)

An Overview

Dines Bjørner and Martin C. Henson

[1] Department of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark (bjorner@gmail.com)
[2] Department of Computer Science, University of Essex, Wivenhoe Park, Colchester, Essex CO4 3SQ, UK (hensm@essex.ac.uk)

Before going into the topic of formal specification languages let us first survey the chain of events that led to this book as well as the notions of the specific specification languages and their logics.

1 The Book History

Four phases characterise the work that lead to this book.

1.1 CoLogNET

CoLogNET was a European (EU) Network of Excellence. It was funded by FET, the Future and Emerging Technologies arm of the EU IST Programme, FET-Open scheme. The network was dedicated to furthering computational logic as an academic discipline.

We refer to http://newsletter.colognet.org/.

One of the editors (DB) was involved in the CoLogNET effort. One of his obligations was to propagate awareness of the logics of formal specification languages.

1.2 CAI: Computing and Informatics

One of the editors of this book (DB) was also, for many years, an editor of CAI, the Slovak Academy journal on Computing and Informatics (http://www.cai.sk/). The chief editors kindly asked DB to edit a special issue. It was therefore quite reasonable to select the topic of the logics of formal (methods') specification languages and to invite a number of people to author papers for the CAI.

The result was a double issue of CAI:

CAI, Volume 22, 2003, No. 3

⋆ **The Expressive Power of Abstract State Machines**
W. Reisig [7]
Abstract: Conventional computation models assume symbolic representations of states and actions. Gurevich's "Abstract State Machine" model takes a more liberal position: Any mathematical structure may serve as a state. This results in "a computational model that is more powerful and more universal than standard computation models".
We characterize the Abstract State Machine model as a special class of transition systems that widely extends the class of "computable" transition systems. This characterization is based on a fundamental Theorem of Y. Gurevich.

⋆ **Foundations of the B Method**
D. Cansell, D. Méry [1]
Abstract: B is a method for specifying, designing and coding software systems. It is based on Zermelo–Fraenkel set theory with the axiom of choice, the concept of generalized substitution and on structuring mechanisms (machine, refinement, implementation). The concept of refinement is the key notion for developing B models of (software) systems in an incremental way. B models are accompanied by mathematical proofs that justify them. Proofs of B models convince the user (designer or specifier) that the (software) system is effectively correct. We provide a survey of the underlying logic of the B method and the semantic concepts related to the B method; we detail the B development process partially supported by the mechanical engine of the prover.

⋆ **CafeOBJ: Logical Foundations and Methodologies**
R. Diaconescu, K. Futatsugi, K. Ogata [2]
Abstract: CafeOBJ is an executable industrial-strength multi logic algebraic specification language which is a modern successor of OBJ and incorporates several new algebraic specification paradigms. In this paper we survey its logical foundations and present some of its methodologies.

⋆ **CASL — The Common Algebraic Specification Language: Semantics and Proof Theory**
T. Mossakowski, A.E. Haxthausen, D. Sannella, A. Tarlecki [6]
Abstract: CASL is an expressive specification language that has been designed to supersede many existing algebraic specification languages and provide a standard. CASL consists of several layers, including basic (unstructured) specifications, structured specifications and architectural specifications (the latter are used to prescribe the structure of implementations). We describe a simplified version of the CASL syntax, semantics

and proof calculus at each of these three layers and state the corresponding soundness and completeness theorems. The layers are orthogonal in the sense that the semantics of a given layer uses that of the previous layer as a "black box", and similarly for the proof calculi. In particular, this means that CASL can easily be adapted to other logical systems.

CAI, Volume 22, 2003, No. 4

⋆ **The Logic of the RAISE Specification Language**
C. George, A.E. Haxthausen [3]
Abstract: This paper describes the logic of the RAISE Specification Language, RSL. It explains the particular logic chosen for RAISE, and motivates this choice as suitable for a wide spectrum language to be used for designs as well as initial specifications, and supporting imperative and concurrent specifications as well as applicative sequential ones. It also describes the logical definition of RSL, its axiomatic semantics, as well as the proof system for carrying out proofs.

⋆ **On the Logic of TLA+**
S. Merz [5]
Abstract: TLA+ is a language intended for the high-level specification of reactive, distributed, and in particular asynchronous systems. Combining the linear-time temporal logic TLA and classical set-theory, it provides an expressive specification formalism and supports assertional verification.

⋆ **Z Logic and Its Consequences**
M.C. Henson, S. Reeves, J.P. Bowen [4]
Abstract: This paper provides an introduction to the specification language Z from a logical perspective. The possibility of presenting Z in this way is a consequence of a number of joint publications on Z logic that Henson and Reeves have co-written since 1997. We provide an informal as well as formal introduction to Z logic and show how it may be used, and extended, to investigate issues such as equational logic, the logic of preconditions, the issue of monotonicity and both operation and data refinement.

1.3 The Stara Lesna Summer School

The preparation of the many papers for the CAI lead to the desire to "crown" the achievements of the many authors by arranging the Logics of Specification Language Summer School at the Slovak Academy's conference centre in Stara Lesna, the High Tatras.

We refer to `http://cswww.essex.ac.uk/staff/hensm/sssl/`.

One of the editors of the present volume (MH) coordinated with the seven sets of authors of the CAI double issue as well as with Drs. John Fitzgerald (VDM: The Vienna Development Method) and Michael Reichhardt Hansen (DC: Duration Calculi) on the schedule of nine sets of lectures of 90 minutes each during the two-week event.

The other editor (DB) was the primary organiser of the event: soliciting funds, participants, and communicating with the local organiser Prof. Branislav Rovan at the Comenius University in Bratislava.

The event took place June 6–19, 2004 at the Slovak Academy's ideally located conference centre in Stara Lesna, the High Tatras.

Besides being substantially sponsored by the EU's CoLogNET effort, much-needed support also came from UNU-IIST, the United Nations University's International Institute for Software Technology (http://www.iist.unu.edu) (located in Macau, China) and Microsoft Research (http://research.microsoft.com/foundations/).

Forty-four young researchers from 22 countries in Asia and Europe took part in this seminal event.

1.4 Book Preparation

The success, so we immodestly claim, of the Summer School then lead to the proposal to rework the CAI papers and the Summer School lecture notes into a book. MH coordinated the first phase of this endeavour, summer 2004 to February 2006. DB then followed up and is responsible for the minute style editing, indexing, etc., and the compilation of the nine individual contributions into this volume.

2 Formal Specification Languages

Here we cull from the introductions to the chapters covering respective languages — and edit these "clips".

2.1 ASM: Abstract State Machines

ASM is a technique for describing algorithms or, more generally, discrete systems. An abstract state machine [specification] is a set of conditional assignment statements. The central and new idea of ASM is the way in which symbols occurring in the syntactic representation of a program are related to the real-world items of a state. A state of an ASM may include *any* real-world objects and functions. In particular, the ASM approach does not assume a symbolic, bit-level representation of all components of a state. ASM is "a computation model that is more powerful and more universal than standard computation models", as Yuri Gurevich, the originator of ASM, claims.

2.2 B

Classical B is a state-based method for specifying, designing and coding software systems. It is based on Zermelo–Fraenkel set theory with the axiom of choice. Sets are used for data modelling. Generalised substitutions are used to describe state modifications. The refinement calculus is used to relate models at varying levels of abstraction. There are a number of structuring mechanisms (machine, refinement, implementation) which are used in the organisation of a development.

Central to the classical B approach is the idea of a software operation which will perform according to a given specification if called within a given precondition. A more general approach in which the notion of *event* is fundamental is also covered. An event has a firing condition (a guard) as opposed to a precondition. It may fire when its guard is true.

2.3 CafeOBJ

CafeOBJ is an executable algebraic specification language. CafeOBJ incorporates several algebraic specification paradigms.

Equational specification and programming is inherited from OBJ and constitutes the basis of CafeOBJ, the other features being somehow built "on top" of it.

Behavioural specification characterises how objects (and systems) behave, not how they are implemented. This form of abstraction is used in the specification and verification of software systems since it embeds other useful paradigms such as concurrency, object-orientation, constraints, nondeterminism, etc.

Preorder algebra (abbreviated POA) specification (in CafeOBJ) is based on a simplified unlabelled version of Meseguer's rewriting logic specification framework for concurrent systems. POA gives a non-trivial extension of traditional algebraic specification towards concurrency. POA incorporates many different models of concurrency, thus giving CafeOBJ a wide range of applications.

2.4 CASL

The basic assumption underlying algebraic specification is that programs are modelled as algebraic structures that include a collection of sets of data values together with functions over those sets. This level of abstraction is commensurate with the view that the correctness of the input/output behaviour of a program takes precedence over all its other properties. Another common element is that specifications of programs consist mainly of logical axioms, usually in a logical system in which equality has a prominent role, describing the properties that the functions are required to satisfy.

Basic specifications provide the means for writing specifications in a particular institution, and provide a proof calculus for reasoning within such unstructured specifications.

The institution underlying CASL, together with its proof calculus, involves many-sorted basic specifications and subsorting.

Structured specifications express how more complex specifications are built from simpler ones.

The semantics and proof calculus is given in a way that is parameterized over the particular institution and proof calculus for basic specifications.

Architectural specifications, in contrast to structured specifications, prescribe the modular structure of the implementation, with the possibility of enforcing a separate development of composable, reusable implementation units.

Finally, libraries of specifications allow the (distributed) storage and retrieval of named specifications. Since this is rather straightforward, space considerations led to the omission of this layer of CASL in the present work.

2.5 DC: The Duration Calculi

Duration Calculus (abbreviated DC) is an interval logic. DC was introduced to express and reason about models of real-time systems. A key issue in DC is to be able to express the restriction of durations of certain undesired but unavoidable states.

By a duration calculus we shall understand a temporal logic whose concept of time is captured by **Real**, whose formula connectives include those of $\Box$ ($\Box P$: always P), $\Diamond$ ($\Diamond P$: sometimes P), $\rightarrow$ ($P \rightarrow Q$: P implies Q [Q follows logically from P]), and the chop operator, ';' ($P;Q$: first P then Q); whose state duration terms, P, include those of $\int P$ (duration of P), $o(t_1, ..., t_n)$, and ℓ; and whose formulas further include those of $\lceil\rceil$ (point duration) $\lceil P \rceil$ (almost everywhere P).

2.6 RAISE and RSL

The RAISE method is based on stepwise refinement using the invent and verify paradigm. Specifications are written in RSL. RSL is a formal, wide-spectrum specification language that encompasses and integrates different specification styles in a common conceptual framework. Hence, RSL enables the formulation of modular specifications which are algebraic or model-oriented, applicative or imperative, and sequential or concurrent.

A basic RSL specification is called a class expression and consists of declarations of types, values, variables, channels, and axioms. Specifications may also be built from other specifications by renaming declared entities, hiding declared entities, or adding more declarations. Moreover, specifications may be parameterized.

User-declared types may be introduced as abstract sort types, as known from algebraic specification. In addition RSL provides predicative subtypes,

union and short record types, as known from VDM, and variant type definitions similar to data type definitions in ML.

Functions may describe processes communicating synchronously with each other via declared channels, as in CSP.

2.7 TLA and TLA+

TLA is a variant of linear-time temporal logic; it is used to specify system behaviour. TLA+ extends TLA with data structures that are specified in (a variant of) Zermelo-Fraenkel set theory. TLA+ does not formally distinguish between specifications and properties: both are written as logical formulas, and concepts such as refinement, composition of systems, or hiding of internal state are expressed using logical connectives of implication, conjunction, and quantification.

2.8 VDM

VDM can probably be credited as being the first formal specification language (1974).

Classical VDM focuses on defining types over discrete values such as numbers, Booleans, and characters — as well as over sets, Cartesians, lists, maps (enumerable, finite domain functions), and functions (in general); and defining applicative ("functional style specification programing") and imperative ("assignment and state-based specification programing") functions over values of defined types, including pre-/post-based function specifications. Set, list and map values can be comprehended, as in ordinary discrete mathematics. Logical expressions include first-order predicate (quantified) expressions.

2.9 Z

Z could be said to be rather close in some aspects to VDM-SL. A main — syntactically — distinguishing feature is, however, the schema. Schemes are usually used in two ways: for describing the state space of a system and for describing operations which the system may perform. From that follows a schema calculus. Another difference from VDM is the logics.

3 The Logics

The nine main chapters of this book comprise a dazzling, and even possibly intimidating, range of approaches; and it will be clear that the work on which this collection is based owes a debt to many researchers, over many years, who have struggled to find appropriate concepts, together with their formalisation, suitable for the task of tackling issues in the general area of system specification.

There are two perspectives which are useful to bear in mind when reading this book in its entirety, or more likely in selecting chapters to study in depth. The first is that these are studies in *applied mathematics*; the second that these are *practical methods in computer science*.

Applied mathematics is a term with a long pedigree and it has usually been identified with applications in, for example, physics, economics and so forth. A naive separation would place topics such as algebra and formal logic in the realm of *pure mathematics*; however it is not the *content* but the *motivation* that differentiates applied from pure mathematics, and the chapters of this book illustrate many areas in which more traditionally pure topics are set to work in an applied setting. ASM, CafeOBJ and CASL are based within algebra, the latter two securely located within category theory, an almost quintessential example of purely abstract mathematics; DC and TLA+ make use of modal logic; B, VDM and Z make use of set theory and (versions of) predicate logic; RAISE draws on ideas from set theory, logic, algebra and beyond. In all these too, the underlying formal structures are drawn from traditional pure mathematics, much of it from developments during the early part of the last century in the introspective realm of metamathematics: initially introduced in order for mathematics to take a closer look at itself.

It may have come as something of a surprise to early pioneers in algebra, set theory and logic to see how such abstract topics could be usefully harnessed to an applications area; but work over the last 30 years or so has demonstrated beyond question that these have become an appropriate basis for formal computer science. These chapters are a testament to, and further stage in, that developing history.

Excellent applied mathematics, however, does not come for free: one cannot simply select existing mathematics *off the shelf* and expect it to be fit for purpose. It is necessary to combine mathematical competence with a high level of conceptual analysis and innovation. In this book there are numerous examples of mathematical developments which have been necessary in order to model what have been identified as the fundamental concepts in the applications' areas, and one might select single examples from hosts of others in the various chapters. For example:

- in ASM one notes the analysis of *states as algebras* and then program statements as transformations of algebras;
- in B one notes the central concept of *generalized substitution* and its interpretation within a calculus of *weakest preconditions*;
- in CafeOBJ ones notes the introduction of *behavioural specification* based on *coherent hidden algebra*;
- in CASL one notes the use of the *institution* of *many-and-sub-sorted algebras*;
- in DC one notes the development of *continuous-time interval temporal logic* and the introduction of the concept of *durations*;

- in RAISE one notes the development of *the logic RSL* with its treatment of undefined terms, imperative and concurrent features;
- in TLA+ one notes the integration of set-theoretic notions with a version of temporal logic which includes *action formulae* and *invariance under stuttering*;
- in VDM one notes the development of the *logic of partial functions* allowing reasoning in the presence of *undefined terms*;
- in Z one notes the analysis of *refinement* and how it is analysed with respect to the *schema calculus*.

These very few observations barely scratch the surface of the wealth of conceptual novelty present within these nine frameworks, but serve to illustrate the way in which they each introduce new conceptual zoology suitable for tackling the issues they aim to address. In doing so, they must, and do, extend the mathematical framework on which they are based, whether that be set theory, some variety of formal logic or a framework of algebraic discourse. And the corollaries of that, of course, are developments of new mathematics.

Turning now to the second key point. However sophisticated the formal treatment, these are intended to be practical methods for the specification and development of systems. The chapters each address examples and applications in various ways and to differing extent. There are, here, a wealth of case studies and examples, both of practical applications and of theoretical infrastructure, all of which shed light on the applicability and fecundity of the frameworks covered. It may be worth remembering that once a perfect tool is developed, it will certainly stand the test of time. For example, consider a *chisel*: it is an ancient and very simple tool; moreover, despite centuries of technological development elsewhere, it is still in use today, essentially unchanged, because of that simplicity and its fitness for purpose. It has, quite simply, never been bettered. It is also a sobering experience to compare the simplicity of the chisel with the complexity and beauty of the wood-carvings which are possible when the tool lies in skilled and experienced hands. This is a good analogy: we will want to show that our specification frameworks are as simple and straightforward as possible, and develop skills in using them which result in applications that are significantly more complex (at least combinatorially) than the frameworks themselves. Have we yet, as a community, achieved that? Almost certainly not – but the challenge is there, and current work is mindful of these considerations. System specification is a truly monumental topic; it is very unlikely we can ever achieve the simplicity of the chisel for our frameworks, but we aim for the contrast: that we can employ them to rigorously, securely and dependably design the large and complex systems which are increasingly required of us. And surely, in that complexity, is there also a certain beauty.

How this area of formal specification will further develop in the future is a very interesting question. One imagines *and hopes* that the readers of this very volume will be among those making significant contributions towards

answering that. Even if those future frameworks little resemble the ones presented here, we can be sure of one thing: their development will require the good taste, conceptual innovation and mathematical sophistication that we see exemplified in this volume.

References

1. Dominique Cansell and Dominique Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [2–7] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
2. Ražvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. CafeOBJ: Logical Foundations and Methodology. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [1, 3–7] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
3. Chris W. George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [1, 2, 4–7] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
4. Martin C. Henson, Steve Reeves, and Jonathan P. Bowen. Z Logic and Its Consequences. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [1–3, 5–7] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
5. Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [1–4, 6, 7] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
6. Till Mossakowski, Anne E. Haxthausen, Don Sanella, and Andrzej Tarlecki. CASL — The Common Algebraic Specification Language: Semantics and Proof Theory. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [1–5, 7] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.
7. Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003. This paper is one of a series: [1–6] appearing in a double issue of the same journal: *Logics of Specification Languages* — edited by Dines Bjørner.

http://www.springer.com/978-3-540-74106-0

Logics of Specification Languages
Bjorner, D.; Henson, M.C. (Eds.)
2008, XXII, 624 p. 69 illus., Hardcover
ISBN: 978-3-540-74106-0