Monographs in Theoretical Computer Science
An EATCS Series

For further volumes:
http://www.springer.com/series/776

Donald Sannella • Andrzej Tarlecki

Foundations of Algebraic Specification and Formal Software Development

Springer

Prof. Donald Sannella
The University of Edinburgh
School of Informatics
Informatics Forum
10 Crichton St.
Edinburgh, EH8 9AB
United Kingdom

Prof. Andrzej Tarlecki
Institute of Informatics
Faculty of Mathematics,
Informatics and Mechanics
University of Warsaw
ul. Banacha 2
02-097 Warsaw, Poland
and
Institute of Computer Science
Polish Academy of Sciences
ul. Ordona 21
01-237 Warsaw, Poland

ISSN 1431-2654
ISBN 978-3-642-17335-6 e-ISBN 978-3-642-17336-3
DOI 10.1007/978-3-642-17336-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011941495

ACM Codes: F.3, D.2

Printed on acid-free paper

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

To Monika and Teresa

Preface

As its title promises, this book provides foundations for software specification and formal software development from the perspective of work on algebraic specification. It concentrates on developing basic concepts and studying their fundamental properties rather than on demonstrating how these concepts may be used in the practice of software construction, which is a separate topic.

The foundations are built on a solid mathematical basis, using elements of universal algebra, category theory and logic. This mathematical toolbox provides a convenient language for precisely formulating the concepts involved in software specification and development. Once formally defined, these notions become subject to mathematical investigation in their own right. The interplay between mathematics and software engineering yields results that are mathematically interesting, conceptually revealing, and practically useful, as we try to show.

Some of the key questions that we address are: What is a specification? What does a specification mean? When does a software system satisfy a specification? When does a specification guarantee a property that it does not state explicitly? How does one prove this? How are specifications structured? How does the structure of specifications relate to the modular structure of software systems? When does one specification correctly refine another specification? How does one prove correctness of refinement steps? When can refinement steps be composed? What is the role of information hiding? We offer answers that are simple, elegant and general while at the same time reflecting software engineering principles.

The theory we present has its origins in work on algebraic specifications starting in the early 1970s. We depart from and go far beyond this starting point in order to overcome its limitations, retaining two prominent characteristics.

The first is the use of many-sorted algebras consisting of a collection of sets of data values together with functions over those sets, or similar structures, as models of software systems. This level of abstraction fits with the view that the correctness of the input/output behaviour of a software system takes precedence over all its other properties. Certain fundamental software engineering concepts, such as information hiding, have direct counterparts on the level of such models.

The second is the use of logical axioms, usually in a logical system in which equality has a prominent role, to describe the properties that the functions are required to satisfy. This property-oriented approach allows the use of formal systems of rules to reason about specifications and the relationship between specifications and software systems. Still, the theory we present is semantics-oriented, regarding models as representations of reality. The level of syntax and its manipulation, including axioms and formal proof rules, merely provide convenient means of dealing with properties of (classes of) such models.

Our primary source of software engineering intuition is the relatively simple world of first-order functional programming, and in particular functional programming with modules as in Standard ML. It is simpler than most other programming paradigms, it offers the most straightforward fit with the kinds of models we use, and it provides syntax ("functors") that directly supports a methodology of software development by stepwise refinement. Even though some aspects of more elaborate programming paradigms are not directly reflected, the fundamental concepts we study are universal and are relevant in such contexts as well.

This book contains five kinds of material.

The requisite mathematical underpinnings:
: Chapters 1 and 3 are devoted to the basic concepts of universal algebra and category theory, respectively. This material finds application in many different areas of theoretical computer science and these chapters may be independently used for teaching these subjects. Our aim is to provide a generally accessible summary rather than an expository introduction. We omit many standard concepts and results that are not needed for our purposes and include refinements to classical universal algebra that are required for its use in modelling software. Most of the proofs are left to the reader as exercises.

Traditional algebraic specifications:
: Chapter 2 presents the standard material that forms the basis of work on algebraic specifications. From the point of view of an algebraist, much of this would be viewed as part of universal algebra. Additionally, Section 2.7 explores some of the ways in which these basics may be modified to cope with different aspects of software systems. Again, this chapter is a summary rather than an expository introduction, and many proofs are omitted.

Elements of the theory of institutions:
: In Chapter 4 we introduce the notion of an *institution*, developed as a formalisation of the concept of a logical system. This provides a suitable basis for a general theory of formal software specification and development. Chapter 10 contains some more advanced developments in the theory of institutions.

Formal specification and development:
: Chapters 5–8 constitute the core of this book. Chapter 5 develops a theory of specification in an arbitrary institution. Special attention is paid to the issue of structure in specifications. Chapter 6 is devoted to the topic of parameterisation, both of algebras and of specifications themselves. Chapter 7 presents a theory of formal software development by stepwise refinement of specifications. Chapter 8

introduces the concept of *behavioural equivalence* and studies its role in software specification and development.

Proof methods:

Chapter 9 complements the model-theoretic picture from the previous chapters by giving the corresponding proof methods, including calculi for proving consequences of specifications and correctness of refinement steps.

The dependency between chapters and sections is more or less linear, except that Chapter 10 does not depend on Chapter 9. This dependency is not at all strict. This is particularly relevant to Chapter 3 on category theory: anyone who is familiar with the concepts of category, functor and pushout may omit this chapter, returning to it if necessary to follow some of the details of later chapters. On first reading one may safely omit the following sections, which are peripheral to the main topic of the book or contain particularly advanced or speculative material: 2.6, 2.7, 3.5 except for 3.5.1, 4.1.2, 4.4.2, 4.5, 6.3, 6.4, 6.5, 8.2.3, 8.5.3, 9.5, 9.6 and Chapter 10.

This book is self-contained, although mathematical maturity and some acquaintance with the problems of software engineering would be an advantage. In the mathematical material, we assume a very basic knowledge of set theory (set, membership, Cartesian product, function, etc. — see for instance [Hal70]), but we recall all of the set-theoretic notation we use in Section 1.1. Likewise, we assume a basic knowledge of the notation and concepts of first-order logic and proof calculi; see for instance [End72]. In the examples that directly relate to programming, we assume some acquaintance with simple concepts of functional programming. No advanced features are used and so these examples should be self-explanatory to anyone with experience using a programming language with types and recursion.

In an attempt to give a complete treatment of the topics covered without going on at much greater length, quite a few important results are relegated to exercises with the details left for the reader to fill in. Fairly detailed hints are provided in many cases, and in the subsequent text there is no dependence on details of the solutions that are not explicitly given in these hints.

This book is primarily a monograph, with researchers and advanced students as its target audience. Even though it is not intended as a textbook, we have successfully used some parts of it for teaching, as follows:

Universal algebra and category theory:

A one-semester course based on Chapters 1 and 3.

Basic algebraic specifications:

A one-semester course for undergraduates based on Chapters 1 and 2.

Advanced algebraic specifications:

An advanced course that follows on from the one above based on Chapters 4–7.

Institutions:

A graduate course with follow-up seminar on abstract model theory based on most of Chapter 4 and parts of Chapter 10.

The material in this book has roots in the work of the entire algebraic specification community. The basis for the core chapters is our own research papers, which are here expanded, unified and taken further. We attempt to indicate the origins of

the most important concepts and results, and to provide appropriate bibliographical references and pointers to further reading, in the final section of each chapter. The literature on algebraic specification and related topics is vast, and we make no claim of completeness. We apologize in advance for possible omissions and misattributions.

Acknowledgements

All of this material has been used in some form in courses at the University of Edinburgh, the University of Warsaw, and elsewhere, including summer schools and industrially oriented training courses. We are grateful to all of our students in these courses for their attention and feedback.

This book was written while we were employed by the University of Edinburgh, the University of Warsaw, and the Institute of Computer Science of the Polish Academy of Sciences. We are grateful to our colleagues there for numerous discussions and for the atmosphere and facilities which supported our work. The underlying research and travel was partly supported by grants from the British Council, the Committee for Scientific Research (Poland), the Engineering and Physical Sciences Research Council (UK), the European Commission, the Ministry of Science and Higher Education (Poland), the Scottish Informatics and Computer Science Alliance and the Wolfson Foundation.

We are grateful to the entire algebraic specification community, which has provided much intellectual stimulation and feedback over the years. We will not attempt to list the numerous members of that community who have been particularly influential on our thinking, but we give special credit to our closest collaborators on these topics, and in particular to Michel Bidoit, Till Mossakowski and Martin Wirsing. Our Ph.D. students contributed to the development of our ideas on some of the topics here, and we particularly acknowledge the contributions of David Aspinall, Tomasz Borzyszkowski, Jordi Farrés-Casals and Wiesław Pawłowski.

We are grateful for discussion and helpful comments on the material in this book and the research on which it is based. In addition to the people mentioned above, we would like to acknowledge Jiři Adámek, Jorge Adriano Branco Aires, Thorsten Altenkirch, Egidio Astesiano, Hubert Baumeister, Jan Bergstra, Pascal Bernard, Gilles Bernot, Didier Bert, Julian Bradfield, Victoria Cengarle, Maura Cerioli, Rocco De Nicola, Răzvan Diaconescu, Luis Dominguez, Hans-Dieter Ehrich, Hartmut Ehrig, John Fitzgerald, Michael Fourman, Harald Ganzinger, Marie-Claude Gaudel, Leslie Ann Goldberg, Joseph Goguen, Jo Erskine Hannay, Robert Harley, Bob Harper, Rolf Hennicker, Claudio Hermida, Piotr Hoffman, Martin Hofmann, Furio Honsell, Cliff Jones, Jan Jürjens, Shin-ya Katsumata, Ed Kazmierczak, Yoshiki Kinoshita, Spyros Komninos, Bernd Krieg-Brückner, Sławomir Lasota, Jacek Leszczyłowski, John Longley, David MacQueen, Tom Maibaum, Lambert Meertens, José Meseguer, Robin Milner, Eugenio Moggi, Bernhard Möller, Brian Monahan, Peter Mosses, Tobias Nipkow, Fernando Orejas, Marius Petria, Gordon Plotkin, Axel Poigné,

John Power, Horst Reichel, Grigore Roşu, David Rydeheard, Oliver Schoett, Lutz Schröder, Douglas Smith, Stefan Sokołowski, Thomas Streicher, Eric Wagner, Lincoln Wallen and Marek Zawadowski. We apologize for any omissions. We are grateful to Stefan Kahrs, Bartek Klin and Till Mossakowski for their thoughtful and detailed comments on a nearly final version which led to many improvements, to Mihai Codescu for helpfully checking examples for errors, to Ronan Nugent of Springer and to Springer's copyeditor.

Finally, we would like to express our very special appreciation to Rod Burstall and Andrzej Blikle, our teachers, supervisors, and friends, who introduced us to this exciting area, brought us to scientific maturity, and generously supported us in our early careers.

Edinburgh and Warsaw, *Don Sannella*
September 2011 *Andrzej Tarlecki*

Contents

Preface vii

0 Introduction 1
0.1 Modelling software systems as algebras 1
0.2 Specifications 5
0.3 Software development 8
0.4 Generality and abstraction 10
0.5 Formality 13
0.6 Outlook 14

1 Universal algebra 15
1.1 Many-sorted sets 15
1.2 Signatures and algebras 19
1.3 Homomorphisms and congruences 22
1.4 Term algebras 27
1.5 Changing signatures 32
1.5.1 Signature morphisms 33
1.5.2 Derived signature morphisms 36
1.6 Bibliographical remarks 38

2 Simple equational specifications 41
2.1 Equations 42
2.2 Flat specifications 44
2.3 Theories 49
2.4 Equational calculus 53
2.5 Initial models 57
2.6 Term rewriting 65
2.7 Fiddling with the definitions 71
2.7.1 Conditional equations 72
2.7.2 Reachable semantics 74
2.7.3 Dealing with partial functions: error algebras 78

2.7.4 Dealing with partial functions: partial algebras 83
2.7.5 Partial functions: order-sorted algebras 86
2.7.6 Other options 90
2.8 Bibliographical remarks 93

3 Category theory 97
3.1 Introducing categories 99
3.1.1 Categories 99
3.1.2 Constructing categories 105
3.1.3 Category-theoretic definitions 109
3.2 Limits and colimits 111
3.2.1 Initial and terminal objects 112
3.2.2 Products and coproducts 113
3.2.3 Equalisers and coequalisers 115
3.2.4 Pullbacks and pushouts 116
3.2.5 The general situation 119
3.3 Factorisation systems 123
3.4 Functors and natural transformations 127
3.4.1 Functors 128
3.4.2 Natural transformations 135
3.4.3 Constructing categories, revisited 139
3.5 Adjoints 144
3.5.1 Free objects 144
3.5.2 Left adjoints 145
3.5.3 Adjunctions 150
3.6 Bibliographical remarks 152

4 Working within an arbitrary logical system 155
4.1 Institutions 158
4.1.1 Examples of institutions 161
4.1.2 Constructing institutions 180
4.2 Flat specifications in an arbitrary institution 187
4.3 Constraints 194
4.4 Exact institutions 198
4.4.1 Abstract model theory 205
4.4.2 Free variables and quantification 209
4.5 Institutions with reachability structure 213
4.5.1 The method of diagrams 216
4.5.2 Abstract algebraic institutions 218
4.5.3 Liberal abstract algebraic institutions 219
4.5.4 Characterising abstract algebraic institutions that admit reachable initial models 221
4.6 Bibliographical remarks 223

5 Structured specifications 229
5.1 Specification-building operations 230
5.2 Towards specification languages 237
5.3 An example 241
5.4 A property-oriented semantics of specifications 245
5.5 The category of specifications 249
5.6 Algebraic laws for structured specifications 253
5.7 Bibliographical remarks 257

6 Parameterisation 259
6.1 Modelling generic modules 260
6.2 Specifying generic modules 270
6.3 Parameterised specifications 276
6.4 Higher-order parameterisation 280
6.5 An example 287
6.6 Bibliographical remarks 290

7 Formal program development 293
7.1 Simple implementations 294
7.2 Constructor implementations 302
7.3 Modular decomposition 309
7.4 Example 316
7.5 Bibliographical remarks 322

8 Behavioural specifications 325
8.1 Motivating example 326
8.2 Behavioural equivalence and abstraction 329
8.2.1 Behavioural equivalence 330
8.2.2 Behavioural abstraction 335
8.2.3 Weak behavioural equivalence 337
8.3 Behavioural satisfaction 340
8.3.1 Behavioural satisfaction vs. behavioural abstraction 343
8.4 Behavioural implementations 348
8.4.1 Implementing specifications up to behavioural equivalence 348
8.4.2 Stepwise development and stability 350
8.4.3 Stable and behaviourally trivial constructors 352
8.4.4 Global stability and behavioural correctness 357
8.4.5 Summary 365
8.5 To partial algebras and beyond 366
8.5.1 Behavioural specifications in **FPL** 366
8.5.2 A larger example 373
8.5.3 Behavioural specifications in an arbitrary institution 384
8.6 Bibliographical remarks 396

9 Proofs for specifications 401
9.1 Entailment systems 402
9.2 Proof in structured specifications 415
9.3 Entailment between specifications 429
9.4 Correctness of constructor implementations 438
9.5 Proof and parameterisation 443
9.6 Proving behavioural properties 453
9.6.1 Behavioural consequence 454
9.6.2 Behavioural consequence for specifications 465
9.6.3 Behavioural consequence between specifications 469
9.6.4 Correctness of behavioural implementations 473
9.6.5 A larger example, revisited 475
9.7 Bibliographical remarks 482

10 Working with multiple logical systems 485
10.1 Moving specifications between institutions 486
10.1.1 Institution semi-morphisms 487
10.1.2 Duplex institutions 491
10.1.3 Migrating specifications 493
10.2 Institution morphisms 501
10.3 The category of institutions 511
10.4 Institution comorphisms 518
10.5 Bibliographical remarks 530

Bibliography 533

Index of categories and functors 553

Index of institutions 555

Index of notation 557

Index of concepts 563

Introduction

Software is everywhere and affects nearly all aspects of daily life. Software systems range in size from tiny (for instance, embedded software in simple devices, or the solution to a student's first programming exercise) to enormous (for instance, the World Wide Web, regarded as a single distributed system). The quality of software systems is highly variable, and everybody has suffered to some extent as a consequence of imperfect software.

This book is about one approach, called *algebraic specification*, to understanding and improving certain aspects of software quality. Algebraic specification is one of a collection of so-called *formal methods* which use ideas from logic and mathematics to model, analyse, design, construct and improve software. It provides means for precisely defining the problem to be solved and for ensuring the correctness of a constructed solution. The purpose of this book is to provide mathematically well-developed foundations for various aspects of this activity.

The material presented here is sufficient to support the entirely formal development of modular software systems from specifications of their required behaviour, with proofs of correctness of individual steps in the development ensuring correctness of the composed system. Although such a strict formal approach is infeasible in practice for real software systems, it serves as a useful reference point for the evaluation of less formal means for improving quality of software.

The following sections discuss some of the basic motivations which underlie this approach to formal software specification and development.

0.1 Modelling software systems as algebras

In order to be useful for the intended purpose, a software system should satisfy a wide range of requirements. For instance, it should be:

Efficient: The system should be tolerably efficient with respect to its usage of time, memory, bandwidth and other resources.

Robust: Small changes to the system should not dramatically affect its quality.

Reliable: The system should not break down under any circumstances. Incorrect user input should be recognized and explicitly rejected, and faults in the system's environment should be dealt with in a reasonable fashion.

Secure: The system should be protected against unauthorized use. It should be possible to restore any data that is lost or corrupted by an attack, and confidential data should be protected from disclosure.

User-friendly: The system should be easy to use, even without extensive prior knowledge or experience with it.

Well documented: The system's functionality, design and implementation should all be appropriately documented.

All of these properties are very important, although in practice some of them may be sacrificed, or even unachievable in absolute terms. But above all, the system must be:

Correct: The system must exhibit the required externally visible input/output behaviour.

Of course, there are various degrees of correctness, and in practice large systems contain bugs. In spite of this grim reality, it is clear that correctness — or at least, a close approximation to correctness — is the primary goal, and this is the property on which work on formal specification and development concentrates.

Software systems are complex objects. When we are interested in input/output behaviour only, it is useful to abstract away from the concrete details of code and algorithms and model software using mathematical functions, focussing solely on the relationship between inputs and outputs.

For example, consider the following four function definitions:

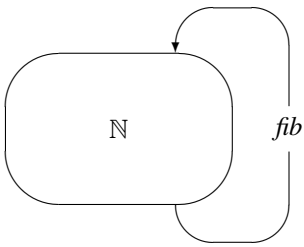
```
fun f1(n) =
  if n<=1 then 1 else f1(n-1)+f1(n-2)

fun f2(n) =
  if n<=1 then 1
  else let fun g(n) =
             if n=1 then (1,1)
             else let val (u,v) = g(n-1) in (u+v,u) end
       in let val (u,v) = g(n-1) in u+v end
       end

fun f3(n) =
  if n<=1 then 1
  else let val muv = ref (1,1,1)
       in (while let val (m,u,v) = !muv in m < n end do
             muv := let val (m,u,v) = !muv in (m+1,u+v,u) end;
          let val (m,u,v) = !muv in u end)
       end

public static nat f4(nat n) {
  nat u = 1, v = 1;
  for (nat m = 1; m < n; m++) {
```

```
      u = u + v;
      v = u - v;
    }
    return u;
  }
```

Each of these definitions of the Fibonacci function over the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ is different and has different properties. First of all, `f1`, `f2`, and `f3` are in Standard ML while `f4` is in Java; we ignore the fact that neither Standard ML nor Java has a type of natural numbers. Next, `f1` and `f2` use recursion and are purely functional while `f3` and `f4` are iterative and use assignment. The functions `f3` and `f4` actually encode the same algorithm in two different notations, an iterative version of the recursive algorithm that `f2` encodes using a local auxiliary function `g`. Also, `f1` runs in time that is exponential in n while `f2`, `f3` and `f4` require only linear time. However, the most important feature of these definitions is that they all encode the Fibonacci function $\mathit{fib}\colon \mathbb{N} \to \mathbb{N}$ defined in the usual way:

$$\begin{aligned} \mathit{fib}(0) &= 1 \\ \mathit{fib}(1) &= 1 \\ \mathit{fib}(n+2) &= \mathit{fib}(n+1) + \mathit{fib}(n) \end{aligned}$$

Before defining *fib*, it was natural to indicate that it takes elements of $\mathbb{N}$ as input and delivers elements of $\mathbb{N}$ as output. We do not really want to consider *fib* in isolation from the set of natural numbers; we view the four function definitions above as defining the function *fib* over $\mathbb{N}$, bundling data and function together:



This simple example illustrates the way in which we will model every software system as an *algebra*, that is, a set of data together with a number of functions over this set.[1] In order to deal with systems that manipulate several kinds or *sorts* of data it is necessary to use so-called *many-sorted* or *heterogeneous* algebras that contain a number of different sets of data (rather than just a single set) with functions between these sets. Functions and data types that are defined and used in a software

[1] By *software system* we mean a collection of type definitions and function definitions in a language like C or Standard ML. A software system in the sense of a traditional imperative language is a software system in this sense together with a sequence of statements (the `main` function, in C) making reference to the defined types and functions; these are not themselves made available to the user. In object-oriented languages, a software system is a collection of objects; again, this may be viewed as a software system in our sense since it essentially defines a family of types and functions, the latter capturing the objects' methods by taking the (global) state of the objects as an additional argument and returning the updated global state as an additional result.

system have names, such as `f2`, `+` and `nat` above. These names are used to refer to components of the algebra in order to compute with them, to reason about them and to build larger systems over them. The set of names associated with an algebra is called its *signature*. The formal definitions of these concepts appear in Chapter 1.

Example 0.1.1 (Timetable). A teaching timetable for a university records an assignment of lecturers to courses and courses to rooms and time slots. Such a timetable may be viewed as an algebra with the following sorts:

Course: Data elements of this sort represent courses offered by the university, e.g. Medieval History.
Lecturer: Data elements of this sort represent lecturers working in the university, e.g. Kowalski.
Timeslot: Data elements of this sort represent hours during the week, e.g. Thursday 9–10 am.
Room: Data elements of this sort represent lecture rooms in the university, e.g. JCMB 3315.

The algebra includes functions for operating on this data, for example:

who-teaches: $Course \to Lecturer$
For any course, this gives the lecturer who teaches it.
what-teaches: $Lecturer \times Timeslot \to Course$
For any lecturer and time slot, this gives the course taught by the lecturer at that time.
where-teaches: $Lecturer \times Timeslot \to Room$
For any lecturer and time slot, this gives the room where the lecturer is at that time.

We have been vague about the actual data elements; also there are more functions than we have listed (e.g. *salary*: $Lecturer \to Nat$, important for the university and the lecturers but not for the timetable).

The functions *what-teaches* and *where-teaches* unrealistically require that every lecturer teaches a course in every time slot. This problem may be resolved by adding an element *nothing* to the set *Course* and an element *nowhere* to the set *Room*, and adjusting some of the details below; see Sections 2.7.3–2.7.5 for much more on this and other options.

Extending this timetable algebra to give timetable and registration information for students as well would involve adding new sorts and new functions. The new sorts would be:

Student: Data elements of this sort represent students enrolled in the university.
Bool: The two boolean values *true* and *false*.

The new functions would include:

enrolled: $Student \times Course \to Bool$
For any student and course, this states whether or not the student is enrolled in that course.

what-attends: $Student \times Timeslot \to Course$
For any student and time slot, this gives the course attended by the student at that time.
where-attends: $Student \times Timeslot \to Room$
For any student and time slot, this gives the room where the student is supposed to be at that time.

All of these functions assume a static, fixed timetable with an unchanging assignment of lecturers and students to courses and time slots. Adding functions that change the timetable would require us to introduce a new sort *Timetable* having all possible timetables as its data elements, with functions like *enrol*: $Student \times Course \times Timetable \to Timetable$, and then the functions above would take the timetable as an additional argument. □

0.2 Specifications

Any attempt to build a software system must begin with some description of the task the system is supposed to perform. Such a description need not tightly constrain every single aspect of the required system; for example, a result is often only required up to a certain accuracy in numerical problems, the efficiency of a system is usually constrained only to fall within certain limits, and details of input/output format may often be left to the programmer. Another example is the description of the task to be performed by a compiler: the compiler should generate correct code, but the exact code to be generated is not prescribed. However loose, such a description characterises which actual software systems would be acceptable for the intended purpose and which would not be.

As discussed in the previous section, we concentrate on the functional behaviour of software systems, modelling them as algebras. Hence, a description of a class of systems amounts to a characterisation of a class of algebras. The term *specification* is used to refer to a formal object, normally in textual form, that defines such a class. It is natural to expect that every specification unambiguously defines both a signature and a class of algebras over that signature, since part of the purpose of the specification is to indicate the names of the types and functions to be provided. In this indirect way, a specification describes a class of software systems that are its acceptable *realisations*. These are the systems whose functional behaviour is captured by one of the algebras in the class defined by the specification.

The standard way of describing a class of algebras is by listing the properties they are to satisfy. Such properties may be expressed as sentences in some logical system such as equational logic or first-order logic. These sentences are called *axioms*. For any given algebra and axiom, the semantics of the logical system determines whether the algebra satisfies the axiom or not. A set of axioms thus describes a class

of algebras, namely the class of all algebras that satisfy all the axioms in that set.[2] Work on algebraic specification, to which the material in this book belongs, is based on these two fundamental principles: first, software systems may be modelled as algebras; second, properties of algebras may be described using axioms. This style of specification, which is covered in Chapter 2, naturally separates the issue of describing *what* the system is to do (given by the axioms) from that of describing *how* those requirements are achieved (given by the algorithms and data structures in the system).

Example 0.2.1 (Timetable, continued). Consider the project of assigning courses to rooms and time slots in a university. This can be viewed as the task of constructing an algebra like the one described in Example 0.1.1. The prerequisites for this are complete lists of the courses, lecturers, time slots and rooms and information concerning which lecturers are able to teach which courses. The latter may be expressed using axioms such as:

$$\mathit{who\text{-}teaches}(\mathit{Calculus}) = \mathit{Smith} \vee \mathit{who\text{-}teaches}(\mathit{Calculus}) = \mathit{Kowalski}$$

The main problem is to make sure that the assignment is done in such a way that no conflicts arise. The signature given in Example 0.1.1 guarantees that lecturers are not required to be in two places at once, since *what-teaches* and *where-teaches* are *functions*, i.e. they map any tuple of inputs (lecturer and time slot, in this case) to exactly one output (a course or a room respectively). However, we must ensure that:

1. No room is simultaneously occupied by two different courses.
2. Lecturers are sent to the courses they are assigned to teach.
3. All courses have time slots allocated to them.

These requirements are formally expressed by the following axioms:

1. $\forall t{:}\mathit{Timeslot}, c, c'{:}\mathit{Course} \bullet$
   $\mathit{where\text{-}teaches}(\mathit{who\text{-}teaches}(c), t) = \mathit{where\text{-}teaches}(\mathit{who\text{-}teaches}(c'), t) \Rightarrow c = c'$
2. $\forall l{:}\mathit{Lecturer}, t{:}\mathit{Timeslot} \bullet \mathit{who\text{-}teaches}(\mathit{what\text{-}teaches}(l, t)) = l$
3. $\forall c{:}\mathit{Course} \bullet \exists t{:}\mathit{Timeslot} \bullet \mathit{what\text{-}teaches}(\mathit{who\text{-}teaches}(c), t) = c$

Extending the problem to include timetable and registration information for students as well involves adding new axioms expressing the consistency between lecturers' and students' schedules. That is:

4. Students are sent to the courses in which they are enrolled:
   $\forall s{:}\mathit{Student}, t{:}\mathit{Timeslot} \bullet \mathit{enrolled}(s, \mathit{what\text{-}attends}(s, t)) = \mathit{true}$
5. Students are sent to each course they are enrolled for, each time it is taught:
   $\forall s{:}\mathit{Student}, l{:}\mathit{Lecturer}, t{:}\mathit{Timeslot} \bullet$
   $\mathit{enrolled}(s, \mathit{what\text{-}teaches}(l, t)) = \mathit{true} \Rightarrow \mathit{what\text{-}attends}(s, t) = \mathit{what\text{-}teaches}(l, t)$

[2] The use of the two distinct terms "set" and "class" has a mathematical justification, to be discussed in Section 3.1.1.1.

6. Students and lecturers are sent to the same place for the same course:
$$\forall l{:}\mathit{Lecturer}, s{:}\mathit{Student}, t{:}\mathit{Timeslot} \bullet$$
$$\mathit{where\text{-}teaches}(l,t) = \mathit{where\text{-}attends}(s,t) \Leftrightarrow$$
$$\mathit{what\text{-}teaches}(l,t) = \mathit{what\text{-}attends}(s,t)$$ □

The process of constructing such a specification is a subject in itself. Capturing desired properties in the form of axioms is sometimes difficult, as is deciding when a given set of axioms captures all of the desired properties. It follows that specifications can contain bugs, just as software systems can, and correctness of a system with respect to a specification is a matter of consistency between two independent definitions, either or both of which may contain errors. But once a set of axioms has been written down, theorem proving tools can be used to *validate* them by exploring their consequences, with unexpected consequences or lack of expected consequences triggering a revision of the axioms.

Up to now the discussion has concentrated on describing the requirements that a software system is to fulfill. A specification of this kind plays the role of a *contract* between a client (the customer) and the programmer (or programming team) responsible for building the system. On one hand, this contract records the features that the programmer has to ensure. On the other hand, it records the features of the system on which the client may rely. It is important that this contract be exhaustive in the sense that it record *all* the expected properties of the system; the programmer is not required to provide any features that are not explicitly stated in the contract, and the client should not rely on such features either. Any actual system will satisfy properties that are not mentioned in the contract. For example, some release of a compiler may happen to ensure that uninitialised variables are set to 0. But if this is not stated in the language definition (which is the compiler specification) it would be dangerous to rely on this feature since it may change in the next release of the compiler.

It is commonly accepted that large systems should be organized into modules that encapsulate logically coherent units. Such units may be modelled as algebras in exactly the same way as discussed above for complete systems. Specifications are required here as well to describe the interfaces between modules. These specifications constrain the programmer responsible for implementing each module to providing the required features in the same way as the specification of the overall system constrains the programming team as a whole. The clients here are other modules in the system, which may use data and/or functions that this module supplies.

As before, it is important that the interface specification record all the expected properties of the module. This means that programmers responsible for other modules are not allowed to take advantage of accidental features of modules on which they rely. Thus interface specifications serve two main purposes. First, they provide a means of communication between a module implementor and the outside world. At the same time, they serve to *prevent* undesirable communication by defining exactly those details on which others are allowed to depend, thus abstracting away from the internal details of the module implementation. A special form of such in-

formation hiding, supported by modern programming languages like Standard ML and Java, is *data abstraction*, where the exact representation of data is kept hidden.[3]

If this discipline is strictly adhered to, then programmers are free to change internal details of their module implementations without restriction, provided that the module interface specification is still satisfied. Another practical advantage of carefully specifying module interfaces is that these specifications provide the documentation necessary to support the reuse of modules in the construction of other systems.

The problems of scale which led to the introduction of modular structuring of large software systems affect large specifications as well. The specification of a large system involves thousands of properties that the system is required to satisfy. If these properties are simply listed one by one in the form of axioms, the specification would be completely unmanageable: it would be difficult to construct, and nearly impossible to understand and use. It is even a non-trivial task to understand the relatively short list of axioms in Example 0.2.1 and ensure that all the desired properties are included. The remedy to this problem is to structure such specifications into units of logically related properties which are then combined to build more complex specifications. In the example the list of axioms has been divided into two groups of related axioms to ease understanding. Mechanisms for structuring specifications are covered in Chapter 5.

The structure of a specification is not just a superficial feature of its presentation. It is important not only for understanding specifications but also for all aspects of their use. For example, in proving that certain additional properties are consequences of those explicitly stated in the specification, the structure of the specification may be exploited in guiding the search for a proof. Similarly, the structure of the specification of a software system may play a useful role in the way that the system is decomposed into modules.

0.3 Software development

As discussed above, and according to the traditional waterfall model of the software life cycle, the specification of a system is the starting point for its subsequent development. Once the specification of a software system is agreed on, the programmer is committed to building a system exhibiting a behaviour that conforms to that required by the specification. The usual way to proceed is to construct the system by whatever means are available, making informal reference to the specification in the process, and then verifying in some way that the result does indeed realise the specification. Other life cycle models take a different view, but in those that do not reject the need for specifications outright, the role of the specification as the definition of correct system behaviour remains, along with the need for verification. In

[3] In Standard ML, data abstraction is achieved using opaque signature ascription. The term "data abstraction" tends to be avoided in Java, but interfaces and access modifiers provide the required support.

the so-called V-model, the specification has an additional role, that of providing an abstract view of the final implemented system.

The most widespread verification method is testing, which checks that in certain selected cases the behaviour exhibited by the system satisfies the constraints imposed by the specification. Testing a system built to satisfy the timetable specification of Example 0.2.1 would involve checking whether the axioms hold for chosen values of variables. For instance, one might check that the axiom

$$\forall l{:}\mathit{Lecturer}, t{:}\mathit{Timeslot} \bullet \mathit{who\text{-}teaches}(\mathit{what\text{-}teaches}(l,t)) = l$$

holds for $l =$ Kowalski and $t =$ Thursday 9–10 am. This has the disadvantage that correctness can be ensured in this fashion only when the system operates on a fixed and finite set of data and exhaustive testing of all combinations is carried out. Model checking is one way of rapidly conducting such exhaustive testing.

An alternative to testing is to provide a formal proof that the system is correct with respect to the specification. For the timetable example this would amount to proving that the system satisfies the axioms listed above. However, after many years of work on software verification it now seems to be more or less widely accepted that full proofs of correctness will probably never be feasible for systems of realistic size. On the other hand, proofs of selected properties of critical parts of important systems *are* done by some software developers.

From a practical point of view, the main ground for pessimism is the huge gap between the high-level specification of requirements and the low-level details of the realisation, including the specific data representation and algorithms used and the coding of these in a particular programming language. The fact that transparency and readability are usually sacrificed for the sake of efficiency makes the gap even wider.

This leads to the idea that software systems should be developed from specifications in such a way that the result is guaranteed to be correct by construction. The approach we follow here is to develop a system from its specification via a series of small refinement steps, inspired by the programming discipline of stepwise refinement. Each refinement step captures a single design decision, for instance a choice between several functions that satisfy the specification, between several algorithms that implement the same function, or between several ways of efficiently representing a given data type. If each of these individual refinement steps can be proved correct then the resulting system is guaranteed to satisfy the original specification. Each of these proofs is orders of magnitude easier than a correctness proof for the resulting system since each refinement step is small. In principle it would be possible to combine all the individual correctness proofs to yield a proof of the correctness of the system with respect to the specification, but in practice this would never be necessary. Formal development of systems from specifications is covered in Chapters 7 and 8.

Even if we consider the very simple problem of developing a software system that realises the specification given in Example 0.2.1 (disregarding the extension to handle students), it is difficult and unnatural to come up with the definitions of all three functions simultaneously. One would tend to define them one after another,

perhaps starting with the decision of which lecturer will teach which course, then assigning time slots to the courses, and finally arranging rooms for courses. The definition of each of these functions constrains the choices available at subsequent steps since the axioms in the specification impose certain compatibility properties.

This methodology does not prevent us from making bad design decisions. For example, for some choices of *who-teaches* and *what-teaches* there may be no way to define *where-teaches* such that the specification is satisfied because of limitations on the number of rooms. This means that backtracking may be necessary during software development.

In the course of refining a large specification it will be necessary to decompose it into appropriately chosen smaller units. The refinement of these units may proceed separately, possibly involving further decomposition. This will result in a collection of modules that can be combined to yield a correct software system. Decomposition and refinement steps may be freely interleaved during the development process.

Once a system has been built in this fashion, the development history which includes all of the intermediate specifications (and possibly even the proofs of correctness) constitutes very complete design documentation. This facilitates later maintenance of the system. Even if the original specification is changed in the course of maintenance, it is normally possible to use this documentation to trace which parts of the system this change affects, localizing the fragment of the system that must be changed.

The rosy picture painted above neglects the fact that all stages of the software development process are arduous and error-prone. Coming up with a formal specification that accurately reflects all the vague and informal requirements of the customer is difficult; ideas for refinement steps are hard to come by and their formalisation is often a struggle as well; an advantageous decomposition of the problem is often difficult to find; and proofs of correctness are laborious. This leaves a lot of scope for the skill of designers and programmers. The scale of the formal objects involved and the need for meticulous accuracy and attention to detail make these creative tasks infeasible for humans to perform with pencil and paper. Many of these problems may be resolved through the use of computer-based tools to support the software development process. The most obvious candidates for this are mechanical theorem provers and proof checkers as well as some means of keeping track of all the different bits, how they interrelate and what remains to be done.

0.4 Generality and abstraction

The motivation for focussing on the functional behaviour of software systems and abstracting away from the concrete details of code and algorithms was discussed in Section 0.1. This led to the decision to model software systems as algebras. There are, however, many important aspects of the functional behaviour of systems that are not captured by this model, for example:

Non-termination: Systems do not always terminate on all inputs. The functional behaviour of such a system does not directly determine a (total) function.

Exceptions: Some operations fail on certain inputs, yielding an exceptional result or an error message. Although such results may be viewed as data values, they must be distinguished in some way. This is not accommodated by the standard definition of an algebra.

Input/output: Systems may interact with their environment during execution and this interaction is part of the functional behaviour of the system. The fact that input and output may be interleaved means that ordinary functions do not accurately model such systems.

Such aspects of systems, and their combinations (for example in reactive systems, which are designed to run forever and to react to input stimuli), are modelled by changing the notion of algebra (e.g. using so-called coalgebras to model reactive systems). Often only relatively minor enrichments are required. For instance partial algebras, where functions may be undefined on some arguments, can be used to model non-termination. Such adjustments are also necessary if we want to handle all of the relevant concepts that are present in programming languages, such as polymorphism, higher-order functions, lazy evaluation, imperative features, concurrency and mobility. Some of these elaborations are discussed in Section 2.7.

Moreover, each of these aspects of behaviour can be specified in different ways. This amounts to a choice between alternative logical systems for writing the axioms in specifications. As a very simple example, for specifying partial algebras using equational axioms there are two standard choices: strong equality with definedness formulae, or existential equality (see Section 2.7.4). Coalgebras can be specified using different modal logics. Even for ordinary algebras, there is a choice of whether to use purely equational axioms or full first-order or even higher-order logic, with trade-offs between expressive power and ease of reasoning.

There are at least three ways to proceed with the formation of a theory of software specification and development given these complications. The first is to start by devising a notion of algebra that accommodates all of the aspects of system behaviour and all relevant concepts of programming languages we can think of, with a logic for writing axioms that is rich enough to conveniently specify all of these aspects of behaviour in all of their combinations. Then we erect an appropriate theory on top of this basis. One problem with this approach is that the whole strategy breaks down when a new aspect of behaviour emerges or a new feature of programming languages becomes popular. For example, we would have to start again from scratch if we had not taken concurrency into account and it became necessary to add this later. Another problem is that the huge variety of features that would have to be considered would make the basic concepts of the theory very complicated indeed. This would yield an unwieldy theory in which one would be unable to see the forest for the trees.

Another possibility is to consider each target programming language separately and design a notion of algebra appropriate for modelling software systems built using just the particular features of this language, with an appropriate choice of logical notation for writing axioms. This has the obvious disadvantage that we must start

afresh for each programming language we consider, or even for different dialects of the same language.

Many aspects of the theory of system specification and development actually turn out to be independent of the particular details of the notion of algebra and the logical system used. This is illustrated by the fact that if we erect a complete theory for several different programming languages as described above, we will find ourselves repeating the same work time after time with only relatively minor modifications. Thus a third possibility is to develop a generic theory which is *parameterised* by the notion of algebra to be used and the definition of what it means for an algebra to satisfy an axiom. Given a particular choice of the notion of algebra, the theory can simply be *instantiated* to adapt it to that choice. Analogously, it is possible to parameterise the theory by the notion of axiom, which enables the use of different logical systems in writing specifications, and by the notion of signature to accommodate different type systems.

This third approach is the one adopted in this book. The theory presented is almost entirely independent of the particular aspects of functional behaviour of systems under consideration, of how these are described by axioms, and of the features of the underlying type system. This general view leads to reusable concepts and results and ultimately to reusable tools, which can be instantiated in particular situations as required. The resulting uniform framework exposes the essential concepts inherent in specification and development, and separates them from the sordid details of specific situations. The foundations required to support this theory are developed in Chapter 4 and then applied in subsequent chapters.

Working at this level of generality necessitates the use of mathematical tools that are appropriate for formulating general definitions and proving general facts. The language and concepts of *category theory* are convenient for dealing with the kind of generality involved. The basic concepts of category theory that are required are presented in Chapter 3.

Despite the advantages of generality, it is necessary to examine specific instantiations for the purposes of both presentation and motivation. Achieving understanding requires examination of concrete situations and examples, and these in turn demonstrate the need for developments in the general theory. Much of the time it will be sufficient to consider the simple situation in which systems are modelled as "standard" algebras, ignoring their inadequacy for the aspects of systems mentioned above, with axioms written using equations and sometimes propositional connectives and first-order quantifiers. This is the situation that is treated in Chapters 1 and 2. Examples that are meant to appeal to the reader's programming intuition are sprinkled throughout the later chapters, using an instantiation of the emerging theory to a context that is akin to a purely functional first-order subset of the Standard ML programming language, based on definitions and notations in Examples 4.1.25 and 6.1.9 and Exercise 7.3.5.

0.5 Formality

Algebraic specification as it is presented here is close to the "hard-core", uncompromising end of the spectrum of existing work on formal methods. Indeed, one of its advantages over competing approaches is that it has complete mathematical foundations. Thus, a claim of correctness of a software system or component with respect to a precise algebraic specification of the problem, when backed with all the relevant formal proofs, amounts to a complete justification without reliance on informal reasoning, guesswork or crossed fingers.

Of course, software is almost never developed in this way in practice. One reason is that 100% confidence in correctness is hardly ever necessary, and achieving it involves an enormous amount of hard work. Furthermore, experience suggests that failure of a proof of correctness is often the result of an error in the specification itself. Achieving confidence that the original specification of a problem spells out what is actually required will generally involve human interaction and other processes that are necessarily informal and error-prone.

In practice, shortcuts are normal and formal proofs are rarely attempted. Even when they are attempted, proofs are often sketched informally to a level of detail that is sufficient to check the main points of importance rather than done in full detail to completion using a proof assistant. Such a mode of use of formal methods is referred to as *rigorous methods*. Sometimes certain critical components of a system, or certain important properties, will be selected for special attention. Such a component might be one containing a complicated and important algorithm, or one that protects the system from catastrophic failure. An important property might be exception freedom or freedom from deadlock, or a security property. The degree of confidence that is justified in the outcome depends on an appropriate choice of the components and/or properties of greatest importance, and the care that is taken with informal or incomplete proofs. The above points notwithstanding, the power and sophistication of automated theorem proving tools and the computing power available to engineers have increased over time to the point where it is becoming feasible to formally verify whole systems or components of systems, and such proofs are increasingly being done in practice, especially for hardware.

From this point of view, the material in this book may be seen as providing a reference point for less formal means of improving quality of software, including rigorous methods. Another approach that puts major emphasis on the trade-off between practical benefits achieved and effort required is called *lightweight formal methods*. Here, some of the techniques of formal methods are used to improve the quality of software via early detection and removal of errors, without any expectation that they can be entirely eliminated. Lightweight formal methods rely on the use of automated analysis tools to provide cost-effective programming support.

In this book we will not explicitly point out opportunities for relaxing formality, and to a large extent that is a matter of engineering judgement in particular circumstances. Neither will we discuss to what extent the material presented provides opportunities for the provision of useful automated analyses. In general we do not provide algorithms or present decidability or complexity results for the decision

problems discussed. There is a clear trade-off between expressibility of notations on the one hand and ease of automation on the other; our approach here is firmly on the side of expressibility, with compromises in favour of automation left as a separate (but practically important) issue.

0.6 Outlook

In the previous sections we have outlined the motivations that underlie the algebraic approach to specification and formal software development. We have also discussed the need for a general approach that abstracts away from specific aspects of software systems.

This book presents mathematical foundations for algebraic specification and software development that support these practically motivated ideas. It concentrates on developing basic concepts and studying their fundamental properties rather than on demonstrating how these concepts may be used in the practice of software construction, which is a separate topic. This provides the necessary foundation for further work towards practical software production, on at least the following levels:

- More user-oriented notations and theories could be developed on top of the rudiments presented here. For example, high-level user-friendly specification languages could be defined, based on the primitive operations presented in Chapter 5.
- A computationally tractable and practically useful subset of the notations and concepts presented here could be selected and used in the style of lightweight formal methods as discussed above.
- Tools, techniques, hints and heuristics could be developed to support and guide the user's specification and development activity.

All of these aspects are beyond the scope of this book, and we deliberately avoid dealing with problems arising at these levels here. In particular we do not present "how-to" guidelines for building specifications or for validating them against real-life requirements, or for coming up with design decisions in software development. Some of the more substantial examples provide some hints in this direction.

The CASL specification language ([BM04], [Mos04]) is an attempt at a user-friendly specification notation, underpinned by many of the ideas presented here, with methodological guidelines for use of the features it provides. The material in this book and languages like CASL are not the end of the story, and they do not by any means solve all of the problems encountered in engineering practical software systems. But they do provide a solid basis for coming to grips with some of the key technical problems in software development.