Matthias Weber Martin Simons
Christine Lafontaine

The Generic Development Language Deva

Presentation and Case Studies

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Authors

Martin Simons
Matthias Weber
Institut für angewandte Informatik, Technische Universität Berlin
Franklinstraße 28-29, D-10587 Berlin, Germany

Christine Lafontaine
Unité d'Informatique, Université Catholique de Louvain Place
Sainte-Barbe 2, 1348 Louvain-La-Neuve, Belgium

CR Subject Classification (1991): D.3, F.3-4, D.1.1

ISBN 3-540-57335-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57335-6 Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Typesetting: Camera-ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Foreword

The Deva endeavor is almost ten years old: the requirements for the Esprit project ToolUse (1985-1990) were discussed in late 1984. The present preface offers a nice opportunity to consider the enterprise in the light of experience. The overall problem tackled in the design of Deva can be outlined as follows; the word *proofs* denotes deductions going from hypotheses to theses or from specifications to programs. On the one hand, we are used to writing and reading *human proofs*; these are sometimes unstructured, imprecise, and even incorrect. On the other hand, some of us strive to write *formal proofs*; such proofs are often too detailed and hard to understand. It is tempting to bridge the gap between human and formal proofs by introducing *formal human proofs*, or human formal proofs if one prefers, so as to remove the shortcomings of the two modes of expression without losing their respective qualities.

This task is not as hopeless as it seems. Indeed, there is a permanent tendency to improve the style of human proofs. Sloppiness, for instance, is combatted by systematic use of consecutive formal expressions separated by careful descriptions of the laws used at each step. Composition is enhanced by nesting proofs: sub-proofs correspond to sub-blocks or lemmas, and hypotheses to declarations. Such improved human proofs could be termed *enlightening proofs*. In these, the initial laws, the consecutive propositions, and the overall structure are all formalized; only the proof steps and the scope rules remain informal. The discipline fostered by such enlightening proofs reduces the temptation to cheat in reasoning; this has been a sobering personal experience. Other efforts towards formal human proofs aim at making formal proofs more human. The corresponding techniques are effective, if not original: they include systematic composition, readable notations, and automatic sub-deductions such as pattern-matching. In spite of these varied efforts from both sides, the gap between human proofs and formal ones remains a wide one. The Deva enterprise was intended to reduce it further by humanizing formal proofs a bit more.

Deva is essentially a typed functional language. The primitive functions express proof steps. Each such step is typed by an input and an output type; this pair of types expresses the propositions connected by the step, and thus amounts to a deduction rule. This view of functions as proofs and of types as propositions has been known in logic since the sixties. It differs, however, from related approaches. Indeed, a fruitful principle in computing is to consider types as abstract values and type elaboration as an abstract computation. A straightforward consequence is to view type expressions as abstract function expressions: the syntax remains the same while the interpretation changes homomorphically. This identification of type terms and function terms, first formalized in λ-typed λ-calculi, has been applied in Deva. Moreover, since the latter is essentially a programming language, its design, implementation, and use benefit from well-established methods: classical composition operators are introduced, operational semantics serves as a formal definition, implementation techniques are available, and teaching material as well as support tools follow standard principles. The difference with ordinary languages is, of course, the application domain: the types

serve here to express propositions such as specifications or programs, rather than just data classes.

Model case studies played an important part in design. This has been one of the benefits of continued cooperation with good industrial partners. A primary objective was to formalize effective methods of software design. In industry, the most productive methods use successive refinements from state-based specifications; the first example in the book illustrates this approach for an application in the field of biology. A promising research direction is the derivation of efficient programs on the basis of algorithm calculi; this is presented here in another case study. Such experiments and existing models of enlightening proofs have continuously influenced the design of Deva. In consequence, its description has been significantly modified a number of times. The genericity and reusability of implementation tools helped in mastering this necessary evolution. In fact, the current version of Deva may well be adapted further.

Before formalizing a topic, we must first understand it and design a good theory for it. In the case of program derivations, this theory-building comprises three layers: there are basic theories from mathematics and computing, then theories of design methods and application domains, and finally theories for specific program derivations. A significant part of formal software development is thus concerned with classical mathematics. This appealing blend of mathematics and programming science could be termed *modern applied mathematics*. The elaboration of a theory must not be confused with its formalization. On the one hand, without an adequate theory, the formalization does more harm than good: the better a theory is, the happier its formalization. On the other hand, one should be able to take any good piece of mathematics and formalize it nicely in a proposed language for *formal enlightening proofs*. Once a design method has been given a good theory and has been formalized accordingly, it is possible to develop formal proofs of theorems about the method itself. The present book, for instance, provides a formal theory of reification, and then a formal proof of the transitivity of reification.

Various languages for formal enlightening proofs are currently being experimented with. The reprogramming of common theories in these languages appears to be counter-productive; it is reminiscient of the republication of similar material in several books or the recoding of software libraries in different programming languages. Happily, the cost of repeated formalizations can be reduced: where the languages are quite different, at least the contents of the theories can be communicated using literate formalization, as in the case of the present book; if the languages are similar, specific translators can be developed to automate recoding. The latter solution can be used in the case of successive Deva versions.

The following views may underlie further work. Firstly, to the extent that proof expressions are homomorphic to proposition expressions, we must be free to work at the level of functions or at that of types. This would allow us to express a proof step not only as the application of a function typed by a rule to a constant typed by a proposition, but also as the direct application of the rule to the proposition; this better matches the nature of enlightening proofs.

Secondly, purely linguistic aspects play a major part: compositional structures, formal beauty, and stylized notations prove crucial for successful intellectual communication. Thirdly, semantics must be understood by minds and not just by machines: to foster higher-level reasoning on proof schemes, algebraic laws are more useful than reduction rules. The formalization of enlightening proofs should add neither semantical nor syntactical difficulties: it must instead clarify the proofs even better. Fourthly, it should be possible to formalize well any component of mathematics: the scientific basis for the design of software systems tends to include any mathematics of interest in system design. Finally, it is mandatory to capitalize on existing symbolic algorithms, decision procedures, and proof schemas; ideally, these should be integrated in specific libraries so as to be understood, communicated, and applied. In a word, we must apply, in the design and use of high-level proof languages, the successful principles established for existing high-level programming languages. The correspondence between proofs and programs also results from the similarities between algebras of proofs and algebras of programs, not only from the embedding of programs within proofs.

To conclude, Deva can be seen as a tentative step towards a satisfactory language of formal enlightening proofs. The authors should be warmly thanked for presenting this scientific work to the computing community.

Michel Sintzoff

Acknowledgments

This book would not have been possible without the continuing support of several people. First of all, we wish to thank Michel Sintzoff, who is, so to speak, the father of the research reported on in this book. His ideas on formal program development were the starting-point for the development of Deva. We are grateful to him for the stimulating thoughts he shared with us, his constant supervision, and his kind and inspiring overall support. We hope that he still recognizes some of his ideas when reading the following pages. Next, we wish to thank Stefan Jähnichen for his guidance and support. His interest in turning theoretical results into practical applications and his insistence on testing Deva on non-trivial case studies greatly influenced our work. Our thanks also go to the other members of the ToolUse project, who patiently experimented with the language and who bore with us, in the early stages, during the frequent changes to the language definition. Their comments and suggestions greatly contributed to the design of Deva. We acknowledge in particular the contribution of Philippe de Groote, who invented and investigated a kernel calculus of Deva. Jacques Cazin and Pierre Michel closely followed the design of Deva and gave helpful comments and advice. Pierre-Yves Schobbens shared with us his knowledge of VDM. We also thank the UCL at Louvain, the Belgian project Leibniz (contract SPPS/RFO/IA/15) and the GMD at Karlsruhe for their financial support beyond the duration of the ToolUse project. The staff of both the UCL and the GMD were also very supportive and took great interest in our work. We wish to express our gratitude to all of these people for their encouragement and friendship. It was this that made the whole project an enjoyable and worthwhile experience.

We had an equally stimulating working environment in Berlin. The BKA group, in particular, provided inspiring insights. Martin Beyer and Thomas Santen read drafts of the book and made valuable comments. Two implementation efforts which are underway in Berlin have had a significant impact on the book. First of all, there is Devil (Deva's interactive laboratory), an interactive environment for developing Deva formalizations, which is being designed and implemented by Matthias Anlauff (aka Maffy). We used Devil to check (almost) all Deva texts contained in this book. Our very special thanks to Maffy for his tremendous implementation effort. He worked night and day incorporating new features to extend the power and usability of the system, and removed the few bugs we discovered. Secondly, there is the DVWEB system, a WEB for Deva, which is being implemented by Maya Biersack and Robert Raschke, and which we used to write the whole book. On the one hand, DVWEB enabled us to work on a single document for both Devil and TEX, and on the other hand, its macro features greatly improved the presentation of Deva formalization. Our thanks also go to Maya and Robert for this valuable tool. Furthermore, we wish to express our gratitude to Phil Bacon for polishing up the final text.

Finally, we wish to express our gratitude to the many known and unknown referees for their helpful criticism and advice. Professor Goos, in particular, helped us to state our objectives more clearly by providing a number of critical comments on the initial draft of the book.

Table of Contents

1 Introduction

The present book presents *Deva*, a language designed to express formal development of software. This language is generic in the sense that it does not dictate a fixed style of development, but instead provides mechanisms for its instantiation by various development methods. By describing in detail two extensive and quite different case studies, we document the applicability of Deva to a wide range of problems.

Over the past few years, the interest in formal methods has steadily grown. Various conferences, workshops, and seminars on this topic have been organized and even the traditional software engineering conferences have established their own formal method sessions. A journal devoted entirely to this subject, entitled "Formal Aspects of Computing", has also been started. The association "Formal Methods Europe" was founded in 1991, as a successor to "VDM Europe" to promote the use of formal methods in general by coordinating research activities, organizing conferences, etc. Formal methods can thus no longer be viewed as the exclusive reserve of theoreticians.

However, despite the fact that current research is concerned with the whole range of activities involved in the software development process, the industrial application of formal methods is mostly limited to specification. Languages such as VDM or Z are enjoying growing acceptance in industry, as is evidenced, for example, by a number of articles in the September 1990 issue of "IEEE Transactions on Software Engineering", two special issues of "The Computer Journal" (October and December 1992) on formal methods , or by ongoing efforts to standardize both languages. This success is due to the fact that a formal specification allows formal reasoning about the specified system; in other words, it enables the question as to whether some desired property is true for the system to be answered by a mathematical proof. This greatly reduces the risk of errors that would be detected at a much later stage in the development process or not at all, thus justifying the allocation of more time and resources to the specification phase. On the other hand, truly formal methods are rarely used beyond the specification phase, i.e., during actual development. Even in pure research environments, completely formal developments remain the exception.

But what exactly *is* a formal method? According to Webster's Collegiate Dictionary, a method is "a procedure or process for attaining an object", this object being, in our case, a piece of software. A method is called *formal* if it has a sound mathematical basis. This means essentially that it is based on a formal system: a formal language, with precise syntax and semantics, some theories about the underlying structures (theories of data types, theories of refinement, etc.), and a logical calculus which allows reasoning about the objects of the language.

The main activity performed during the specification phase is the modeling of the problem in terms of the language. During the development phase, it is the refinement of the specification down to efficient code using techniques such as data refinement, operation decomposition, or transformations. This process is, again, expressed in terms of the language, and the proof obligations associated

with each refinement step are carried out within the logical calculus making use of the method's theories.

The major reason why formal developments are so rare is that far more proofs are required during development than during the specification phase. In fact, the nature of the proofs called for during the specification phase is quite different from that of the development phase. During specification one usually proves properties that are desired to hold true for the specified model. This is done in order to convince oneself or the customer of the adequacy of the specification. During design, however, one is obliged to discharge all the proof obligations associated with a refinement or transformation step, so that, in the end, one can be sure that the product of the design is correct with respect to the specification. These proof obligations are, in most cases, not profound, but fairly technical. It frequently happens that a proof obligation which is obvious to the developer requires a tricky and lengthy proof. Generally speaking, the amount of work involved in discharging a particular proof obligation is quite disproportionate to the quality of the new insights gained into the product.

The burdensome requirement of proving every little detail of a proof obligation is therefore relaxed by most methods to the point where the proof obligations are stated in full without the need to prove all of them. We call methods which adhere to this paradigm *rigorous*. Although, with a rigorous development, it is once again up to the designer to decide whether he is satisfied that the result meets the specification, it is, in principle, still possible to prove the development correct. (One might object that this is similar to the situation faced when verifying a piece of code, but the crucial difference is that, during the development process, all the vital design decisions have been recorded together with the proven and unproven proof obligations.)

Here, it may be asked why it is not possible to give reasons for the correctness of a development in the same way a mathematician gives reasons for the correctness of a proof in the first place. In fact, there is no proof given in any mathematical text we know of which is formal in the literal sense. Instead, proofs are presented in an informal, descriptive style, conveying all the information (the exact amount depends on how much background knowledge is expected of the reader) necessary to construct a formal proof. However, there are a vast number of proofs to be carried out during development, and the traditional mathematical procedure of judging proofs to be correct by submitting them to the mathematical community for scrutiny is inadequate in this situation. It must also be remembered that, here, for the first time, an engineering discipline is faced with the task of producing, understanding, and managing a vast number of proofs — a task whose intellectual difficulty is not to be underestimated. Machine support is therefore needed and this calls for formality. In this sense, we call a proof *formal* if its correctness can be checked by a machine.

However, despite the fact that full formality is needed to enable a machine ultimately to check a proof, this cannot mean that one is forced to give every little detail of a proof. This would definitely prevent formal developments from ever gaining widespread acceptance. The aim should be to come as close as possible

to the informal way of reasoning (from the point of view of the designer), while at the same time remaining completely formal (from the point of view of the machine).

In this book, we present a *generic development language* called *Deva* which was designed to express formal developments. Syntax and static semantics can be checked by a machine, their correctness guaranteeing the correctness of the development. In order to ensure independence from a specific development methodology, a major concern during design of the language was to isolate those mechanisms essential for expressing developments. Accordingly, the language allows us to construct from these basic mechanisms new mechanisms specific to a particular method. In this sense, Deva may be said to be a *generic* development language, since the formal language underlying a specific formal method can be expressed in terms of Deva.

Ideal requirements for generic development languages

The above discussion yields in several ideal requirements for generic development languages which we now go on to summarize. We will subsequently show how and by what means Deva satisfies these requirements.

First of all, a development language must provide a medium for talking about specifications, developments (i.e., refinements, transformations, and proofs), and programs. A generic development language must, in addition, provide means for expressing mechanisms for the developments themselves.

Good notation is a frequently neglected aspect of languages, and yet it is one of the most important as regards usability and acceptance [6]. Good notation should be as concise as possible, but, at the same time, suggestive enough to convey its intended meaning. In the context of development languages, this means that the notation should support various different ways of reasoning and development styles, the notational overhead introduced by formality being kept as low as possible. The developments expressed in this formal notation should compare favorably in style and size with those demanded by rigorous methods. A generic development language must, in addition, provide means for defining a new notation in a flexible and unrestrictive manner.

The language must provide means for structuring formalizations. The lesson learned from programming languages is that structural mechanisms are indispensable for formalization, even on a small scale. In the context of generic development languages, this is even more important because of the wide range of different levels of discourse. Hence, such a language must provide mechanisms for structuring in-the-large and in the small. For structuring in the large, this means that the language must have some sort of modularity discipline, which includes definition of modules, parameterizations and instantiation of modules, and inheritance among modules. Experiments have shown that, for formal developments, the following mechanisms are useful for structuring in the small: serial and collateral composition of deductions, declaration of axioms, abbreviation of deductions, parameterization and instantiation of deductions.

Since proving is one of the most important activities in formal development, it must be possible to express proofs in the language. The correctness of such proofs must be checkable with respect to an underlying deduction system. Since a formal method usually comes with its own deduction system and underlying logic, a generic development language must be flexible enough to handle a variety of logics and deduction systems.

As we have argued above, the amount of detail required to enable a machine to check the correctness of a proof has a considerable influence on the usability of the formal approach to software development. Ideally, we envisage a situation where the designer gives a sketch of a proof, just as a mathematician would, and lets the machine fill in the details. But, this is not yet state-of-the-art, and so we must be a little more modest in our demands. The language should, however, go as far as possible in allowing incomplete proofs and should also incorporate some basic mechanisms for defining so-called tactics — which can be understood as means for expressing proof sketches. Functional languages such as ML have been used with considerable success to program recurring patterns of proof into tactics and to design systems supporting semi-automatic proofs based on tactics. It is certainly a desirable goal to completely automate the task of proof construction. So far however, this approach has been successful only in very limited areas, and, all too often, has resulted in systems that obscure rather than clarify the structure of proofs.

Of course, the language should be sound in the sense that any errors contained in a formalization must be due to the formalization itself and not to the language. For example, a correct proof of a faulty proposition must ultimately result from the (correct) use of a faulty axiom in one of the underlying calculi rather than from an internal inconsistency in the language.

Finally, the language should be supported by various tools. The most important tool is certainly a checker, which checks the correctness of a formalization expressed in the language. Around such a checker, a basic set of support tools should be available. Users should be able to experiment with their formalizations in an interactive environment; the user should be able to draw on a predefined set of standard theories containing formalizations of various logics, data types, etc; likewise, they should be able to store their formalizations for later reuse; and they should be assisted in preparing written documents containing formalizations.

Note that all the above requirements for a generic development language are intended to guarantee that one can express, or better formalize, the formal system underlying a formal method. We do not intend to deal with other aspects of a method such as recommendations, guidelines, and heuristics. Thus, when we speak, in the sequel, of formalizing a method, we invariably mean the formalization of the underlying formal system. This implies that formalization of a formal method in terms of a generic development language gives no indication as to how to *invent* a development; this is left to the pragmatics of the method and to the intelligence of the designer.

The Deva Approach

We now wish to describe the concrete approach adopted when developing Deva with a view to meeting the above requirements.

The most important design decision was the choice of a higher-order typed λ-calculus as a basis for the language. This decision was motivated by several considerations. Typed λ-calculi have served as the basic formalism for research into the formalization of mathematics. The languages which grew out of this research include, for example, the AUTOMATH family of languages [27], [28], [29], [80], the Calculus of Constructions [24], the Edinburgh Logical Framework [50], and the NuPRL system [22] [70]. These so-called Logical Frameworks were mainly used for formalization of mathematics, functional programming, and program synthesis from proof (cf. [54] and [53]). One of the main results of this research was that these logical frameworks proved to be an effective approach to the task of formalization in general, and one which is also amenable to implementation on a machine. The underlying principle here is the so-called Curry-Howard paradigm of 'propositions-as-types' and 'proofs-as-objects'. We do not wish to go into greater detail at this point, but the basic idea is that there is a one-to-one correspondence between the propositions of (constructive) logic and the types of a typed λ-calculus, and between the (constructive) proofs of propositions and the objects of a type. Given this correspondence, proving amounts to functional programming, and proof-checking to type-checking, which is what makes the approach so attractive for implementation on a machine. In the next chapter, we explain this paradigm in more detail and present a number of intuitive examples.

Starting from this design decision, we proceed as follows: we wish to view specifications, programs, and deductions as formal objects which can be formally manipulated and reasoned about, and for which we can formulate correctness properties. For specifications and programs, this is nothing new — they are considered to be formal objects of study in other contexts as well. However, in the case of deduction, it is a somewhat new perspective: deductions are viewed as formal objects relating specifications to programs. This is the key concept in our approach to the formalization of formal methods. When formalizing a method, we do so by stating axiomatically, among other things, which deductions are allowed by the method. Such an axiom describes how a specification is related to a program by a particular deduction.

In the context of a typed λ-calculus, we realize this aim by representing specifications, programs, and deductions as λ-terms. They can be manipulated and reasoned about with the usual machinery that comes with such a calculus. Correctness and consistency issues are handled by the typing discipline of the calculus.

A particularly well-suited logical framework was selected as the starting point for the design of the Deva language: Nederpelt's A, one of the latest members of the AUTOMATH family [80]; see [33] for a recent presentation of A in the spirit of Barendregt's Pure Type Systems [10]. This calculus was chosen, after a number of others had been evaluated, because it is comparatively simple and economical, and because it supports some of the major concepts of structuring in the small,

namely parameterization and instantiation of deductions. Although Λ has not been experimented with in the AUTOMATH project, it does constitute a major scientific contribution of that project. However, Λ remains very much like an ordinary λ-calculus: it is based primarily on binding and substitution, and it fails to provide composition, product, and modules, such as are needed in our approach to program development. The definition of Deva grafts these concepts on to Λ. A second major extension to Λ concerns the distinction, in Deva, between an explicit and an implicit level. The explicit level includes all the extensions to Λ we have just mentioned. The implicit level adds constructs which are instrumental in meeting another important requirement: that of allowing incomplete sketches of deductions, proofs, etc. Parallel to these extensions to Λ, the normalization proofs established for Λ were adapted. These language-theoretical properties are important for demonstrating the soundness of the language, i.e., that Deva itself does not introduce errors into a formalization. To summarize: Deva is to Λ what a functional language is to the pure λ-calculus.

Tool Support

Right from the beginning of the design process, prototypical implementations of type checkers and other support tools for Deva were built and experimented with. These prototypes were not, however, intended for use in full-scale Deva developments, but were rather developed for experimentals purposes. Hence, they supported only selected features of the language's functionality as presented in this book. The two case studies examined here provide ample evidence that medium-scale formal developments are feasible, provided that the user is assisted by the machine via a set of tools.

The design and implementation of such a tools for the full Deva version is the subject of a current Ph.D. thesis [3]. Initial (beta-) versions of this tool-set — called "Deva's Interactive Laboratory" or "Devil" — have been available since late 1992, and they are currently being used for a number of ongoing case studies (e.g. [12], [89], see below). Since the tools are being continously further developed, we give only a brief summary. The structure of the tool-set is shown in Fig.1. This diagram illustrates the current state of the support environment. Direct user experiences will shape its future development. A syntax-check, a consistency-check and an explanation (cf. Chap. 3) constitute the central components of the system. Once a formalization has been checked, it can be stored in an efficient ("compiled") form for later retrieval. The interactive design of and experimentation with Deva formalizations is made possible through an interactive user interface for which both a plain TTY and an X-Windows-based realization exist. Through a database, the user may access previously defined or compiled formalizations.

The design of formal specifications, formal developments or any other formalizations should go hand in hand with the design of their documentation. In fact, good documentation of formal specifications or developments is even more important than documenting or "commenting" programs, because, like any other

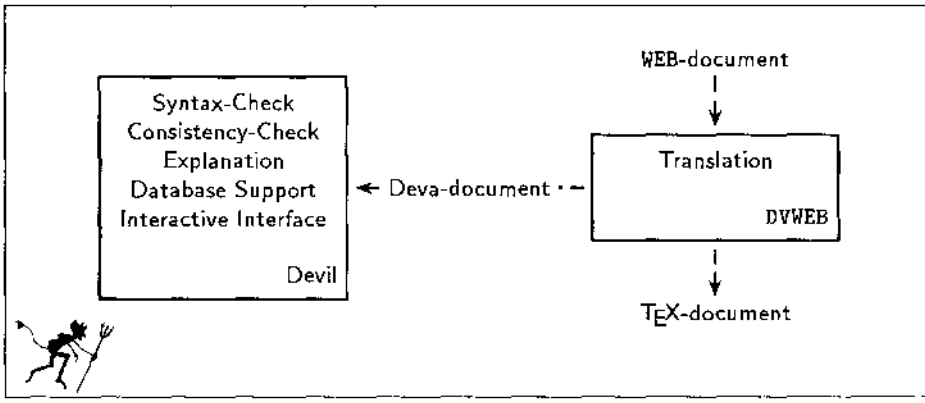

Fig. 1. Structure of the support environment

specification or development, formal specifications and developments are primarily intended as a means of *communicating* with other people. Knuth gives the following motivation for *literate programming* in [63]: "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do." A similar statement might be applied to formal specifications or formal developments: Instead of imagining that our main task is to explain to a computer why a specification or development is correct, let us concentrate rather on explaining to human beings why they are correct. To realize his idea of literate programming, Knuth designed the WEB system of structured documentation. Originally intended for Pascal programs only, various WEB's are being developed for other languages and formalisms (cf. Knuth's recent book on literate programming [64]). Such a WEB tool is also being developed for Deva. The user writes a single WEB document which combines the documentation and the formal Deva code. The presentation of the formalization can be given in a natural, *web*-like, manner, unhindered by the syntactic restrictions of the formal language (an example being this book!). The Deva code portions are pasted together in a preprocessing step to produce a formal Deva document which can then be checked by the Devil system. A second preprocessing step produces a TEX-document which maintains the web-like structure of the presentation and in which the Deva code portions are typeset in an esthetical manner. Furthermore, an index of variables is produced. Details of DVWEB are given in [13].

Since all of these tools were not available at the time we started writing the book, all the formalizations contained in this book were just typed in. The only "tool" around was a set of LATEX-macros which made this task a little less painful. But with these tools available, we decided to rewrite all the chapters containing Deva formalizations, so that Chaps. 2, 4, 5, and 6 are now self-contained WEB-documents which have been checked by the system. It is worth noting that the Devil system revealed numerous errors in the original document. While most of

these errors were easy to correct, some pointed to serious flaws in the reasoning and required significant repair efforts.

Intended Readership

The above introduction and overview of the language should have made it clear — and we will try to demonstrate this in the course of the book — that generic development languages in general, and Deva in particular, offer a useful framework for tackling formal developments. Four different application areas for Deva come to mind. Deva can be used to formalize a method in order to

- impart a precise understanding of the method,
- spot shortcomings in the method or its documentation,
- experiment with libraries of basic theories for software developments based on the method, or
- obtain a prototypical support environment for the method so that formal developments may be constructed, documented, and checked for correctness with respect to the formalization.

The book can be read by anyone with a basic background in formal approaches to software development, as, for example, is given in [60].

Synopsis

The book is divided into two parts. The first part describes the Deva language; in the second documents two case studies. Part one comprises

- Chapter 2 which gives an introduction to Deva, presenting intuitive examples of theories and proofs chosen from elementary mathematics and logic. The goal is to convey an intuitive understanding of the use and properties of the language.
- Chapter 3 which presents and explains the formal definition of Deva and briefly summarizes some theoretical results and still open questions. The goal is to convey a thorough technical understanding of the notation and its design. The material contained in this chapter is based on [104] and [105]. On a first reading, this chapter may be skipped.

Part two comprises

- Chapter 4 which presents a selection of basic logical and mathematical theories, formalized in Deva. The goal is to demonstrate the principal formalization power of Deva on a number of well-known examples.
- Chapter 5 which presents a case study on the formalization of VDM-style developments in Deva. In particular, VDM data reification is formalized. Then, a data reification step from a VDM development in the context of a biological case study is formalized in detail. The material presented in this chapter is based on [67] and [103]. Finally, the formalized data reification is formally proven to be transitive.

– Chapter 6 which presents a case study on the formalization of algorithm calculation. It consists of a formalization of representative parts of Squiggol, also known as the "Bird-Meertens formalism", an algorithmic calculus for developing programs from specifications. A selection of the structures and laws of Squiggol and two complete program developments are described in Deva. The material presented in this chapter is based on [102].

The two case studies can be read independently of each other. It is no accident that they deal with quite different areas; in fact, they have been selected to illustrate the genericity of Deva and to address a wider readership.

Historical Background

The design of an early precursor of Deva was set out by Michel Sintzoff in a series of papers ([91], [92], and [93]) in the early '80s. Deva itself, as presented in this book, was developed mainly between 1987 and 1989 in the context of the ESPRIT project TOOLUSE ([57], [20], [95]). The objective of the TOOLUSE project was to study a broad spectrum of development methods (e.g. Jackson System Design, VDM, or Burstall-Darlington's fold/unfold method for program transformation) and to design a method-driven support environment. Deva was intended to serve as a notational framework to help promote an understanding of such methods. The language was developed by the collaborative effort of three different project subgroups: a group headed by Michel Sintzoff at the Université Catholique de Louvain (UCL), a group headed by René Jacquart at the French Research Center for Technology (CERT) in Toulouse, and a group headed by Stefan Jähnichen at the German National Research Center for Computer Science (GMD) in Karlsruhe. A kernel calculus of Deva was proposed by Philippe de Groote, who also developed its language theory [30], [31], [33], [32]. In the course of the project, several prototype support tools for the evolving versions of Deva have been developed [41]. This book presents the complete language as set out by one of the authors in [104].

Related Approaches

Before turning our attention to currently evolving approaches, we would like to mention a pioneering experiment in the formalization of mathematics conducted in the context of the AUTOMATH project mentioned above. It consisted of the translation of Landau's "Grundlagen der Analysis" into one of the AUTOMATH languages, a translation that was completely checked by machine [99].

Machine support for formal development can be roughly divided into specific support, i.e., support for a single and fixed logic or programming method, and generic support, i.e., support for a range of logics or methods. While the focus of this discussion will be on generic systems, we wish to mention first of all several currently evolving systems that demonstrate the usability of two key techniques for making formal program developments more accessible to humans: The first technique is to program recurring patterns of proof into *tactics* [44]; the second

is to express domain knowledge in abstract development schemes. The *Karlsruhe Interactive Verifier* KIV [51] uses tactics to implement high-level strategies for the development of verified imperative programs. Similarly, in the *Larch Prover* LP [42], a variety of tactical mechanisms are used to provide interactive proof support. LP has been experimented with in a variety of case studies, including the debugging of module interface specifications [48]. The *Kestrel Interactive Development System* KIDS [96] uses a hierarchy of algorithm design schemes such as divide-and-conquer and dynamic programming for schema-driven interactive development.

Of the generic systems, some are primarily oriented towards formalization of logics, and others towards program developments, which means a further sub-division. In fact, a number of generic systems of both kinds are under development, some having been just recently announced, so that it becomes rather difficult to discuss all of them in detail. Instead, we wish to draw attention to three specific generic approaches. All three approaches are characterized by an early focus on building or improving an interactive support environment for — a significant portion of — formal proofs. The design of Deva, which was begun at a later date, is characterized rather by the successive approximation of a notation which could express some representative styles of formal program development reasonably well. Experiments and comparisons were conducted in paper-and-pencil form, using quickly constructed Deva prototypes, or with existing systems from related approaches. This may help to explain some of the differing design choices.

The **B**-Tool is a rule-based inference engine with rule-rewriting and pattern-matching facilities [1]. Initially, the **B**-Tool concentrated rather on automatic proving; explicit proof mechanisms and tactics were added later. The **B**-Tool is generic in the sense that it has no pre-defined encoding of any specific logic. It can be configured to support a variety of different logics by specifying them as so-called "theories". To describe these logics, the **B**-Tool offers a number of built-in proof mechanisms, such as reasoning-under-hypotheses, scoping-for-variables, a notion of quantifiers, metavariables, substitution, equality, etc. Proof strategies may be described in the **B**-Tool by tactics. Both forward reasoning and backward reasoning are supported. The **B**-Tool has been used and tested in a number of formal program developments, each of them typically requiring several hundred mathematical theorems to be proved.

The Isabelle system is an interactive theorem-prover based on intuitionistic higher-order logic [83]. It allows support of the proof construction in a variety of logics. Its main orientation is towards the machine support of proof synthesis. To this end, it offers powerful proof tactics and a concept of backwards proof construction. Unlike Deva, Isabelle is not based on propositions-as-types, but uses instead predicates to formalize the propositions and theorems of various logics. This entails a number of — relatively small — technical complications in the formalization of logics. Isabelle benefits from the firm foundations of intuitionistic higher-order logic, which allow adequacy of formalizations to be proven, and the reuse of well-known techniques such as higher-order unification. A re-

cent article describes the experimentation with Isabelle to prove theorem about functions in Zermelo-Fraenkel set theory [81]. The overall orientation towards genericity and sound foundations makes Isabelle similar to Deva. However, the two approaches do quite differ in practice, perhaps because Deva is oriented more towards theories and program development, while Isabelle is oriented more towards interactive proof synthesis. In addition to Isabelle, several similar systems based on higher-order logic are being developed, some of them oriented towards hardware design [4]. The most widely used one among these is probably the HOL system [45] [46]. A translation of dependent type theory into HOL is proposed in [56].

The mural system is a formal development support system consisting of a generic reasoning environment and a support facility for VDM [61]. The reasoning component is based on a logic with dependent types. mural can be instantiated by a variety of logics. Logical theories are organized in a hierarchical store, containing declarations, axioms, derived rules, and proofs. The main emphasis in the design of mural has been on the interface for interactive proof construction and theory organization. Proofs are constructed interactively in a natural deduction style. The VDM support component provides support for the construction of VDM specifications and refinements. It also generates proof obligations stating the correctness of refinements. The proofs themselves must then be constructed inside the reasoning environment. In [39], the application of mural is demonstrated in the the specification and verification of a small VDM development. The mural system could easily be extended by support facilities for methodologies other than VDM. The Deva approach shares with mural the orientation towards genericity and method support. Compared with the **B**-Tool, Deva was geared to a more powerful use of logical genericity. For example, the formalization of VDM presented in this book can be viewed as a *formal specification* of selected parts of VDM. This VDM specification is then used in the book not only to reason *inside* VDM but also to reason *about* VDM, by proving the transitivity of VDM data reification.

Furthermore, there are a number of relatively new generic systems implementing generalized typed λ-calculi; these include LEGO [74], Coq [36], and ALF [75]. An application of the LEGO system to formalize various logics and proofs is described in [5]. LEGO has also been used to formalize program specification and data refinement in the extended calculus of constructions [73]. For an application of the ALF system, we refer to [23]. Also, we would like to mention the logic programming language Elf [85], based on the Edinburgh Logical Framework [50]; Elf can be used to encode a dependent-type λ-calculus [38] or to directly formalize theories and proofs [86].