Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development

H. EHRIG,[1] B. MAHR,[1] I. CLASSEN[1] AND F. OREJAS[2]

[1] *Technical University of Berlin, Fachbereich 20 Informatik, Institut für Software und Theoretische Informatik, Franklinstrasse 28/29, W-1000 Berlin 10, Germany*

[2] *Technical University of Catalunya*

The intention of this part 1 of an overview paper on algebraic specifications is an informal introduction to formal methods for software development in general and to applications of algebraic specifications in particular. Horizontal structuring and vertical refinement techniques for algebraic specifications are shown to support the general software development process. Moreover, a short overview of case studies and tools in the ESPRIT projects LOTOSPHERE and PROSPECTRA is given. In part 2 of this paper we give a survey of the research field of algebraic specifications developed within the last two decades, which shows how the classical view of algebraic specifications has been extended towards a general theory of foundations of system specifications.

Received June 1992

1. INTRODUCTION

Algebraic specification is a formal specification approach that emerged in the mid-70s as a technique to deal with data structures in an implementation-independent manner. The approach was based on specifying data types in a similar way to that used for the study of different mathematical structures (e.g. groups, rings, fields, etc.) in modern algebra. In this sense, equational logic was chosen as the specification formalism and universal algebra and category theory provided the underlying semantical techniques.

Since then, the research efforts have led to a good understanding of various specification concepts, including structuring mechanism and constructs that can now be found in a number of existing specification languages. Moreover, the influence of algebraic specification has gone beyond the area of specification itself. On the one hand, several modern programming languages (e.g. object-oriented languages, ADA, ML, Miranda) provide constructs whose origins can only be found in algebraic specification. On the other hand, this research field has also contributed to a better language-independent understanding of certain programming constructs. An obvious example are data structures, whose presentation and study have been considerably influenced by the introduction of algebraic specification.

Although, originally, algebraic specification was intended as a technique for the description of data types, it soon grew into a formal specification technique aiming to cover the whole specification phase within the software development process. In this sense, algebraic specification shares the advantages of other formal specification approaches in comparison to informal ones, as follows.

- By 'forcing' the specifier to analyse the problem in question until a level of detail is reached that may not necessarily be reached when using an informal method.
- By allowing for tools and methods to detect ambiguities and inconsistencies otherwise hidden.
- By allowing for prototyping tools that in an early stage of development can be used for detecting inadequacies of the system specification with respect to the customer/user needs.

In addition, within the algebraic specification approach several structuring and modularisation constructs have been defined such that:

- Specifications can be built in a stepwise manner, facilitating the analysis of the problem in question by making systematic use of the principle of separation of concerns in a systematic manner.
- The design phase can make use of the modular structure introduced in the specification phase in order to simplify the system's correct implementation.

Nowadays, algebraic specification is regarded as providing support for the whole software development process.

- Conceptually, algebraic specification provides a framework to formally describe software design. This framework allows for a better understanding of the software development process providing methodological insight concerning different issues.
- Practically, algebraic specification provides methods and tools that can be used in actual software design. In Section 5 two examples of this are shown.

The rest of the paper is organised as follows. In Section 2, we describe a number of aspects of software development in which formal methods (and, in particular, algebraic specification) play a major role. In Section 3, some constructs for the horizontal structuring of specifications are briefly described. Section 4 is devoted to discussing correctness aspects related to vertical development. In Section 5, two algebraic specification-based systems, namely LOTOSPHERE and PROSPECTRA, are described. Finally, in Section 6 some new perspectives are discussed.

2. GENERAL ASPECTS OF SOFTWARE DEVELOPMENT

In this section we give a short review of requirements for software systems, try to summarise the main conceptual stages and steps of the software development process, and discuss the role of formal methods for both of these topics.

2.1 Requirements for software systems

Instead of a detailed discussion of requirements for software systems, which may be found in textbooks on software engineering, we summarise the main aspects with the keywords 'adequacy', 'quality', 'modifiability' and 'reusability'.

Adequacy. First of all the software system should be an adequate solution of the given problem.

Quality. The quality of a software system is mainly determined by its reliability, correctness, efficiency, security and error tolerance.

Modifiability/version update. Software system modifications for the update of versions should be possible with reasonable effort in order to adapt the system to new requirements or to improve adequacy or quality.

Reusability. It is most desirable to be able to reuse the architectural design of the system, and the specification or code of suitable components of the system for the development of other software systems.

2.2 Conceptual software development process

We are aware that there is no commonly accepted model for the software development process from a given problem via specification and design to an efficient version of the software system. But we would like to summarise in Fig. 1 those conceptual stages and steps which we consider to be mainly important in order to discuss the role of formal aspects of software development

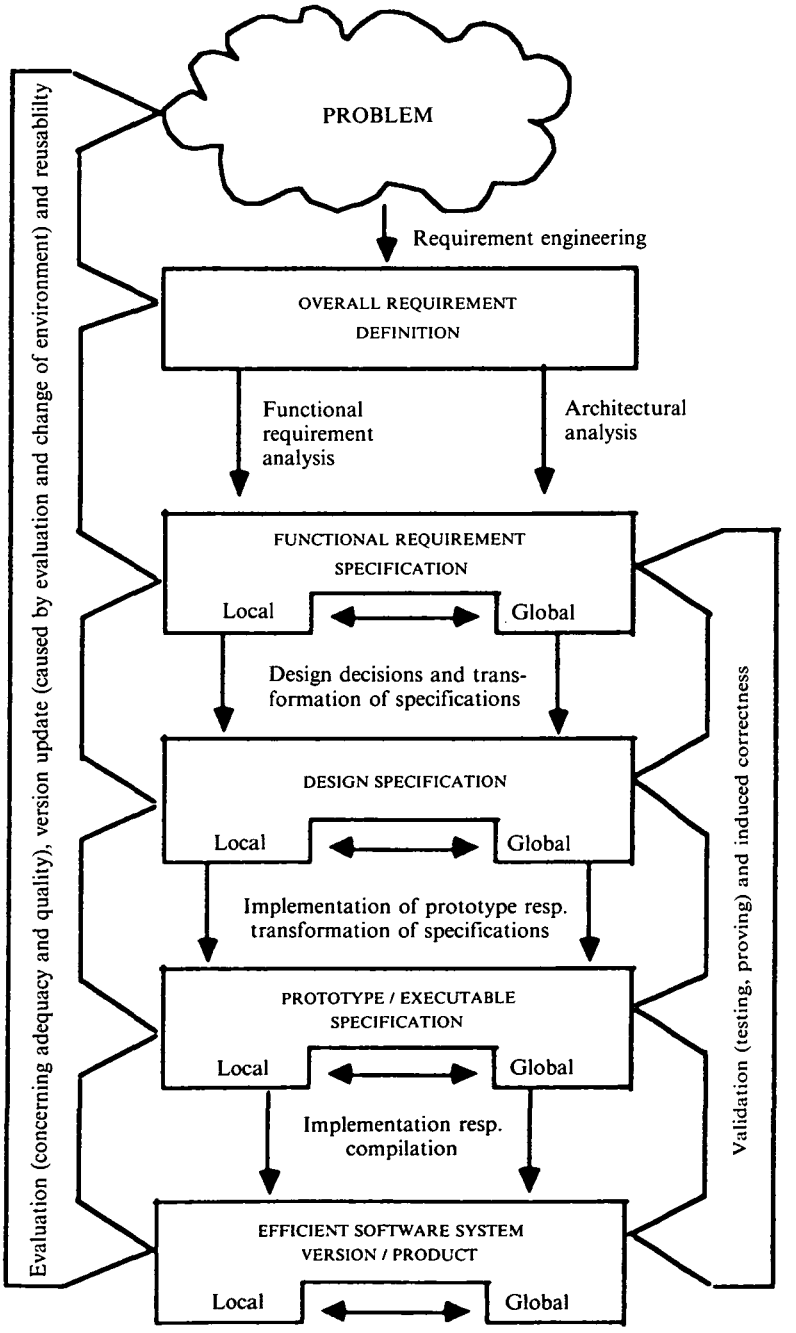


Figure 1. Main conceptual stages and steps in the software development process.

below (for this reason we disregard, for example, all the aspects of software management).

The stages in Fig. 1 are indicated by a cloud for the given problem and by boxes for later conceptual stages in the development. The development steps are indicated by horizontal and vertical edges and the big clamps left and right. The stages and steps are 'conceptual' in the sense that not all of them have to be taken within the actual development process; parts of the system may actually be in different stages and different steps may be performed concurrently.

The bone-like structure of most of the boxes representing the conceptual development stages indicates the 'horizontal structuring process' within each of them which is shown in Fig. 2 in more detail.

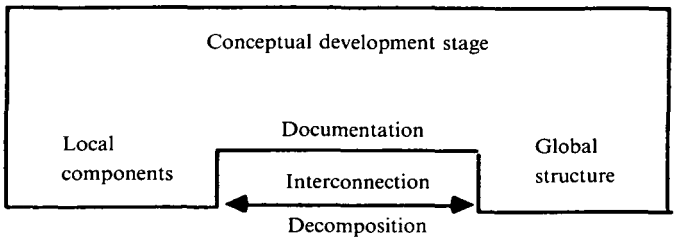


Figure 2. Horizontal structuring of a conceptual development stage.

In each of these conceptual development stages we distinguish between local components and the global structure of the system within this stage. The horizontal structuring process includes the interconnection of local components leading to a global structure as well as the decomposition of large components into small ones.

The complexity of the overall development process increases by the fact that horizontal structuring steps and vertical development steps can be mixed, i.e. an arbitrary path of vertical and horizontal edges in Fig. 1 may be used to proceed from the problem to the product.

2.3 Role of stages and steps and support by formal methods

In the following we want to discuss briefly the role of the conceptual stages and steps within the software development process and the role of formal methods for software specification and development.

(1) *Support for requirements of software systems.* Formal methods are essential to support quality, mainly concerning reliability and efficiency aspects, as well as modifiability and reusability by formal concepts for data abstraction and modularity. Suitable notions of semantics and correctness for both of them should lead to semantically integrated interfaces between the components of the system on all conceptual development stages.

(2) *Conceptual development stages.* The first conceptual stage following the given problem is the overall requirement definition for the software system to be developed. It is usually written in natural language in order to be understandable by the user and the software engineer.

In the functional requirement specification the functionality of the operations of the entire system and – as far as possible – of suitable subsystems should be given, together with the main properties which are required for these operations. Conceptually this corresponds to an algebraic or logical specification including equational, first- or higher-order axioms and constraints, which can be considered as requirements for the operations and their domains. The semantics should be the class of all algebras or structures satisfying the given axioms and constraints.

The design specification is intended to be an abstract model of the intended software system. It should be independent of any particular representation. Conceptually this corresponds to an algebraic specification with tight semantics, like initial or final semantics. This means that the semantics is given by one abstract data type, i.e. an isomorphism class of data types.

If the design specification is already executable it can be used as an early prototype of the system. Otherwise it might be transformed to become an executable specification. Executability means that the operations of the corresponding abstract data type can be correctly simulated by interpretation or compilation of the term rewriting system which can be automatically derived from the axioms of the specification.

Finally an efficient version of the software system, especially the software system product to be delivered, should be implemented in a suitable programming language. This language should have a formal semantics in order to be able to prove correctness w.r.t. the system specifications in previous development stages.

(3) *Conceptual development steps.* The main idea of the conceptual development steps is to describe the development process within one stage and between different conceptual stages of development. The process within one stage, also called 'horizontal development', essentially means to interconnect local components to global structures or to decompose large components into smaller ones. These horizontal development steps should be supported by a suitable formal notion of components (or modules) and semantically well-defined interconnection mechanisms, which can be considered as operations *on* these components (not to be confused with the operations *in* these components defined by the corresponding data types).

The conceptual development steps between different stages are called 'vertical development steps'. Like the horizontal steps, they are mainly governed by design decisions. But the vertical steps should be supported by suitable transformation concepts and tools, which in most cases will be interactive to allow design decisions and in some cases automatic, e.g. the compilation of an executable specification into some programming language. For all these development steps formal methods are highly desirable in order to obtain semantic compatibility within one stage and to allow interactive or automatic correctness proofs between different stages which eventually should lead to correctness of the software product w.r.t. the specifications in all conceptual development stages. In addition to these formal correctness proofs adequacy and quality of the system (see Section 3.1) have to be checked by suitable evaluations leading to updated versions of the system in different conceptual development stages.

3. HORIZONTAL STRUCTURING OF SPECIFICATIONS

In this section we discuss construction mechanisms for specifications which lead to a horizontal structuring of the system specification in the sense of Section 2. This idea was first advocated for algebraic specifications by Burstall and Goguen in Ref. 1, leading to the first algebraic specification language, called CLEAR. Later on several other languages were designed where the language features for specification in the large correspond exactly to certain construction mechanisms for specifications. One important concept for structuring and construction of specifications is that of parameterised specifications,[32] i.e. specifications with a designated formal parameter part, and actualisation of the formal by an actual parameter. In analogy to procedures in programming languages the actualisation process is also called parameter passing.

Another important concept of structuring and decomposing large software systems into smaller units is the notion of modules, first advocated by Parnas in 1972.[29] For all stages within the software development process modules or module specifications are seen as completely self-contained units which can be developed independently and interconnected with each other. In the context of algebraic specifications several different module concepts have been developed supporting these ideas, e.g. in the algebraic specification languages Extended ML[30] and ACT TWO.[13,33] For a discussion of different module concepts in specification and programming languages we refer to Ref. 12.

In the following we restrict ourselves to discussing only some construction mechanisms for basic and parameterised specifications as used in the language ACT ONE,[11] and we only discuss the syntactical aspects, while the corresponding semantics will be discussed in Part 2 of this paper.

3.1 Basic types and operations

A specification for a basic type is given by an algebraic specification

$$SPEC = (S, OP, E)$$

consisting of a set S of sorts, a set OP of operation symbols and a set E of equations or axioms.

Let us consider some structuring mechanisms to build up larger specifications from smaller pieces, called extension, union, and renaming. The *extension concept* allows us to add sorts, operation symbols and equations to a given specification. If **List0** is a basic list specification with empty list and an operation for appending elements from one side only, **List**, including a concatenation operation for lists, is an extension of **List0**. The *union concept* allows us to construct the union of given specifications with shared subspecifications. If we need a length function and an equality predicate for lists, we take the union of the specification **List** with a specification **Nat** for natural numbers and shared subspecification **Bool** for boolean values and extend this union by operation symbols for the length function, the equality predicate on lists and suitable axioms. The *renaming concept* allows us to have a bijective renaming of sorts and operation symbols of a specification.

3.2 Parameterised specifications

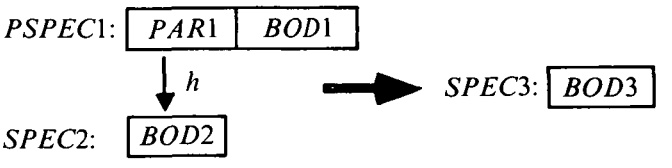
In software development data structures are often needed that share a common principle. For example, lists of natural numbers, lists of characters and lists of records are instances of lists of arbitrary data elements. To exploit such a polymorphism, a parameterisation and an instantiation mechanism are necessary. In the algebraic specification context parameterisation is given by parameterised specifications and instantiation by actualisation.

A *parameterised specification PSPEC* consists of a pair (PAR, BOD) of algebraic specifications, where the formal parameter PAR is a subspecification of the body specification BOD.

Example. In a parameterised specification **List** for lists over arbitrary data elements the formal parameter contains a specification of data elements and the construction of lists over these data elements is given in the body (see Section 3.3 below for more detail).

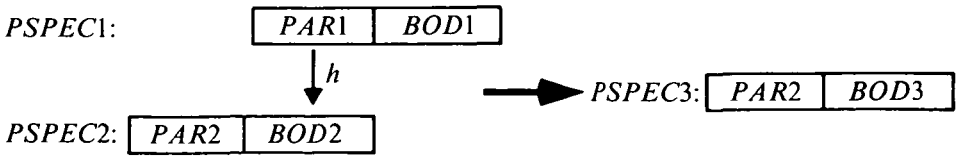
A parameterised specification $PSPEC1 = (PAR1, BOD1)$ can be actualised by an algebraic specification $BOD2$ or by another parameterised specification $PSPEC2 = (PAR2, BOD2)$. These actualisation mechanisms are called standard and parameterised parameter passing respectively. Both of them are using a 'parameter passing morphism' $h: PAR1 \to BOD2$, which defines the replacement of formal sorts and operation symbols by actual ones.

Standard parameter passing



Example. The actualisation of lists over (arbitrary) data (elements) by natural numbers leads to lists over natural numbers.

Parameterised parameter passing



Example. The actualisation of 'lists of data' by 'lists over data' leads to 'lists of lists over data'.

In both cases the body specification $BOD3$ can be constructed by textual substitution of $PAR1$ in $BOD1$ by $BOD2$ according to the parameter passing morphism $h: PAR1 \to BOD2$.

3.3 Example (Construction of texts from lists)

The parameterised specification **List** mentioned is a basic type given in ACT ONE by

```
Type list is
  Parameter
    Sorts  Data
    ...
  Body
    Sorts  List(Data)
    Opns   λ: → List(Data)
           {empty list}
           [_]: Data → List(Data)
           {construction of lists of length 1}
           ∘: List(Data),List(Data) → List(Data)
           {concatenation of lists}
    ...
    Eqns   for all s1, s2, s3: List(Data)
           (s1 ∘ s2) ∘ s3 = s1 ∘ (s2 ∘ s3)
           {associativity}
    ...
Endtype
```

This means that the formal parameter of **List** consists of a sort Data and in the body lists are constructed in the sort **List**(Data) using an empty list, one-element lists and associative concatenation of lists. The dots in the parameter and body part of **List** indicate that the parameter can be extended by further sorts, like **Bool**; operation symbols, like a less-equal predicate considered as an operation symbol with range sort **Bool**; and equations specifying an order relation of data elements; and that the body part can be extended to specify a lexicographical order on lists of arbitrary elements and other operations on lists.

Now we show how to use **List** to obtain lists of lists of data (first actualisation) and lists of lists of characters (second actualisation) leading to a specification **Text** where the sorts List(Char) and List(List(Char)) have been renamed by Line and Text respectively, using a morphism r:

```
Type Text is (List actualised by    List using
                  Sortnames         List(Data) for
                                    Data
              actualised by         Char using
              Sortnames             Char for Data
              renaming
              Sortnames             Line for
                                    List(Char)
                                    Text for
                                    List(List(Char))
Endtype
```

The parameter passing morphisms, called $h1$ and $h2$, map the sort 'Data' to 'List(Data)' in the first actualisation and 'Data' to 'Char' in the second actualisation.

The horizontal structuring of the type **Text** according to this specification is visualised in Fig. 3.

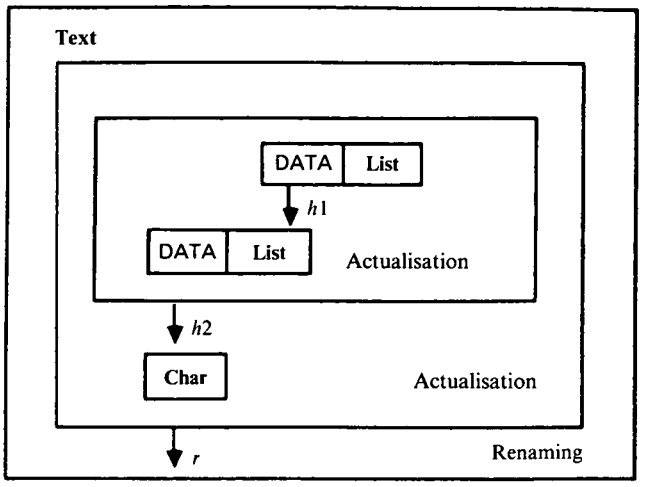


Figure 3. Example of horizontal structuring.

4. VERTICAL REFINEMENT AND CORRECTNESS

In this section we review the main ideas concerning the implementation process starting from a given (formal) specification and leading to a final software system. This process is always seen as stepwise, in the sense that it consists of a sequence of implementation steps. Starting from the 'high-level' specification S_0, it yields a final 'low level' system S_n. In between, different 'intermediate-level' specifications $S_1, \ldots, S_n$ are produced, i.e.:

$$SP_0 \rightarrow SP_1 \rightarrow \ldots \rightarrow SP_n$$

Additionally, implementation is considered to be a modular process in the sense that every step, $S_i \rightarrow S_{i+1}$, consists of a *local* transformation of some specification units of S_i. However, sometimes global transformations are considered: this may be the case in re-structuring steps. It must also be noted that the process may be considered as never-ending, in the sense that the evolution of a software system in the maintenance phase can be considered as part of the development process.

One of the main aims of the use of formal methods in software development is to ensure the correctness of the whole process. In the rest of the section, we first study the different correctness notions involved in the software development process and then we discuss the kind of implementation steps, together with their associated proof obligations, that may occur within that process. In our discussion on correctness we exclude the problem of adequacy and restrict ourselves to 'internal' correctness, i.e. correctness with respect to the given specification. Note however that formal methods also help in the solution of the adequacy problem by construction of prototypes in very early stages of development. Various approaches to vertical refinement in the literature are discussed and summarised in Refs 28 and 31.

4.1 Software correctness

Several different notions of correctness may be associated with a software unit, depending on its kind and its application. For instance, correctness notions for a specification unit and for a code unit are different. Also, correctness of a function implementation is different from correctness of a data-type implementation. Even for the same unit, different correctness notions may be considered depending on its application. For example, in a given context a procedure may be asked to be *totally correct*, while in a different context the same procedure may be asked to be *robust*.

In general, correctness of a software unit refers to the existence of certain relations with other units. This includes the existence of such relations with 'old versions' of the unit, e.g. correctness of a code unit with respect to the implemented specification unit. As discussed in Section 2, we may distinguish between horizontal and vertical relations. Correctness conditions associated with horizontal relations refer to the interconnection of two units, e.g. the correct matching of the formal and actual parameter in the instantiation of a parameterised specification. Correctness conditions associated with vertical relations refer to the fact that a given unit is a refinement or an implementation of another one.

In this sense, global correctness throughout the whole development process must be a consequence of the correctness of all its units. This implies that the given framework must satisfy different compositionality properties. The property of vertical composition states that if a unit U1 is a correct implementation of U2 and U2 is a correct implementation of U3, U1 must be a correct implementation of U3. The property of horizontal composition states that if a unit U1 can be interconnected correctly with U2, and U1′ and U2′ are, respectively, correct implementations of U1 and U2, it must be possible to interconnect correctly U1′ with U2′ and, additionally, the interconnection of U1′ and U2′ must be a correct implementation of U1 and U2.

4.2 Refinement steps and verification

As said above, software development may be considered as a sequence of steps, each of them consisting, normally, in the refinement of a unit into another unit. Steps have associated certain proof obligations to ensure the correctness of the resulting unit. The task of satisfying these proof obligations is called verification.

In general, development steps are 'correctness preserving'. However, in the maintenance phase it is common to perform development steps where some parts of the system are redesigned, providing new units that cannot be considered in any way correct refinements of previous ones. Conversely, in correctness-preserving steps, a (specification) unit is replaced by a 'more concrete' unit (a program unit or less abstract specification unit) that implements the former in a well-defined manner. Algebraic specification has paid special attention to the study of the refinement relations (and their associated proof obligations) associated with data-type units (e.g. see Ref. 28).

Given a specification unit U1 there are, basically, three ways of performing a development step and providing a new unit U2 implementing U1, as follows.

- By direct implementation: the implementor constructs U2 directly 'by hand'. In this case the proof obligations associated with this step concern the implementation relation between U1 and U2.
- By transformation: the implementor constructs U2 by applying some transformation to U1. This may be supported by tools in the software development environment. In this case the proof obligations associated with this step concern the conditions that define the applicability of that transformation.
- By reuse: the implementor constructs U2 by adapting some other unit that had been previously developed in a different context. Again, this may (should) be supported by tools in the software development environment. In this case the proof obligations associated with this step may be related to how the reused component is adapted to fit, since the proofs for the vertical relations of the new unit would probably also be reused.

5. CASE STUDIES AND TOOLS

In this section we give a short overview of case studies and tools that have been developed within DFG (Deutsche Forschungsgemeinschaft) project ACT and the ESPRIT projects LOTOSPHERE and PROSPECTRA. The goal of both projects was to support the specification of realistic software systems and to provide methods and tools for the stepwise development of implementations from specifications. The stepwise development consists of vertical refinements in the sense of Section 4, and is achieved by transformational methods.

5.1 LOTOSPHERE and ACT

The aim of the project LOTOSPHERE was to support the specification and implementation of distributed and concurrent systems. Using methods and tools for specification in the ISO-standardised language LOTOS (Language of Temporal Ordering Specification),[23] we obtained the following results.

- A design structuring methodology for the initial specification as well as for development steps.
- A catalogue of LOTOS-to-LOTOS transformations. These transformations can be used for development steps and to ensure correctness of the resulting implementation-oriented specification w.r.t. the initial specification.
- A comprehensive tool environment to check, transform, simulate, compile and test specifications. The Lotos Integrated Tool Environment (LITE) provides a user-friendly development system with a menu-based user interface.
- Large case studies of specification and implementation of OSI protocols and of an ISDN application. Transaction processing and the related standards for association control and commitment, concurrency and recovery were selected as a basis for the work on OSI protocols. As an ISDN application a service called 'mini mail' was selected that provides a simple mail facility for ISDN subscribers.

The specification language LOTOS is based on a process calculus derived from CCS[26] and CSP[22] and on a variant of the algebraic specification language ACT ONE.[9,11] ACT ONE has been developed within the DFG project ACT (Algebraic specification techniques for Correct design of Trusty software systems). A first tool environment (called the ACT system)[20] was implemented in 1984 on an IBM mainframe. Further developments of the language (see Ref. 5) and experience in the ESPRIT project SEDOS[8] have led to a reimplementation of the system on SUN workstations. The current system consists of a static semantics checker, a completeness and consistency checker and a powerful simulator based on narrowing techniques. The ACT system is mainly used in student projects dealing with the specification of non-trivial software systems. Examples are a syntax-directed

editor, the user operations of the UNIX file system, and the semantics of a functional programming language.

5.2 PROSPECTRA

PROSPECTRA (PROgram development by SPECification and TRAnsformation)[24] is a successful example of a system supporting a methodology covering the complete software life-cycle (including redevelopment), where algebraic specification provides the unifying theoretical framework. PROSPECTRA has been the result of an ESPRIT project involving nine research groups from industry and university coordinated by Bremen University.

The main idea underlying PROSPECTRA methodology is that any activity can be regarded, both at the conceptual and the technical levels, as a transformation of a 'specification'. In particular, not only development steps but also system interaction and proof construction are regarded as transformations. On the other hand, the fact that transformations themselves are considered to be specifications allows the use of transformations for the development of new transformations. This is the basis of PROSPECTRA's meta-development methodology that provides the means for the definition of development tactics, both for specifications and programs and for proofs.

At the technical level, the uniformity achieved through this unified view is the reason for the simplicity of the PROSPECTRA system. The core of this system is formed by four subsystems: the *Library Manager* providing version and configuration control; the *Controller* that handles user interaction with the system and supplies interfaces with all other components; the *Transformer Shell* takes care of displaying, at any moment, the set of applicable transformations in a given context and of actually applying a selected one; finally, the *Proof Subsystem* is the tool to support the (interactive) construction of proofs. In addition to these components, the system consists of a number of editors and translators associated with the various system languages.

An interesting example of a case study developed with PROSPECTRA is the development of two specifications with different degrees of abstraction for describing topologies of solid objects. The more concrete specification was proved to be a correct refinement of the more abstract one.

6. CONCLUSION

Algebraic specification has emerged from concepts in programming, category theory and equational logic and has evolved into a full-grown theory which is concerned with the systematic study of formal techniques applicable to all phases of the software development process. Although initially confined to the specification of data types, the topics of research and development in algebraic specification soon addressed questions of horizontal structuring, vertical refinement and modularisation predominant in all software development stages. Algebraic specification is first of all a mathematical theory with the goal to provide foundations and techniques for software production. Case studies have not only served for insight into questions and requirements arising with system specification, but also validated the techniques and tools in view of their applicability and adequacy. Fundamental to algebraic specifications and to structuring concepts and modularisation in algebraic specification languages are the following: *the algebraic point of view*, i.e. modelling with sorts and operations; *the denotational meaning*, i.e. tight, constrained or loose semantics in terms of classes of algebra; *the principle of compositionality*, i.e. the meaning of composed units as composition of the meaning of the unit's components; *the semantic correctness*, i.e. all syntactic and operational units and steps received their justification through invariants expressed by their denotational meaning.

In keeping with these fundamentals, more and more elaborate techniques have been developed and applied to various languages and systems design. Furthermore, algebraic specification has influenced new language design and new systems architectures also in a less direct way. Examples are functional programming languages like ML or OPAL and the management for open distributed processing in a medical environment at the German Heart Centre, Berlin,[14] where also the use of algebraic specification techniques is being explored within a large research and development project including several academic and industrial partners.

Future work in algebraic specification will emphasise extensions of the formal techniques and the study of prototypical applications with the goal to make the body of work from almost two decades better understood and better available to practical software development.

7. LITERATURE

For reasons of space in this section we have preferred to provide a very small list of references and bibliographic notes that should not be considered representative of the state of the art in the area. Entries have been included in this list sometimes for historical reasons and sometimes for their generality. For a more complete picture of the work in algebraic specification, the reader is addressed to Reference 7.

REFERENCES

1. R. M. Burstall and J. A. Goguen, Putting theories together to make specifications. *Proc. Int. Conf. Artificial Intelligence* (1977).
2. D. Björner and C. B. Jones, *The Vienna Development Method: the Meta-language*. L[ecture] N[otes in] C[omputer] S[cience] 61. Springer (1978).
3. CIP Language Group, *The Munich Project CIP*, Vol. 1: *The Wide Spectrum language CIP-L*. LNCS 183. Springer (1985).
4. CIP Language Group, *The Munich Project CIP*, Vol. 2: *The Transformation System CIP-S*. LNCS 292. Springer (1987).
5. I. Classen, *Semantik der revidierten Version der algebraischen Spezifikationssprache ACT ONE*. Technical Report No. 88-24, TU Berlin, FB 20 (1988).
6. COMPASS Basic Research Working Group No. 3264, *A Comprehensive Algebraic Approach to System Specification and Development, ESPRIT BRA-Proposal* (1988); see also *Bulletin EATCS* **40**, 144–157 (1990).
7. COMPASS ESPRIT Basic Research Working Group No.

3264, *A Comprehensive Algebraic Approach to System Specification and Development, Final Report*. Technical Report, University of Bremen 7/91 (1991).
8. M. Diaz, C. Vissers and S. Budkowski, ESTELLE and LOTOS software environments for the design of open distributed systems. *Proc. 4th Annual ESPRIT Conference*, pp. 543–558. North-Holland, Amsterdam (1987).
9. H. Ehrig, W. Fey and H. Hansen, *ACT ONE: An Algebraic Specification Language with Two Levels of Semantics*. Technical Report No. 83-03, TU Berlin, FB 29 (1983).
10. H. Ehrig and M. Grosse-Rhode, *Structural Theory of Algebraic Specifications in a Specification Logic, Part 1: Functorial Parameterized Specifications*. Technical Report No. 91-23, TU Berlin, FB 20 (1991).
11. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification. 1. Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer (1985).
12. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification. 2. Module Specifications and Constraints*. EATCS Monographs on Theoretical Computer Science, vol. 21. Springer (1990).
13. W. Fey, *Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language*, Diss. TU Berlin, 1988; also Technical Report No. 88-26, TU Berlin, FB 20 (1988).
14. E. Fleck and H. Oswald (eds.), *Neue Techniken und Konzepte der Diagnoseunterstützung bei Herz-Kreislauferkrankungen* (mainly the contributions by Fleck, Hansen, Mahr and Oswald), Blackwell Wissenschaft, Berlin (1992).
15. J. A. Goguen and R. M. Burstall, *CAT, a System for the Structured Elaboration of Correct Programs from Structured Specifications*. Technical Report CSL-118, Comp. Sci. Lab., SRI Int. (1981).
16. J. A. Goguen and R. M. Burstall, Introducing institutions. *Proc. Logics of Programming Workshop, Carnegie-Mellon*, pp. 221–256. LNCS 164, Springer (1984).
17. J. V. Guttag and J. J. Horning, *Preliminary Report on the Larch Shared Language*. Technical Report CSL 83-6 (Xerox), Palo Alto (1983).
18. J. V. Guttag, The specification and application to programming of abstract data types. *Ph.D. Thesis*, University of Toronto (1975).
19. J. A. Goguen, J. W. Thatcher and E. G. Wagner, *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*. IBM Research Report RC 6487 (1976). Also: *Current Trends in Programming Methodology IV: Data Structuring*, edited R. Yeh, pp. 80–144. Prentice-Hall, London (1978).
20. H. Hansen, The ACT-System: experiences and future enhancements. In *Recent Trends in Data Type Specification*, pp. 113–130, LNCS 332. Springer (1987).
21. C. A. R. Hoare, Proofs of correctness of data representations. *Acta Informatica* **1** 271–281 (1972).
22. C. A. R. Hoare, *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, London (1985).
23. ISO, *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard ISO 8807, ISO (1989).
24. B. Krieg-Brückner and B. Hoffmann (eds.), *PROgram Development by SPECification and TRAnsformation, Part I: Methodology, Part II: Language Family, Part III: System*. PROSPECTRA Reports M.1.1.S3-R-55.2, -57.2. University of Bremen (to appear in Springer LNCS).
25. J. Meseguer, General logics. In *Logic Colloquium '87*, pp. 275–329. Elsevier (1989).
26. R. Milner, *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, London (1989).
27. B. Mahr and J. A. Makowsky, *An Axiomatic Approach to Semantics of Specification Languages*, 6. GI-Fachtagung Theoret. Informatik, Dortmund 1983, LNCS 145, pp. 211–220.
28. F. Orejas, M. Navarro and A. Sanchez, Implementation and behavioural equivalence: A survey. To appear in *Proc. 8th Workshop on Alg. Spec.* (*Dourdan, 1991*). Springer LNCS.
29. D. C. Parnas, A technique for software module specification with examples. *CACM* **15**, 5, 330–336 (1972).
30. D. T. Sannella and A. Tarlecki, Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Comp. Programming, Guildford*. LNCS 240, pp. 364–389. Springer (1986).
31. D. T. Sannella and A. Tarlecki, Toward formal development of programs from algebraic specifications: implementations revisited. Extended abstract in: *Proc. Joint Conf. on Theory and Practice of Software Development, Pisa* (1987), pp. 96–110, LNCS 249, Springer; full version to appear in *Acta Informatica*.
32. J. W. Thatcher, E. G. Wagner and J. B. Wright, Data type specification: parameterization and the power of specification techniques. *Proc. 10th Symp. Theory of Computing* (1978), pp. 119–132. *Trans. Prog. Languages and Systems* **4**, 711–732 (1982).
33. H. Weber and H. Ehrig, Specification of modular systems. *IEEE Transactions on Software Engineering*, **SE-12** (7), 784–798 (1986).

Special issue on Spatial Data – Call for Papers

Later in 1993 *The Computer Journal* will be publishing a special issue on **Spatial Data**. This is a particularly important subject at the present time and the issue is seen as being a major contribution to the scientific literature. For this reason a formal Call for Papers is being issued. Papers are invited related to techniques for the processing, storage or retrieval of spatial data or related applications involving spatial data. Novel technical contributions, surveys and tutorial papers all will be considered. They must be written for a general audience of computer scientists, not just for specialists. Please send all contributions to the Editor-in Chief at the address given on the front inside cover of this Journal. **Deadline**: 15 March 1993.