Introduction

Bernhard Möller[1], *Helmut A. Partsch*[2], *Stephen A. Schuman*[3]

[1] Institut für Mathematik, Universität Augsburg, D-86135 Augsburg, Germany
[2] Fakultät für Informatik, Universität Ulm, D-89069 Ulm, Germany
[3] Dept. of Mathematics, University of Surrey, Guildford, Surrey GU2 5XH, U.K.

1 The Topics

The book attempts to survey the area of *Formal Program Development.* The most important subthemes of this area are

- formal specifications (as starting points for subsequent calculations),
- particular calculi and their theoretical foundations (inclusive of development calculi),
- rules and strategies (contents of calculi) used in such calculations, and
- systems to support these formal calculations.

1.1 Formal Specifications

A *formal specification* is intended to give a precise description of a problem to be solved by some piece of software. Research on formal specification deals mainly with appropriate concepts and the corresponding language constructs, including their theoretical foundations. These aspects deal with the use of specification constructs in describing concrete problems, with the construction of specifications in a systematic way. Additional aspects of this research are concerned with methodological issues such as acquisition, evolution, development and validation of formal specifications. Today, there is no commonly agreed general-purpose specification formalism. Rather, there is a spectrum of specific techniques based on different theoretical foundations and aimed at describing particular classes of problems (or systems) in the most appropriate way. In this book, formal specifications are explicitly addressed in the contributions by PARTSCH, PEPPER and FEATHER. In the algebraic approach, as dealt with in the papers by PARTSCH and PEPPER, a software system is specified in terms of its components through a hierarchically related structure of *algebraic types.* In each of these types, objects, object classes and operations are defined through algebraic axioms. These axioms describe the essential properties of the individual operations without refering to their operational realization. Propositions on the overall behaviour may then be inferred from the individual axioms, the interactions of the various operations and the hierarchical relationships between the component types.

In the behaviour-oriented approach, as discussed in FEATHER's contribution, a system is specified in terms of its possible events and its reactions to those events. Static

characteristics of the system concerned are captured in the notion of state, and sequences of states (or state-changes) describe the system's dynamic behaviour. Particular aspects within this overall framework are constraints on states and their maintenance during state-changes, inference mechanisms to extract information on prior and future states, as well as demons which cause certain changes of state.

The issue of formal specification is also implicitly addressed by some of the other contributions focussed mainly on calculi. Here too, different formalisms are used, partly due to the specific semantic framework, but also influenced by the class of problems considered. PETTOROSSI/PROIETTI's and BOYLE/HARMER's derivations start from functional specifications (ML and pure LISP with data abstraction, respectively), as do those by BIRD/DE MOOR and SWIERSTRA/DE MOOR. SMITH's specifications include set theoretic data types, notations from first-order logic, as well as specifications by pre/post conditions. MORGAN and PETTOROSSI/PROIETTI use logical specifications (in Z and Prolog, respectively). MÖLLER also deals with set-theoretic specifications.

1.2 Calculi

Calculi are the main focus of WG 2.1's current activities. Therefore, various calculi and their theoretical foundations are comprehensively dealt with in this book. Common to all these calculi is the intent to provide a kind of *mathematics of program construction.* Differences arise mainly in their underlying theoretical basis and, consequently, in the concrete rules of the particular calculus.

As to the theoretical foundations, in all approaches theories are imported from mathematics. Predicate logic and implicational reasoning form the basis of MORGAN's refinement calculus. PEPPER's calculus also uses first-order logic, but in the form of Gentzen style rules of natural deduction. SMITH's approach to algorithm design is essentially based on the idea of translating one theory into another. Set theory with relational theory and formal language theory as subtopics is the basis of MÖLLER's calculus. Relational theories are also used in the calculi of BACKHOUSE/HOOGENDIJK and BIRD/DE MOOR. In addition, they adapt notions from category theory, which is a source of ideas for SINTZOFF's general development calculus as well.

The practically important part of each calculus lies in the formulation and use of its associated rules and strategies. The rules and strategies discussed by PEPPER, PETTOROSSI/PROIETTI and SWIERSTRA/DE MOOR are general in the sense that they are largely language-independent and applicable to a variety of problems – although particular languages and problem domains are used in their presentation. BOYLE/HARMER's rules also are applicable to a variety of problems, but they are mainly syntactical and, as such, language-specific. In contrast to this, BIRD/DE MOOR, MÖLLER and SMITH present problem-oriented rules, i.e., language-independent rules targeted towards particular classes of problems.

1.3 Systems

There are many experimental systems which support formal program development. Likewise, a number of aspects amenable to automation are addressed by such systems. In the context of this book, emphasis is given to those aspects related to the central activity,

viz. calculation of programs; the relevant literature should be consulted for information on support for other important aspects of software development. From the wide range of possible approaches, two extreme positions w.r.t. the way programs are calculated are represented here. BOYLE/HARMER's system is essentially driven by syntactic rewriting, according to built-in strategies and without user interaction. In SMITH's system the user is offered a choice between various built-in strategies for algorithm design, from which the system then automatically derives the requested program using its knowledge-base and inference mechanism.

2 Additional Interests of WG 2.1

In addition to the topics explicitly addressed by the contributions in this book, there are other areas represented within WG 2.1 but only marginally reflected here. Nevertheless, it seems worthwhile to at least mention them, in order to give an impression of the breadth of the group's interests. Moreover, many of these latter topics are subjects of ongoing research, building on the results summarized in this book, rather than being state of the art.

In addition to the semantic issues explicitly addressed in the context of calculi, theoretical aspects of languages such as type theory, non-determinism, parallelism, concurrency and distributed systems are discussed within WG 2.1, as are many more pragmatic issues of language design.

With respect to methodology, reuse and adaptation of designs or developments and transformation towards parallel execution are fairly recent research topics. On the borderline between theory and methodology, the incorporation of efficiency considerations and the formalization of reasoning must also be mentioned.

The issue of system support is considered within the Working Group in a much wider sense, covering nearly all aspects related to automating the production of software. Such interests include optimizing compilers, integrated development environments, end-user interfaces, all kinds of language-oriented tools as well as their underlying databases or knowledge-bases.

Although this book gives a fairly comprehensive account of the state of the art in formal program development, not all subjects could be treated in depth (owing to its mainly tutorial objectives). However, further information may be obtained by pursuing the references provided here. In particular, the reader is referred to reports of ongoing research in this area which are contained in the proceedings of the IFIP Working Conferences organized by WG 2.1.

3 Summaries of the Contributions in this Book

BACKHOUSE/HOOGENDIJK's paper introduces an *algebra of data types* oriented towards the calculation of *polymorphic functions and relations.* Their approach is similar to theories of types in a functional setting, but differs in including *non-determinism.* Moreover, it achieves a uniform treatment of data and control structures. The major goal of the paper is to construct a framework in which a large class of type manipulation problems can be reduced to straightforward calculation. Economical notation and elegant programming

laws are used to express powerful fundamental concepts. Particular emphasis is laid on comparing and contrasting the calculus with the *Bird-Meertens formalism* as used in the contribution by SWIERSTRA/DE MOOR.

BIRD/DE MOOR introduce a calculus based on a categorical setting and involving relational concepts and the theory of free inductive data types. They state and prove a general theorem about problems treatable by *greedy algorithms*. The use of the calculus and the theorem are demonstrated with the *minimum lateness problem*, a job-scheduling application. BIRD/DE MOOR view greedy algorithms as refinements of *dynamic programming*. This latter paradigm is applicable if the principle of optimality applies (i.e., an optimal solution to a problem can be composed of optimal solutions to subproblems if a certain monotonicity condition holds). Whereas dynamic programming decomposes the input in all possible ways, a greedy algorithm considers only one (usually unbalanced) decomposition and reduces the input in each step as much as possible. BIRD/DE MOOR's work has an obvious relationship with SMITH's studies on divide and conquer algorithms.

BOYLE/HARMER's paper deals with the use of automated program transformations as realized in the *TAMPR system*. The purpose of the TAMPR transformations discussed in this paper is the derivation of efficient *vectorizable programs* from functional specifications in pure LISP enhanced by a data abstraction mechanism. TAMPR is built on Chomsky's idea of a transformational grammar; the laws of the underlying program algebra are expressed as rewrite rules. TAMPR also tackles some general problems that are inherent to functional languages, e.g., the speed and storage costs of higher-order functions and the lack of selective updating. It has been used to derive a large number of realistic programs, several of which are in everyday use. The particular application area considered in BOYLE/HARMER's contribution is *numerical solutions* to a practical fluid dynamics problem, stated in terms of hyperbolic partial differential equations. The results obtained are remarkable. For instance, the derived program given in the paper runs faster on a CRAY X-MP vector supercomputer than its hand-coded counterpart.

FEATHER gives a guided tour of the language *Gist*, a representative of formalisms for behaviour-oriented specification. This kind of specification is based on the notions of state and state-changes, which is motivated by the observation that it is difficult to express *ongoing behaviours* in a functional setting. Such approaches address the need to describe the *interactions* of a system with its environment, and to express complex behavioural requirements. Apart from considering the advantages of constructing a formal specification in general, FEATHER deals mainly with constructs appropriate for describing such properties. Among others, the following specification constructs are identified as useful for these purposes: sequences of states (to denote behaviours), access to information from prior and future states, *nondeterminism*, *constraints* and *demons*. In addition, operations on such specifications are discussed. Such operations are construction and maintenance (modification and re-use) of the specification, presentation and analysis (paraphrasing, symbolic evaluation, prototyping) and the transition to an implementation. The discussions are illustrated by a package router (i.e., a mechanism to sort postal packages into one of several bins according to their destinations), and an elevator system (for bringing passengers to their destinations in a multi-story building). The paper is rounded off by a brief comparison with related approaches.

MÖLLER introduces some operators and laws of an algebra of *formal languages*, a subalgebra of which corresponds to the algebra of *multiary relations*. Central operations

are join and composition, in particular in their iterated forms where they describe sets of paths and reachability. The common algebraic structure of these iterations is that of a *Kleene algebra*, which allows the statement of a powerful uniform induction principle. Using this structure, a number of lemmas on paths in subgraphs are proved in a very concise way. The algebra is then used in the formal specification and derivation of some *graph and pointer algorithms*. The examples treated are cycle detection and reachability in graphs, in-situ concatenation and reversal of singly-linked lists, a copying algorithm for general pointer structures and parts of a garbage collection problem.

MORGAN gives an introduction to his refinement calculus, a method of deriving *imperative programs* and presenting their developments. While based on Dijkstra's calculus of weakest preconditions, MORGAN's approach has a number of novel characteristics. In contrast with Dijkstra's original approach, weakest preconditions do not appear explicitly in program developments. Moreover, specifications (which may contain predicates) and programs are not distinguished semantically, and may therefore be mixed. For the derivations, use is made of an extensible collection of *refinement laws* derived from a basic refinement relation between programs. Particular emphasis is laid on *literate programming* when constructing program developments using the refinement calculus starting from specifications in Z. In addition to simple examples (e.g., swap of two variables), a complete derivation of a square root program is presented.

PARTSCH introduces the theoretical foundations and the major concepts of an algebraically based formalism for *problem specification* (using essentially the one developed within the CIP project as a representative). This approach is based on the notion of an *algebraic type*, which defines objects, object classes, and operations on these object classes by means of algebraic axioms. Particular emphasis is laid on the use of such a formalism for the specification of concrete problems, including the *methodological aspects* of formalization. Many standard examples (sets, sequences, bags, maps) are covered. In addition, more comprehensive examples are treated such as various formalizations of finite directed graphs (mainly to illustrate the formalization process), a bounded buffer (as part of a simple communication system) and the *cube problem* (a not too sophisticated puzzle that shows many pitfalls of formalization).

PEPPER's contribution models the programming activity as a deduction in a formal *calculus for program development* based on concepts from algebra and logic. In this framework, programming is viewed as a process that successively extends the program under consideration by adding new axioms or theorems to it. In this setting, axioms constitute design decisions, whereas theorems make deducible knowledge explicit. Technically, the power of the approach is achieved by combining concepts from two related areas: algebraic specifications are used to represent programs, and *Gentzen style rules* of natural deduction are used to represent derivation processes. The paper presents the *algebraic framework* and a collection of characteristic derivation rules illustrated by various examples such as binary logarithm, fast integer division and majority voting.

PETTOROSSI/PROIETTI give an overview of traditional transformational methods, focusing on the *rules and strategies* approach (as opposed to the schematic or dictionary approach). Their contribution deals with the transformation of *functional programs*, formulated in a variant of ML, and using basic rules such as definition, unfold, fold, together with laws of the underlying data algebra. Various strategies are surveyed, such as elimination of composition (to avoid intermediate data structures), tupling (to avoid repeated

visits of data structures, thus enabling on-the-fly garbage collection), generalization and a particular strategy for *online programs*. The topic of transforming *logic programs* is also covered. In this context, strategies similar to the functional case are given. The treatment is illustrated by a large collection of examples, such as evensum, Towers of Hanoi, factorial, collecting tree leaves, Fibonacci numbers, palindrome recognition, Hilbert curves, common sublists, minimal leaf replacement and prime numbers.

SINTZOFF's contribution on typing endomorphisms proposes the kernel of a *meta-calculus* for formal program derivations. This kernel is described in terms of an intuitive semantics and of corresponding algebraic concepts inspired by the theory of cartesian closed categories. It defines operations for composing deductions in such a way that typing is defined as an endomorphism on deductions. The approach is compared with others based on typed λ-calculi and is related to the language DEVA.

SMITH's contribution is composed of two parts, one on the *automation of program construction* (as implemented in the KIDS system) and another one on a *general theory of algorithm design*. The *KIDS system* provides knowledge-based support for the derivation of correct and efficient programs from specifications. The specification language includes set theoretic data types, notations from first-order logic and extensions that support specifications by pre/post conditions. The KIDS system has components for performing algorithm design, deductive inference, program simplification, partial evaluation, finite differencing optimizations, data type refinement and case analysis. All of the KIDS operations are automatic except the algorithm design tactics, which at present require some user interaction. The use of KIDS is traced in deriving a scheduling algorithm as a representative for the many programs that have been derived using that environment. This derivation illustrates various aspects such as design, deductive inference, simplification, finite differencing, partial evaluation, data type refinement and other techniques. The second part discusses the theory of algorithm design used in KIDS. Important concepts are problem theories, algorithm theories, program schemes as parameterized theories, design as interpretation between theories (theory morphisms), algorithm design tactics and refinement hierarchies of algorithm theories.

SWIERSTRA/DE MOOR demonstrate a number of techniques that may be used in calculating algorithms for sequence-oriented problems, using the *Bird-Meertens formalism*. Their central theme is the use of *virtual data structures*, which allow optimization by reasoning at the level of function compositions and then eliminating intermediate data structures at the final transformation step. These ideas are illustrated by two segment problems on lists, viz. the maximum segment sum and the length of a longest low segment.