Part I

Invited Lectures

A Decade of TAPSOFT:
Aspects of Progress and Prospects in Theory and Practice of Software Development

Hartmut Ehrig
Bernd Mahr
Technical University of Berlin
Franklinstraße 28 / 29, 10587 Berlin
e-mail: {ehrig, mahr}@ cs.tu-berlin.de

Abstract. The relationship between theory and practice of software development on the background of the driving forces in the 70'ies and 80'ies was the main topic of the first TAPSOFT conference in 1985. After a decade of TAPSOFT the intention of this survey is not so much to give a complete review of the TAPSOFT conferences but to discuss the general background and to focus on specific aspects of theory and practice which seem to be typical for TAPSOFT: The support of software development by algebraic methods, techniques and tools, in particular corresponding activities at TU Berlin. The survey in this paper shows that there is quite a different kind of progress in the decades before and after TAPSOFT'85: Before 1985 the focus was more on the development of new concepts while consolidation and attempts to adapt to practical needs was dominant after 1985.

Finally the expectations for the future of theory and practice of software development are discussed on the background of the driving forces in the 90'ies hoping that TAPSOFT will be able to meet these requirements.

Contents

1 History, Aims and Expectations of TAPSOFT

The idea to have a joint conference on theory and practice of software development was born following the offer to organize the 1985 Colloquium on Trees in Algebra and Programming (CAAP) in Berlin. At that time a joint seminar on the role of formal methods in software development was organized by Christiane Floyd and Hartmut Ehrig. Although it was most difficult to understand each others' terminology and intentions, all of us in the seminar were

convinced that it would be desirable to understand each others research aims and to launch joint projects to bridge theory and practice in software development.

Since there was already an offer to have CAAP'85 in Berlin we proposed that it might be a good idea to combine CAAP'85 with a corresponding Colloquium on Software Engineering (CSE). Discussing this proposal in Ugo Montanari's office in 1983 Maurice Nivat advocated that we should have a third part "An Advanced Seminar on the Role of Semantics in Software Development": The idea of TAPSOFT was born. Last but not least Jim Thatcher was enthusiastic about this idea and joined us to organize the "Joint Conference on Theory and Practice of Software Development", short TAPSOFT, in Berlin 1985.

Although the organization of TAPSOFT'85 with 12 invited speakers for the advanced seminar was not at all easy from the financial and the organizational point of view it finally turned out to be a real success. In fact, it was remarkable to have most distinguished invited speakers like John Backus, Peter Naur, Rod Burstall, John Reynolds, Manfred Broy, Ugo Montanari, Dave Parnas, Cliff Jones, Jim Horning and A. P. Ershov at the same conference, interesting contributions in CAAP and CSE including a session on industrial experience, an extra evening session on social responsibility and several working groups on specific topics.

1.1 Aims and Expectations of TAPSOFT

The overall aim of TAPSOFT was formulated in the call for papers in 1985 as

> "to bring together theoretical computer scientists and software engineers (researchers and practitioners) with a view to discussing how formal methods can usefully be applied in software development"

This overall aim includes two different aspects. The first one is to bring together researchers and practitioners within one conference to discuss each others problems. The second one is the relevance of formal methods for software development. Both of these aspects were lively discussed during TAPSOFT'85.

In fact, TAPSOFT'85 brought together more than 400 participants from research and practice, and a discussion of problems of mutual impact was started at least in some cases. We will discuss below whether this first aim of TAPSOFT could really be fulfilled in the last decade.

Concerning the second aspect let us have a closer look at the aims and expectations formulated in the position papers by Christiane Floyd and Hartmut Ehrig in the proceedings of TAPSOFT'85 [Ehr 85], [Flo 85] and the contributions of the organizers during the panel at TAPSOFT'85.

The main statements in the position papers and during the TAPSOFT panel concerning the relevance of formal methods in software development are more or less the following:

Maurice Nivat pointed out that direct applicability of formal techniques is not his main problem. However, his students should learn formal methods, because good mathematical knowledge of a person leads to good computer science capabilities.

Jim Thatcher reported on his experience within IBM and came to the conclusion that formal specification techniques for large companies like IBM are almost hopeless.

Christiane Floyd discussed the relevance of formal methods in her position paper. She criticized several of the claims in favor of formal methods, but admitted that there are at least some useful aspects. However, the essential part of software development - according to her statement during the panel - is the human interface, which cannot be improved by formal methods.

Hartmut Ehrig focused in his position paper on the question "In what way can formal specification methods and languages improve the software development process?" His conclusion during the panel was that there is a good chance that formal methods will improve the software development process at least in the long run.

So the influence of theory on practice and vice versa was considered quite differently among the panelists. It also turned out that these different views which do in fact not contradict each

other were shared by different groups in the audience.

Before we discuss the role of formal methods and the progress of TAPSOFT in the next sections let us have a closer look at the focus of the TAPSOFT conferences during the last decade.

1.2 A Decade of TAPSOFT

After the initial TAPSOFT conference in 1985 combining CAAP, CSE and the advanced seminar "On the Role of Semantics in Software Development" it was decided to continue TAPSOFT every two years combing CAAP with some more applied conference and a suitable advanced seminar. The proceedings of the TAPSOFT conferences have been published as separate volumes of "Springer Lecture Notes in Computer Science".

The focus of TAPSOFT'87 in Pisa, organized by Ugo Montanari and Pierpaolo Degano, shifted from general software engineering to new programming paradigms, a "Colloquium on Functional and Logic Programming" (CFLP). The advanced seminar was on "Innovative Software Development".

TAPSOFT'89, organized by Fernando Orejas and Joseph Diaz in Barcelona, combined CAAP with a "Colloquium on Current Issues in Programming Languages" (CCIPL) and an advanced seminar on "Foundations of Innovative Software Development".

Tom Maibaum and Samson Abramsky organized TAPSOFT'91 in Brighton and combined CAAP with a "Colloquium on Combining Paradigms for Software Development" (CCPSD) and an advanced course on "Advances in Distributed Computing".

The concept of TAPSOFT'93 in Orsay, organized by Marie-Claude Gaudel and Jean-Pierre Jouannaud, was to combine CAAP with a "Colloquium on Formal Approaches in Software Engineering" (FASE) and to have invited talks on a variety of relevant topics. The same concept is being used now for TAPSOFT'95 in Aarhus, organized by Peter Mosses, Mogens Nielsen and Michael Schwartzbach.

While the contents of the CAAP part was more or less stable from 1985 to 1995, covering in addition to trees a wide range of topics in theoretical computer science, the focus of the second colloquium has shifted from general software engineering, via new programming paradigms to formal approaches in software engineering, especially to algebraic methods, techniques and tools.

Accordingly the focus concerning participants has changed from "general software engineers" to "theoretical software engineers", i.e. to those software engineers, who are already convinced of formal methods and try to find suitable theoretical foundations of software engineering.
This means that the first aspect of the overall aims of TAPSOFT "to bring together researchers and practitioners within one conference to discuss each others problems" has not been fulfilled in the last decade.

Is this already a failure of TAPSOFT? Does this imply a failure of formal methods? In order to find an answer to these questions let us have a closer look at the driving forces of software development and at the relationship between theory and practice. It will become clear that the shift from general to theoretical software engineering was natural and that formal methods had a steady influence on concepts and development in software specification and programming.

2 Driving Forces of Software Development and its Major Concepts in the 70'ies and 80'ies

Already in the late 60'ies it was seen that software could not be produced as fast as desired, in a predictable way, and with guarantees comparable to those customary for other goods like cars or electronic equipment. There was common agreement on the fact that the engineering process of

software development was not well enough understood and that methods, techniques and tools were missing to support this process appropriately. Moreover, there was agreement, though only to a lesser extent, that theory and formal methods would show a way out of the so called 'software crisis'. But in any case it was also quite clear that new techniques had to be developed and better means of abstraction had to be found which would allow to cope with the tremendous complexity of large software systems.

2.1 The Situation in the 70'ies

The 70'ies have then seen an enormous progress in concepts and tools for systems development and programming which laid the ground for most of the achievements in the next decade. Economically, hardware was still the dominant factor in information technology but the awareness grew that software would have a growing market in the future and that productivity of software development was a major aspect to be regarded in research and development.

From a systems point of view, emphasis was on operating systems, compilers, programming tools and interfaces, data bases, information systems as well as application programming and simulation and control. The theories of parsing and translation, program semantics, program verification, queuing theory, data base theory and fundamental concepts for information systems and information retrieval have in the 70'ies been the topics of major concern. Research on Petri nets as a model for concurrency started in this decade and first concepts for parallel programming and synchronization have been developed. Software engineering was established as a new discipline focussing on the software life cycle as a model for the software development process and on programming, specification and verification. The reason for errors in software were analyzed and most of the concepts, languages, tools, and organizational schemata discussed in the 70'ies were seen under the aspect of correctness, effectiveness and rational control of programming and software development. While on the one hand it was enthusiastically discussed how programming could be understood, as a craft, a discipline or an art, the basic fundaments for abstract data types and modularization were laid in the papers by Hoare [Ho 72] and Parnas [Pa 72]. Both, abstract data types and modularization have deeply influenced the design of new more high level programming languages like CLU, MODULA and later ADA as well as the theory of abstract data types and their specification, resulting in specification languages like CLEAR, ACT or ASL.

Also in the 70'ies the study of efficient algorithms and complexity flourished, mainly motivated from the papers by Cook [Co 71] and Strassen [St 69].

Typical for the 70'ies is a new relationship between theory and practice: existing mathematical theories like algebra, logic, probability theory, computability theory, category theory, and theorem proving were found useful in information technology and have been applied yielding new methods and techniques, usually in a less machine dependent way and with all means of abstraction. However, the distance between the newly developed concepts and theories on the one hand, and commercial software development on the other was by no means bridged. Instead there was a steady diffusion of originally theoretical results into systems, systems development and programming. This could be seen in the domains of operating systems, data bases, programming languages and compilers.

2.2 The Situation in the 80'ies

The 80'ies have been different in many respects. The 'software crisis' more and more became an economic problem: a shortage in manpower, low productivity rate, demands for application software that could not be met. The spendings for hardware and software became almost equal

and there was a vision for an even further growing market in the future. Information systems, simulation and computer control became weapons in a cold technology war and in a race for the leading position in the world market. The Japanese 5th generation project, proclaimed 1981, threatened the community with an offensive in super-computing and natural language processing based on unification grammars and PROLOG. The United States responded with their strategic computing and strategic defense initiatives SCI and SDI in 1983. The information processing requirements for the proposed developments in these initiatives were immense. The proposals heavily relied on advances in artificial intelligence and software engineering. Feasibility studies argued in any direction, but increased activities also in Europe, namely in the EUREKA and ESPRIT programs, lead to strong support in parallel computing, information systems and basic research.

The concepts and theories originating from the 70'ies were further elaborated and applied and new programming and specification techniques were developed on their basis. In as much as there was a general belief in the potentials of the theoretical and conceptual developments for their effective use in practical software production, there was a growing scepticism in the adequacy of the software life cycle model (starting from ideas via requirements and design specification to implementation and validation) and in the usefulness of formal methods in software development. The relationship between theory and practice became a matter of concern resulting in an emphasis on prototyping and on tools in software engineering. Theoretical and conceptual developments had to prove their applicability in the implementation of tools, prototypes and in case studies. The rigor of a mathematical treatment was no longer unanimously seen as a value in its own right.

An interesting phenomenon in the 80'ies was the role of the software life cycle model. While it still strongly influenced the theoretical work on programming languages, specification languages and requirement specifications as well as on formal implementation and program transformation concepts, it was objected by those seeing software development mainly as a social process and by those who developed techniques and tools which should support the software life cycle as a whole. The experience from large scale software production showed the need for a much more open process model and the means for general modularization adequate to allow for revision, portability, reusability and unrestricted ways of abstraction to separate concerns. Based on this understanding modular and object-oriented design techniques and architectures, building on object and class libraries, became fashionable and popular.

2.3 New Concepts of Programming

In the 80'ies several new programming paradigms were studied, elaborated and widely applied. Some built on mathematical concepts, others on operational models different from that of the von-Neumann computer:

Functional Programming has its roots in LISP, denotational semantics and a functional style of programming. Backus' Turing Award lecture [Ba 78], in which he presents FP, was a starting point for many researchers in this field. Another source of inspiration was ML that grew out of a meta language for theorem proving and employed strong typing checkable at compile-time. There are also close connections between ML and algebraic specifications. Today ML is a full grown programming language available on PCs.

Logic Programming dates back to the paper by Colmerauer [Col 70] and became known through Kowalski [Ko 79]. But it was the 5th generation project in 1981 that created awareness and world wide activities. Origins of Logic Programming are two-level grammars as used in the definition of ALGOL 68 and theorem proving. Efficient implementations are available now and Logic Programming has proven its usefulness in prototyping and in certain applications like natural language processing.

Object-Oriented Programming originates from the SIMULA class concept [DN 66] and is

motivated from abstract data types. Objects have an autonomous behaviour and cooperate through message passing or shared variables. Objects encapsulate a state and data and have methods for operation. Methods may be inherited from classes or other objects. Object-orientation starting with the SMALLTALK language [GR 83] became strongly influential to system design and system architectures and today plays an important role in software engineering. Since the general concept of object is weakly determined and there are various approaches and interpretations, a common and accepted theoretical foundation of object-oriented systems is still an unsettled question.

Process-Oriented Programming is based on process models which define the means for communication of sequential processes. In the models of Hoare [Ho 78] and Milner [Mil 80] communication takes place via ports or channels in a static communication topology. While process-oriented programming has not yet found its way into practical programming, it originated deep studies of process calculi and process semantics.

Data flow Programming which is based on the asynchronous execution of function application in a static interconnection topology and thereby supports the single assignment principle of programming grew out of the data flow operational model [AA 82]. Computers operating on the data flow principle have been built and shown to be adequate. The architectural concepts of these computers have influenced other approaches, like the LINDA approach to coordination.

A key question in the 80'ies has been to deal with concurrent systems and to gain operational speed by the use of parallel languages. This was also a major motivation for functional and logic programming which by their declarative nature could avoid unwanted sequentialization. The support of software development by formal methods in the 80'ies had its contribution in tools for programming and specification. Conceptually the integration of these tools and their suitability for concurrent systems have been the most prominent topics. From a mathematical and a theoretical point of view the major achievements have come from type theory, rewriting, semantic models and general frameworks for logics and specification. The gap between theory and practice in software development, however, could not be closed, though the influence of conceptual and theoretical work on techniques and tools in practice was, as it seems, very strong.

3 Support of Software Development by Algebraic Methods, Techniques, and Tools

Support of software development by algebraic methods, techniques, and tools concerns the specification phases of the life cycle. A detailed survey and annotated bibliography is given in [BKLOS 91] within the ESPRIT Basis Research Action COMPASS. We here will discuss some of the main concepts and results occurring in different algebraic approaches and look at the progress during the 70'ies to 90'ies, the relation to corresponding mathematical theories, their relevance for software development in theory resp. practice and aspects for acceptance in practice. To be precise, the main focus of algebraic specification techniques in the 70'ies was the specification of abstract data types. The focus was shifted to specification of software systems only in the 80'ies and 90'ies.

3.1 Models and Semantics

The predominant interest in the 70'ies was on "initial" and "terminal" semantics of algebraic specifications, i.e. one model for each specification up to isomorphism. The initial algebra approach, initiated by the ADJ-group [GTW 76], was especially useful for the specification of

basic abstract data types, like natural numbers, characters, strings, stacks and queues. It can be characterized by the slogans "no junk" and "no confusion". The main idea of terminal semantics in contrast is to have one model in the sense of "fully abstract semantics" considered for programming languages.

The next step in the development of algebraic specifications for abstract data types was to consider "loose" semantics, i.e. the class of all models of a given algebraic specification. This was especially useful for requirements specifications within early stages of the software development process and for formal parameters of parameterized specifications, like **data** in **stack(data)**. The focus in the 80'ies then was to consider loose semantics with different kinds of constraints, like generating and free generating constraints, which allow to restrict the semantics to all those algebras satisfying specific properties for suitable subparts of the specification.

Motivated by operational aspects and the idea of hidden states different kinds of behavioural semantics of algebraic specifications have been studied in the late 80'ies and 90'ies, although behavioural semantics had been introduced already by Reichel [Rei 81] as a unifying concept for initial and final semantics. The main idea in behavioural semantics is to define some notion of observation and to consider all models which satisfy the observable consequences of the axioms, but not necessarily the axioms themselves. Behavioural semantics is an important concept to model state oriented systems, where the state is nonobservable.

These different kinds of models and semantics for algebraic specifications were studied in the context of compositionality as well as consistency and completeness w.r.t. corresponding logical calculi. Different models and semantics are seen relevant in different stages of software development from the theoretical point of view, but they are certainly a burden for acceptance in practice because of the large variety of existing approaches and the formal difficulties of the mainly mathematical constructions.

3.2 Logic of Specifications

In the 70'ies and begin of the 80'ies the main focus of research was on many sorted algebras and equational axioms, especially in the initial algebra approach [GTW 76, EM 85]. In [MM 82/84] it was shown that the logic of universal Horn formulas was the most general one within first order logic to admit initial semantics. Problems with error handling in [GTW 76] have motivated to study order-sorted specifications in the 80'ies which also allow to model partiality. In addition algebraic specifications of partial algebras have been studied in the Munich CIP Group [BW 82] focussing on semantics of programming languages.

The intention to have a unified framework to define semantics for different kinds of programming languages has motivated the concept of action semantics and unified algebras [Mos 89] which can be seen as an extension of order sorted specifications and polymorphism. In the late 80'ies the demand to specify higher order functions as they appear in functional programming was the motivation to incorporate higher-order functions into algebraic specifications ([MTW 88]).

Another demand extending algebraic specifications was to deal with problems of concurrency. To consider processes as special data types which interact with each other was adopted in the SMoLCS approach [AR 87] together with corresponding concurrent calculi. In other approaches algebraic specifications are combined with metrics, stream processing functions, Petri nets and modal or temporal logic. All these logics were introduced and studied in view of their semantics and proof calculi in order to support different specification styles and demands as the ones mentioned above.

In order to deal with the different kinds of logic in a unified framework the informal notion of a logical system has been formalized by Goguen and Burstall through the concept of "institutions" [GB 84]. This and other concepts, like "specification frame", have been used in the 80'ies and

90'ies to formulate logic, semantics and structuring of algebraic specifications in a unified way, independent of a specific logic. This was an important step to unify the various theories of specification. Meseguer and others have extended the concept of "institution" by an entailment relation and proof calculus leading to the concepts of "general logics" and "logical systems" thereby following to study the interrelationships between different logics [MM 94]. This is certainly a most important theoretical contribution for the theory of specifications and also for the mathematical theory of logic within the last decade. On the other hand the categorical formulation of these concepts may be an additional burden for its adoption in more practical approaches to specification.

3.3 Structuring and Refinement of Specifications

Structuring concepts for specifications were considered to be important for data types already in the 70'ies. Horizontal structuring concepts allow to build up large specifications from small pieces, while vertical refinement and implementation means transformation steps between different abstraction levels of specifications, e.g. from requirements to design specification. These structuring and refinement concepts are most important for the software development process. In the 80'ies parameterization concepts have been studied for different kinds of semantics and logics and compatibility results were shown for different notions of refinement. In the second half of the 80'ies and in the 90'ies different modularization concepts were studied w.r.t. compatibility and compositionality [BEP 87, EM 90]. The relationship between module concepts in specification and programming was considered in [LEFJ 91] and especially for ML by Sannella and Tarlecki [ST 86]. They also provided a systematic study of implementations in an institutional framework [ST 87]. Moreover parameterization and modularization was extended to behavioural semantics. By now most structuring and refinement concepts are formulated in institutions or specification frames, s.t. they are independent of the underlying logic. This again is elegant from a theoretical point of view but an additional burden for people with more practical intentions usually not familiar with categorical methods.

3.4 Correctness by Verification and Transformation

For the development of correct software systems suitable notions of correctness and corresponding verification techniques are most important. However, one has to make clear which kind of correctness one has in mind and how to verify it.

On one hand there is "internal" correctness of a specification w.r.t. a given base-specification, which means completeness and consistency of new operations w.r.t. the base operations. This notion is important mainly for constructive specifications, like those with initial semantics, and has been studied in the 80'ies and 90'ies based on techniques and results from term rewriting in the case of equations, conditional equations and order sorted equations.

On the other hand there is correctness of refinement or implementation steps between different abstraction levels of specifications. In the case of loose semantics one has to show the inclusion of model classes. For this purpose a proof is required which shows that the axioms of the more abstract specification are derivable from those of the more concrete one. This can be done by correctness preserving transformations or by techniques of theorem proving which, however, are in general very inefficient for first order logic. In the last decade some improvements have been made on the theoretical level concerning computer-aided proof checking. In spite of advances in methods for automatic theorem proving in special cases it seems to be more reasonable to concentrate on interactive proof checkers for verification in the near future. Some examples are the LCF theorem proving system, the theorem prover of LARCH [GG 89], the Boyer/Moore theorem prover, the Edinburgh Logical Framework, Isabelle, and KIV (Karlsruhe Interactive

Verifier) which have been developed or improved during the last decade.

An alternative to verification of correctness of a refinement step between specifications is the use of correctness preserving transformations. Corresponding techniques for program transformation have been developed by Burstall and Darlington and the CIP-group in Munich in the 70'ies and begin of the 80'ies. In the last decade these techniques have been extended to correctness preserving transformations of specifications, for example in the ESPRIT project PROSPECTRA in which the PROSPECTRA transformation system [Kri 90] has been developed. Today such transformation systems are still highly user interactive and experts are needed to achieve the desired results.

A third important notion of correctness is that of specifications w.r.t. programs in a suitable programming language. In some restricted cases there are compilers from specifications into imperative languages, like Pascal [GHM 87]. The more promising way is to transform algebraic specifications into functional programming languages like ML or OPAL, for which efficient compilers exist or have been constructed recently [FGGP 93]. Correctness of specifications w.r.t. programs can be considered as a special case of model correctness where results on induced model correctness [EM 85] are important to construct models for large specifications from those of the components.

3.5 Specification Executability, Language and Tool Support

Executability of specifications is a most important property from the practical point of view because it allows early prototyping on a very high level. The prototyping tools for algebraic specifications are mainly based on term rewriting and narrowing where significant improvements have been achieved during the last decade. For the efficient implementation of functional programming and term rewriting languages graph reduction is a common place technique, where terms are represented as graphs so that common subterms are shared and graph transformations is used.

The need for specification languages to support the construction of specifications for data types and software systems was clear from the very beginning.

The first specification languages for algebraic specifications like CLEAR, OBJ, LOOK, ASL, ACT ONE, PLUSS and LARCH were developed in the late 70'ies and early 80'ies, where OBJ and ACT ONE are based on initial and the others on loose semantics. In the last decade several new algebraic specification languages have been developed to support new specification concepts: Extended ML to support functional programming in standard ML, OBJ2 and 3 for order sorted specifications, ACT TWO for module specifications, SPECTRAL and SPECTRUM for higher order types and TROLL light for object orientation. Especially important from the practical point of view is tool support for specifications including editors, analyser, interpreters, compilers, theorem proving and graphical visualization tools. Experimental versions of several tools were developed before 85, but more advanced versions and the development of support environments were done only in the last decade. Typical examples are the RAP system, the ACT system and ASSPEGIQUE supporting specification in ASL, ACT ONE and PLUSS respectively.

3.6 Development Methodology and Integration of Techniques

It was pointed out in section 2 that an important aspect of software development from the practical point of view is a suitable development methodology for the software development process including all stages and steps from informal problem description, via formal requirement and design specifications to efficient implementation in suitable programming languages. In the 70'ies and early 80'ies the methodology problem was discussed at most on an informal level but in the

last decade it has gained more and more importance, especially in ESPRIT projects with industrial partners. Today it is considered an integral part of most joint projects between academic institutes and industrial companies. In the KORSO-methodology [PW 94] one of the main problems is to provide a suitable description of all the activities and the technical documents which are created and updated during the development process. Especially it indicates how this process can be supported by graph transformations. While formal specification techniques have focussed on the phase between formal requirement and design specifications in the 70'ies and early 80'ies the problem of requirements engineering with formal methods has been considered in the last decade [DP 89]. Meanwhile it turned out that the acceptance of formal methods can be increased considerably if it is possible to extend semi-formal methods to formal ones [GM 87] or to integrate well-established semi-formal methods used in practice with suitable formal specification techniques (see next section for more detail). The last topic was the subject of the GI-workshop "Integration of semi-formal and formal methods for software development" during the IFIP-world congress in Hamburg in 1994. Another topic which is most important for applicability and acceptance of formal methods in practice is the integration of different formal specification techniques. Presently it is a deficiency of algebraic specification techniques, in contrast to model oriented techniques based on VDM and Z, that there are too many different approaches which are not sufficiently integrated with each other.

In fact, it would be of great importance for the acceptance of formal methods in practice if there would be a suitable integration of model-oriented and algebraic techniques. A first indication that this is possible from the semantical point of view is given by Hodges in [Hod 94] where specific semantical constructions corresponding to the different approaches are shown to be natural isomorphic functors. Concerning integration of specification techniques for concurrency like CCS and Petri nets with algebraic ones we refer to LOTOS [LOTOS 88] and algebraic high level nets [EPR 94]. Although these integrated concepts are not yet fully developed it seems to be a major progress within the last decade that their fundament could be worked out.

4 Algebraic Specification and Graph Transformation at TU Berlin

Berlin was among the first places in Europe which strongly emphazised the field of software development from a practical point of view mainly influenced by Kees Koster in the 70'ies and Christiane Floyd in the 80'ies. Research on mathematical concepts and theoretical foundations of data types and software systems started also in the early 70'ies concentrating on graph transformation and algebraic specification, and in the 80'ies on software development in its various phases and on the integration of specification techniques in the 90'ies.

In both cases, algebraic specification and graph transformation, the main idea was to study fundamental properties first, using constructions and results from category theory in the context of total algebras and homomorphisms. In a second phase the results were generalized in a categorical framework s.t. not only the total, but also the partial case as well as other cases interesting from an applications point of view could be handled as explicit examples of the general case. Moreover the integration of algebraic specification with graph grammars was studied as an interesting example for the integration of different specification techniques relevant for practical applications.

4.1 Algebraic Specification

Research on algebraic specification techniques in Berlin started in the late 70'ies. We were mainly influenced by the ADJ-group to follow the initial algebra approach, especially concerning parameterization and implementation of algebraic specifications. The state of the art of our specification techniques at TU Berlin in 1985 is more or less given in our first EATCS-Volume

[EM 85] on "Fundamentals of Algebraic Specification: Equations and Initial Semantics". In fact, this approach was still within the spirit of universal algebra and varieties leading to a clean mathematical theory of specification and correctness of abstract data types using algebraic specifications with initial algebra semantics. Moreover, it included parameterized specifications and parameter passing techniques as the main concepts of our algebraic specification language ACT ONE.

The restriction to total algebras, equational axioms and initial semantics in [EM 85] is nice for an introduction into the theory but turned out to be not fully appropriate for practical applications in software development. This became clear in the ESPRIT-projects SEDOS and LOTOSPHERE where ACT ONE was used for the data type part of the language LOTOS for specification of concurrent and distributed systems. This observation has motivated several extensions of ACT ONE during the last decade, including partiality, constraints and shared subtypes. Some of these, especially operational semantics and tool support are presented in our AMAST-book [CEW 93].

Another major development in Berlin - in cooperation with the universities in Dortmund and Los Angeles (USC) - was the concept of algebraic module specification, where in addition to the parameter part of a parameterized specification explicit import and export interfaces have been introduced. It could be shown that there are powerful interconnection mechanisms, like composition, union and actualization with important compositionality results concerning correctness and semantics. Our second EATCS-Volume [EM 90] presents this theory of module specifications as well as a theory of constraints which is most useful for the interfaces in practical applications. The language ACT TWO is purely based on these concepts [Fey 88] and a more practical variant, called Π-language, was developed and implemented in the Eureka Software Factory (ESF) project and the Fraunhofer Institute (ISST) in Berlin. There the module concept was extended to study module and configuration families and the relationship to module concepts in programming languages [LEFJ 91].

A third major development at TU Berlin within the last decade was a generalization of parameterized and module specifications on the level of specification logics or frames in the sense of indexed categories [EBCO 91]. In fact, specification frames can be considered as a variant of institutions in the sense of Burstall and Goguen, which define directly model categories for specifications without explicit satisfaction relation. There are four important existence properties - to be stated as axioms - which allow to obtain the main results of parameterized and module specifications given in [EM 85] and [EM 90] for abstract specifications within a specification frame [EBCO 91, EG 94]: Pushouts, Free Constructions, Amalgamation, and Extension.

In fact, some of the results need only part of these properties and amalgamation implies extension. These properties are valid for equational algebraic specifications as considered in [EM 85, 90] and a large number of variations, including algebraic specifications with conditional equations, universal Horn axioms, partial algebraic specifications, projection specifications and in a restricted sense also behavioural specifications. It is worthwhile to mention, that a variant of behavioural specifications, called view specifications, does not satisfy amalgamation but only extension, so that only part of the general theory can be applied.

The main reason for these general studies is the fact that most of the constructions and proofs in [EM 85, 90] have been given already on a categorical level, which justifies from today's point of view the restricted kind of models and logics used in 1985 in [EM 85]. It turns out that the same results are now applicable to specification techniques that are much more general and suitable from an applications point of view.

Most recently interesting extensions of algebraic specifications and abstract data types meant to model systems with dynamic behaviour have been introduced by Gaudel, Astesiano and others, where dynamic operations are modelled by transformations between algebras and abstract data types. This motivated us to propose a concept of dynamic abstract data types in [EO 94] which

seems to be promising for the future development of algebraic specifications and their integration with other specification techniques (see 4.3).

Finally let us mention two other activities at TU Berlin: the design and efficient implementation of the language OPAL (by the group of Peter Pepper) which can be considered as a significant progress towards algebraic programming languages [DFG+ 94] useful as target languages for the transformation of algebraic specification with efficient implementations.

Motivated from the need for a unified approach to the various algebraic specification techniques and from studies of semantics in natural language processing a general type theoretic framework for specification and type disciplines was proposed and investigated in the group of Bernd Mahr [Ma 93]. This framework also allows for selfapplicable functions, reflexive domains and an intensional theory of non-well-founded sets.

4.2 Graph Transformations

The algebraic approach of graph grammars has been created at TU Berlin in 1973 and developed in cooperation with B. Rosen (IBM Yorktown Heights) in the 70'ies and early 80'ies as a graph transformation technique with various applications in Computer Science and Biology [Ehr 79]. At the time of TAPSOFT'85 there was already a well-developed theory for algebraic graph transformations including results about independence, parallelism, concurrency, amalgamation, and embedding of derivations and canonical derivation sequences. In this approach a direct derivation is defined via two pushouts in the category of graphs and total graph morphisms. For this reason it is also called "double pushout approach".

Within the last decade the algebraic approach to graph transformations was extended in several ways. First of all distributed graph transformations have been introduced where a distributed state can be modelled by a family of local state graphs which share suitable interfaces. A distributed derivation is a family of local derivations which preserve the interfaces or change them in a compatible way. This concept was later integrated into an approach for modular graph transformations [EE 94], which also includes an inheritance concept and an import-export concept in analogy to algebraic module specifications.

Another important step was the development of high-level-replacement (short HLR) systems [EHKP 91] which allow to handle transformation systems for several variants of graphs, like hypergraphs, graphs with partially ordered labels, relational structures, algebraic signatures and algebraic specifications, in a uniform way. The main idea was to reformulate the double-pushout approach in a general category instead of the category of graphs, and to formulate specific categorical properties for their use in the constructions and proofs, as axioms for the theory of HLR-systems. These HLR-properties have to be checked for the corresponding application categories. In addition to the examples mentioned above HLR-systems have been applied to different kinds of Petri nets leading to net transformation systems which allow a rule-based change of the net structure in addition to the token game [EPR 94].

A third important step in the last decade was the development of the "single-pushout approach" for graph transformations in the PhD Thesis of M. Löwe [at TU Berlin, 1990]. The idea to formulate graph rewriting by a single-pushout is due to Raoult [Rao 84] and was reformulated by Kennaway [Ken 87] but the appropriate mathematical formulation as a basis for a powerful theory is due to Löwe [Löw 93]. The single-pushout approach is based on the idea that a graph rule is given by a partial graph homomorphism and a graph transformation is given by a (single) pushout in the category of graphs and partial graph morphisms. The explicit single-pushout construction for partial morphisms is much more complex than the double-pushout construction for total graph morphisms but avoids the specific applicability condition of the double-pushout approach, called gluing condition. This admits a wider range of applicability, e.g. in information systems with implicit deletion, but can also be restricted to simulate exactly the double-pushout approach.

Although the explicit construction of a derivation step in the single-pushout approach is in general more complex than in the double-pushout approach it turned out that the theory concerning independence, parallelism, concurrency and amalgamation is easier in the single-pushout approach due to the categorical treatment of one instead of two pushouts. The single-pushout and the double-pushout approach have both been implemented in the AGG-System and the GRADE-ONE-System at TU Berlin and have been used in several projects.

Finally let us mention a new interesting development for graph transformations in cooperation of Pisa and Berlin. Petri nets are well-known as a powerful concept for true concurrency. Nevertheless there are some limitations concerning the modelling of states and transitions which are avoided in algebraic graph grammars, like graph structure of states instead of sets of markings and context conditions for graph transformations.

In [CELMR 94] we have started to develop an event structure semantics for graph grammars similar to that for place transition nets which is a first step towards a theory of graph grammars as a new model for true concurrency.

4.3 Integration of Specification Concepts

In the 70'ies and begin of the 80'ies specification techniques, like algebraic and model theoretic ones, CCS, Petri nets and graph transformations have been developed more or less independently so that there was almost no integration of different concepts.

In the last decade it became more and more important to integrate techniques for process specification with those for data types in order to handle practical applications in an adequate way.

At TU Berlin we were involved since 1985 in the development of the language LOTOS within the ESPRIT projects SEDOS and LOTOSPHERE, where process specifications based on CCS were combined with data type specifications based on ACT ONE. Although this integration is not fully satisfactory from todays point of view it was an important step leading to an international standardization of LOTOS and to several practical applications in the area of communication protocols. Another important step was the integration of Petri nets with algebraic specifications, leading to the notion of algebraic high-level nets, a specific version of coloured nets and high-level nets in the sense of Jensen. Our approach in [Hum 89, DHP 91] and [EPR 94], influenced by Vautherin [Vau 87] and Reisig [Rei 91], allows to extend structuring and compositionality results from algebraic specifications to algebraic high-level nets.

An application of HLR-systems (see above) to algebraic high-level nets leads to the concept of algebraic high-level net transformation systems which are an interesting integration of Petri nets with graph transformation techniques [PER 93]. This has been applied to improve the requirement analysis phase of a larger case study in the BMFT-project KORSO.

Another application of HLR-systems to algebraic specification led to the notion of algebraic specification grammars which have been used by Parisi-Presicce for a rule-based approach to modular system design.

An integration of graph grammars with algebraic specification yields the concept of attributed graph grammars [LKW 93] which is suitable to model the graphical and the data type parts of system states in a unified framework. This integration of graphical and data type features within a single formal method seems to be very important for the acceptance of formal methods in practice. In fact, most of the semi-formal methods used in practice combine graphical and textual parts and it is therefore promising to develop techniques which allow a smooth integration of semi-formal methods and attributed graph transformation techniques. A first step in this direction is taken in [CLWW 94] where the static aspects of entity relationship diagrams are modelled by attributed graphs and the dynamic ones by attributed graph transformations. Moreover, this technique has been used to define dynamic abstract data types based on algebraic graph transformations [ELO 94/95] and has been applied to the problem of shipping software in [EB 94].

5 Theory and Practice of Software Development in the 90'ies

Following the considerations in the previous sections we summarize now the achievements in the 70'ies and 80'ies and discuss the contributions of TAPSOFT in the last decade. Finally we formulate our expectations for the future based on a discussion of driving forces in the 90'ies and the role of software development and formal methods in a true application context.

5.1 Achievements in the 70'ies and 80'ies

The relationship between theory and practice of software development in the 70'ies and 80'ies was not stable and appeared under different perspectives in a quite different light. In retrospective one can say that, no matter what the causes of the software crisis were, there was a definite lack of theory and conceptualization of the software development process. But it is at the same time clear that not all aspects of this process are accessible by formal methods. It is fruitless to debate which aspects are more important, those which can be addressed by a mathematically based theory or those which concern social, psychological or cognitive phenomena of the development process. With the advance of technology also the characteristics of the software crisis changed. It is therefore not surprising, that the cure for the deficiencies seen in the 70'ies did not equally apply in the 80'ies or even 90'ies. In fact, progress in theory and formal methods became effective in a pace slower than progress in hardware, in end user facilities and in means of communication, and at the same time pressure from the market and from applications for quick solutions grew strongly. As a consequence the distance between theory and formal methods on one side and practice of software development on the other widened and became increasingly harder to bridge.

On the other hand, the achievements in theory and formal methods, in the 70'ies and 80'ies had an essential influence on the practice in software development. The study of abstract data types in an algebraic framework is a good example. It influenced not only the features of new programming languages but also lead to the abstraction techniques of modularization and object-orientation which are key elements in todays software development and architectures. But also vice versa, the practical requirements had an essential influence on the theory. The development of specification techniques, for example, reflected the need for improved expressiveness, higher flexibility, integration of concepts and availability in the form of tools. It also turned out that abstract frameworks, like institutions or specification frames, were not only motivated from a purely theoretical point of view, but also from considerations of applicability of formalisms, namely when these formalisms had to be changed.

In summary one can say that there was a mutual influence in theory and practice of software development in the 70'ies and 80'ies. Major conceptual developments fall in the 70'ies, while the progress in theory and formal methods in the 80'ies is to a large extent due to elaboration and consolidation of results from the previous decade. The expectation of some people that formal methods could be applied directly and with considerable benefit in practical software development, however, turned out to be too optimistic. First, because the state of development of these methods had not even in the 80'ies reached a point where they could readily be used by the software writers in competitive environments of industry or application sites, and secondly, because failure or success of software development was dependent on many other factors, which relativised the role of formal methods in the opinion of those having responsibilities in the development process. On the other hand formal methods are of increasing importance for specification and verification of safety critical systems.

5.2 The Contribution of TAPSOFT

Looking back now at the series of TAPSOFT conferences and the effect this series had on theory and practice in software development, one may come to the following conclusion: TAPSOFT had started in a time where the pressure on science and academia was very high to contribute to ways out of the software crisis. This pressure was an international phenomenon and TAPSOFT'85 reflected this not only in its technical part but also in its panel discussions. The expectations expressed with this conference turned the open questions at that time into a program. The position statements of Jim Thatcher, Christiane Floyd, Maurice Nivat and Hartmut Ehrig, however, already express the difficulties to be faced. It turns out that, in retrospect, all four have made a good point and seen clear enough the reasons why a bridge between theory and practice in software development was hard to build and to maintain.

On the other hand, the idea of "bringing together researchers and practitioners within one conference to discuss each others problems" was still appealing and, in a sense, unavoidable if one wanted to address the relationship between theory and practice in software development. In the run of the series TAPSOFT has made a shift from "general software engineering" to "theoretical software engineering". The counterpart to CAAP as the theory component of the joint conference became a series of conferences which from a theoretical point of view were more applied, but from a practical point of view were still very theoretical. The choice of topics reflects the focus of research of the time proceeding. The contribution of TAPSOFT is therefore in line with the general progress in the late 80'ies and early 90'ies. But TAPSOFT is not a forum for the difficult topic of theory and true practice of software development and also it does not cover the full spectrum of formal methods. Instead, TAPSOFT is successful in combining true theory with applied topics.

5.3 Driving Forces in the 90'ies

The driving forces for software development in the 90'ies have changed again. The idea of reusability of software became critical and for reasons of productivity and cost effectiveness by far more challenging than initially expected:

Migration of existing software to other new hardware or software platforms, interoperability with existing and often as stand alone systems designed components, and systems re-engineering and code transformation are industrial and application demands with the potential of large economic losses or wins.

New systems development is today more like the configuration of existing components rather than design and implementation from scratch. The large amounts of available software products have lead to software life cycle phases which follow strategies completely different to those of the old 'waterfall'-model of the 70'ies and 80'ies.

Also the landscape of software applications has changed since the 80'ies:
Communication nets, the processing and transfer of data of different mediality, like text, image, film, video, sound, and the progress made with end user facilities have broadened the range of computer support in a revolutionary way. Functionality of systems can no longer be seen in isolation, but is tightly connected with processes of real life enterprises. Widespread or even world wide networks have introduced the concepts of autonomy of system components, of heterogenity and of openness in the most extreme sense. Software development in this context requires new techniques and organizational structures since systems borders can not be assumed to be fixed and no central authority can be presupposed which controls the consistency of the systems behaviours.

Another type of application has gained increasing importance. Safety critical applications with extreme requirements on timeliness, security, operability and correctness, namely in the field of

embedded systems, appear more frequently and require assurances for which appropriate and reliable techniques are not yet ready at hand.

Now, after two decades of conceptual development an abundance of methods, techniques and tools has been created to support the various phases of software development. Their practical usability is proven in most cases but the questions of their use in an industrial context in daily routine and of their contribution to system quality are still open to a large extend. Even though the limitations in most of the underlying concepts were seen, the theoretical and conceptual achievements form a rich basis suitable for pragmatic approaches to build on. This is already visible in object-orientation which became the most prominent and successful concept of todays software and systems development. It grew out of the search for a flexible modularization technique and is appropriate to cope with dynamic topologies of interconnection, heterogenity and openness. It applies to several of the software life cycle phases and is the key for standardization of architectures, interfaces and behaviour.

Theory and practice today have further separated and the pressure for marketable solutions and routine application has increased. But again, it seems that new technology can not be thought without the contributions from theoretical and conceptual work. The question is therefore anew what formal methods can do in the future.

5.4 Software Development and Formal Methods in a True Application Context

Software development is not an isolated activity but is bound to certain intentions and usually embedded in a competitive environment. In one extreme the development of software is part of the production of a product to be marketed. In another extreme the development is part of the realization of a particular application in a real life enterprise. In both cases the development is strongly influenced by factors outside the life cycle model, but also in both cases the rationality of the development process is an important aspect which among others is crucial for the developments success. The following example is of the second type and shows well the relationship of theory and practice of software development in the beginning 90'ies , and the potentials and deficiencies of formal methods.

In 1989 the Project Group Medicine Informatics at the German Heart Institut and the Technical University in Berlin was established, headed by the cardiologist Eckart Fleck and by Bernd Mahr. The group, directed by Horst Hansen, developed a distributed information management system for the integration of heterogeneous networks, system components and patient related data and documents (called HDMS). The system was designed to be open and to evolve with increasing integration, new applications and extended use [FHMO 91]. Based on TCP / IP protocols the session layer and the application layer of the ISO reference model were implemented by communication libraries upon which the core of the management system, the so-called object machines with their object societies were placed. The HDMS object model was designed to meet the requirements of the medical environment and allowed for integration of heterogeneous software components and data, see [Ku 94]. It is conceptually based on type-theoretic considerations and implemented in ML. The prototype implementation was used as the management component which on the one side integrated the extremely heterogeneous and distributed complex infrastructure in the German Heart Institute, and on the other side served as a platform for the various medical applications including medical documentation, image processing and archiving. Two integrated medical application systems, an in-house information system and a net interconnecting cardiological practitioners and hospitals with the German Heart Institute were implemented and ran on the HDMS platform.

The described work was preceded by a thorough document-oriented analysis of the data and data flow in the clinic. Among the investigated 120 documents and clinical processes several were used in the implemented applications. These applications were later reimplemented on a more

conventional data base-oriented platform, using DCE as a network operating system, and the HDMS development was no longer continued. The reasons for abandoning this line of development were manifold and can be explained by the time and the environment in which it took place. The more practical the system became the more it became a subject of economical and strategical considerations disregarding the conceptual advantages and potentials it has proven in its clinical routine use. The basic concepts of HDMS, however, have been used as a fundament for the next generation of integrated medical applications [Fl 94] developed by the PMI.

An interesting question is that of formal methods in the HDMS development. Most of the project group members were mathematicians and well-trained in formal methods. The systems implementation, however, was not preceded by a formal specification phase. It was mainly the pressure from the application site and from the German Telecom which funded the project, that not sufficient time was found to proceed that way. On the other hand the development was based on mathematically well-founded theoretical concepts and the functional language ML was used for its advanced type discipline, higher-order functions and modularization features.

But there is another activity which relates HDMS and formal methods: In 1992 the KORSO-project, funded by the German Minister of Research and Technology, has chosen HDMS for a case study on formal methods in correct software development (see [BJ 94], [CHL 94]). For this reason an abstract version of some of the medical applications were formulated, called HDMS-A, and then used to study the use of algebraic specification techniques and issues of correctness. The choice of HDMS-A focussed on some of the key documents of the patient record and their interconnection as well as their processing in the clinical procedures. In case studies these were specified and investigated. This mainly included the following activities (see [CHL 94]):

1. Based on the document-oriented analysis of the data and data flow in the clinic, a semi-formal state-oriented system definition of HDMS-A was given using the algebraic specification language SPECTRUM for the static data types and condition event nets for the dynamic processing.
2. Using the technique of "functional essence" by McMenamim and Palmer, the semi-formal description of data by entity relationship diagrams and of data flow by data flow diagrams was given.
3. Entity relationship diagrams and data flow diagrams were then transformed into a formal requirement specification in SPECTRUM, using axioms for integrity constraints and stream-processing functions for data flow diagrams [Nic 93].
4. Safety and security aspects have been added to the SPECTRUM specification using the PASCAL-like language of the Karlsruhe Interactive Verifier (KIV).
5. The integration of pre-existing components, which are not formally specified, was considered and two possible solutions have been proposed: Either by an event driven specification of the user interface in SPECTRUM, or by translating parts of the formally specified system as standard system calls of the non-specified system.

As a result of this case study one can conclude that means of formal specification are at hand to tackle in principle the problem of formally describe complex systems like HDMS, or at least components of it in some phases of the life cycle model. But the integration of specification techniques is still not available in a satisfactory way and the steps from semi-formal to formal specification still need careful consideration.

At the level of requirements these observations have motivated a second case study, called HDMS-AHL, executed at TU Berlin. Based on the SPECTRUM specification of data types and condition event nets of the dynamic processing an integrated specification was produced using algebraic high-level (AHL) nets and net transformations as considered in [Rei 91] and [EPR 94]. In [PRCE 93] the process of a heart catheter examination is specified and the steps from the state-oriented system specification to the formal requirements specification is modelled and it is shown

how to apply structuring and transformation techniques, concurrency properties and compatibility results from the theory of AHL-nets [PER 93].

This case study demonstrates well the suitability of theoretical concepts and results, already in the early phases of the software life cycle model.

The experience around HDMS shows both advantages and difficulties of formal methods in software development and hints at ways of further research and at the same time teaches the limitations of formal methods in regard to the overall task of software development.

5.5 Expectations for the Future

In [CGR 93] a report is given on an international survey of industrial applications of formal methods, which was provided by the National Institute of Science and Technology (NIST) in the United States.

In the NIST survey twelve projects have been analysed using different formal specification techniques. The main conclusions are the following:

1. Formal methods are maturing, slowly but steadily from small case studies to systems of significant scale and importance.
2. The primary uses of formal methods, as shown in the case studies, are re-engineering existing systems.
3. Tool support, while necessary for the full industrialization process, has been found neither necessary nor sufficient for the successful application of formal methods in the case studies. But for the future a software development tool suite is needed.
4. The current educational base in the U.S. is weak in teaching formal methods for software development but several organizations have formal methods technology transfer efforts in progress. Added emphasis on developing notations more suitable to use by individuals not experts in formal methods or mathematical logic is required.
5. There is a clear need for improved integration of formal methods techniques with other software engineering practices and computing science trends, such as visualization, multimedia, object-oriented programming and CASE.

These observations fit very well to the analysis and report in this paper. They also hint at the future direction of research and development in formal methods. But the focus here is more closely on the use of formal specification methods and not on the wider topic of theory and practice in software development. It seems that additional requirements must be stated if this wider spectrum is taken into consideration:

Software configuration and reusability create new types of questions which, as it seems, are in principle tractable by theoretical considerations and formal models. The abstraction techniques of modularization and object-orientation seem suitable to apply. But the practical problems of software migration and re-engineering are major problems at the code level and are highly dependent on the system and application context so that techniques bound to particular specification formalisms do not suffice.

The trend to general concepts of modularization and encapsulation of behaviour in object-orientation is at the same time producing an uncontrolled variety of variants which differ only slightly, but have large effects in practice. Ways out of this diversity have to be found if the requirements of todays software development can be met.

In the light of open systems which today gain world wide influence, standardization is necessary in both, the way to look at the systems and the interfaces for systems interconnection. Powerful standardization organizations have developed recommendations and frameworks which, sometimes even formally, prescribe components, features and architectures of systems, like ISO, SGML or ODP. There is no sufficient theoretical treatment of these models which evolve at the

borderline of established research communities but are widely considered in practical software development.

We hope that the TAPSOFT conferences in the next decade will be able to meet these expectations for the future.

6 References

[AA 82] Agerwala, T., Arwind, C.: Data Flow Systems. Computer 15,2 (1982)

[AR 87] E. Astesiano and G. Reggio: SMoLCS-driven concurrent calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, eds. Proc TAPSOFT'87, Vol 1, no 249, LNCS, 1987, pp. 169-201, Springer Verlag Berlin

[Ba 78] Backus, J.: Can programming be liberated from the von-Neumann style?, a functional style and its algebra of programs. Communication of the ACM 21,8 (1978)

[BEP 87] Blum, E.K.; Ehrig, H.; Parisi-Presicce, F.: Algebraic Specification of Modules and Their Basic Interconnections, JCSS 34,2/3 (1987), 293-339

[BJ 94] Broy, M., Jähnichen, S. (eds.): KORSO: Correct Software by Formal Methods Draft Version, Univ. Bremen 1994, to appear in Springer LNCS

[BKLOS 91] Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., Sannella, D. (eds.): Algebraic System Specification and Development: A Survey and Annotated Bibliography, Springer LNCS 501, 1991

[BW 82] Broy, M.; Wirsing, M.: Partial abstract data types. Acta Informatica 18 (1982), 47-64

[CELMR 84] Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Rossi, F.: An Event Structure Semantics for Safe Graph Grammars. Proc. PROCOMET'94, IFIP TC2 Working Conf., San Miniato 1994, 412-439

[CEW 93] Claßen, I., Ehrig, H., Wolz, D.: Algebraic Specification Techniques and Languages for Software Development - The ACT Approach. World Scientific Pub. 1993

[CGR 93] Craigen, D., Gerhart, S., Ralston, T.: An International Survey of Industrial Applications of Formal Methods, National Institute of Science and Technology, US Dept. of Commerce, NIST GCR 93/626, 1993

[CHL 94] Cornelius, F., Hußmann, H., Löwe, M.: The KORSO Case Study for Software Engineering with Formal Methods: A Medical Information System. In [BJ 94]

[CLWW 94] Claßen I., Löwe, M., Waßerroth, S., Wortmann, J.: Static and Dynamic Semantics of Entity-Relationship Models Based on Algebraic Methods. Proc. GI-Fachgespräche, Hamburg 1994

[Co 71] Cook, S.: The complexity of theorem-proving. Proc. 3rd ACM Symposium on Theory of Computing (1971)

[Col 70] Colmerauer, A.: Les systems-Q on une formalisme pour analyser et synthetise des phrases sur ordinateur. Internal Report 43, Department d'Informatique, Université de Montreal, Canada (1970)

[DFG+94] Didrich, K., Fett, A., Gerke, C., Grieskamp, W., Pepper, P.: OPAL: Design and Implementation of an Algebraic Programming Language. Proc. Conf. Progr. Lang. and Syst. Architecture 1994

[DHP 91] Dimitrovici, C., Hummert, U., Petrucci, L.: Composition and net properties of algebraic high-level nets. In Advances of Petri Nets, vol. 483 LNCS, Springer Berlin 1991

[DN 66] Dahl, O.-J., Nygaard, K.: Simula - An Algol-based simulation language. Communications of the ACM 9,9 (1966)

[DP 89] van Diepen, N., Partsch, H.: Some aspects of formalizing informal requirements. Proc. METEOR Workshop, Mierlo, 1989

[EB 94] Ehrig, H., Bardohl, R.: Specification Techniques Using Dynamic Abstract Data Types and Application to Shipping Software. Proc. Int. Workshop on Advanced in Software Technology, Shanghai Workshop 1994

[EBCO 91] Ehrig, H., Baldamus, M., Cornelius, F., Orejas, F.: Theory of Algebraic Module Specifications Including Behavioural Semantics, Constraints and Aspects of Generalized Morphisms. Proc. 2nd AMAST Conf., Iowa (U.S.A.) 1991

[EE 94] Ehrig, H., Engels, G.: Towards a Module Concept for Graph Transformation Systems: The Software Engineering Perspective. Proc. Graph Grammar Workshop, Mallorca 1994, to appear

[EG 94] Ehrig, H., Große-Rhode, M.: Functorial Theory of Parameterized Specifications in a General Specification Framework, accepted for TCS 1994

[EHKP 91] Ehrig, H., A. Habel, H.-J. Kreowski, F. Parisi-Presicce: From Graph Grammars to High-Level Replacement Systems, Proc. 4th Int. Workshop on Graph Grammars and Application to Computer Science, Springer LNCS 532 (1991) pp. 269-291

[Ehr 79] Ehrig, H.: Introduction to the algebraic theory of graph grammars (A Survey) in: Graph Grammars and Their Application to Computer Science and Biology, Springer LNCS 73, (1979), 1-69

[Ehr 85] Ehrig, H.: Introduction. Springer LNCS 185 (1985), 1-3

[ELO 94/95] Ehrig, H., Löwe, M., Orejas, F.: Dynamic Abstract Data Types Based on Algebraic Graph Transformations. Proc. ADT-COMPASS Workshop 1994, to appear in Springer LNCS 1995

[EM 85] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer (1985)

[EM 90] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. EATCS Monographs on Theoretical Computer Science, Vol. 21, Springer-Verlag (1990)

[EO 94] Ehrig, H., Orejas, F.: Dynamic Abstract Data Types: An Informal Proposal. Bull. EATCS 53 (1994), 162-169

[EPR 94] Ehrig, H., Padberg, J., Ribeiro, L.: Algebraic High-Level Nets: Petri nets revisited. Springer LNCS 785 (1994), 188-206

[Fey 88] Fey, W.: Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language, Diss. TU Berlin, 1988

[FGGP] Fett, A., Gierke, C., Grieskamp, W., Pepper, P.: Algebraic Programming in OPAL, Bull. EATCS 50 (1993), 171-181

[FHMO 91] Fleck, E., Hansen, H., Mahr, B., Oswald, H.: Systementwicklung für die Integration und Kommunikation von Patientendaten und -dokumenten. Forschungsbericht 02-91, PMI am DHZB, 1991

[Fl 94] Fleck, E.(ed.): Open Systems in Medicine, IOS Press, 1994

[Flo 85] Floyd, Ch.: Introduction On the Relevance of Formal Methods to Software Development, Springer LNCS 186 (1986), 1-11

[GB 84] Goguen, J.A.; Burstall, R.M.: Introducing institutions. Proc. Logics of Programming Workshop, Carnegie-Mellon. LNCS 164, Springer (1984), 221-256

[GG 89] Garland, S.J., Guttag, J.V.: An overview of LP, the Larch Prover. Proc. 3rd Conf. on Rewriting Techniques and Applications, Chapel Hill, North Carolina, Springer LNCS 355, 137-151 (1989)

[GHM 87] Geser, A., Hußmann, H., Mueck, A.: A compiler for a class of conditional rewrite systems. Proc. Int. Workshop on Conditional Term Rewriting, Orsay. Springer LNCS 308, 84-90 (1987)

[GM 87] Goguen, J. A., Meseguer, J.: Models and Equality for Logic Programming, Springer LNCS 250 (1987), 1-22

[GR 83] Goldberg, A., Robson, D.: Smalltalk 80: The language and its implementation. Addison-Wesley (1983)

[GTW 76] Goguen, J.A.; Thatcher, J.W.; Wagner, E.G.: 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 (R. Yeh, ed.), Prentice Hall (1978), 80-144

[Ho 72] Hoare, C.A.R.: Proof of correctness of data representation. Acta Informatica 1 (1972)

[Ho 78] Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21,8 (1978)

[Hod 94] Hodges, W.: The Meaning of Specifications I: Domains and initial models, accepted for TCS 1994

[Hum 89] Hummert, U.: Algebraische High-Level Netze. PhD thesis, TU Berlin, Dept. of Comp. Sci., 1989

[Ken 87] Kennaway, R.: On "on graph rewriting", TCS 52 (1987), 37-58

[Ko 79] Kowalski, R.: Logic for problem solving. North Holland (1979)

[Kri 90] Krieg-Brückner, B.: PROgram development by SPECification and TRAnsformation. Technique et Science Informatiques. Special Issue on Advanced Software Engineering in ESPRIT (1990)

[Ku 94] Kutsche, R.: An Application-Oriented Object Model and Concepts of its Formal Foundation, PhD Thesis, TU Berlin, 1994

[LEFJ 91] Löwe, M., Ehrig, H., Fey, W., Jacobs, D.: On the Relationship Between Algebraic Module Specifications and Program Modules, Springer LNCS 494 (1991), 83-98

[LKW 93] Löwe, M., Korff, M., Wagner, A.: An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep et. al. eds. Term Graph Rewriting: Theory and Practice, Wiley 1993, 185-199

[LOTOS 88] Brinksma, E.(ed.): Information processing systems - open systems interconnection. LOTOS: a formal description technique based on the temporal ordering of observational behaviour, International Standard, ISO 8807 (1988)

[Löw 93] Löwe, M.: An Algebraic Approach to Single-Pushout Graph Transformations. TCS 109 (1993), 181-224

[Ma 93] Mahr, B.: Applications of Type Theory. In Proc. of TAPSOFT'93, LNCS 668, pp. 343-355, Springer Verlag, 1993

[Mil 80] Milner, R.: A Calculus of Communicating Systems. Springer LNCS Vol. 92 (1980)

[MM 82/84] Mahr, B.; Makowski, J.A.: Characterizing specification languages which admit initial semantics. Technion Techn. Report 232, Haifa 1982, and TCS 31 (1984), 49-59

[MM 94] Marti-Oliet, N., Meseguer, J.: General Logics and Logical Frameworks. In: What is a Logical System? (D.M. Gabbay ed.), Oxford University Press, 1994

[Mos 89] Mosses, P.D.: Unified Algebras and Institutions. LICS'89, Proc. 4th Ann. Symp. on Logic in Comp. Sci., IEEE, 1989, 304-312

[MTW 88] Möller, B., Tarlecki, A., Wirsing, M.: Algebraic specifications of reachable higher-order algebras. Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types, Gullane, Scotland. Springer LNCS 332, 154-169 (1988)

[Nic 93] Nickel, F.: Ablaufspezifikation durch Datenflußmodellierung und stromverarbeitende Funktionen, Techn. Report TUM-9334, TU München 1993

[Pa 72] Parnas, D.C.: A technique for software module specification with examples. Communications of the ACM 15,5 (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM 15,12 (1972)

[PER 93] Padberg, J., Ehrig, H., Ribeiro, L.: Algebraic high-level net-transformation systems. Techn. Report No. 93-12, TU Berlin, 1993; revised version in MSCS

[PRCE 93] Padberg, J., Ribeiro, L., Cornelius, F., Ehrig, H.: Formal Requirements Analysis Using Algebraic High-Level Nets and Transformations. Techn. Report TU Berlin No. 93-34 (1993)

[PW 94] Pepper, P., Wirsing, M. (eds.): KORSO: A Methodology for the Development of Correct Software, in [BJ 94]

[Rao 84] Raoult, J.C.: On Graph Rewriting. TCS 32 (1984), 1-24

[Rei 81] Reichel, H.: Behavioural equivalence - a unifying concept for initial and final specification methods. In Proc. 3rd Hungarian Comp. Sci. Conf., Budapest, pp. 27-39, 1981

[Rei 91] Reisig, W.: Petri nets and algebraic specifications. TCS 80:1-34, 1991

[St 69] Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13 (1969)

[ST 86] Sannella, D.T.; Tarlecki, A.: Extended ML: an institution-independent framework for formal program development. Proc. Workshop on Category Theory and Comp. Programming, Guildford. LNCS 240, Springer (1986), 364-389

[ST 87] Sannella, D.T.; Tarlecki, A.: Toward formal development of programs from algebraic specifications: implementations revisited. Extended abstract in: Proc. Joint Conf. on Theory and Practice of Software Development, Pisa. LNCS 249, Springer (1987), 96-110; full version to appear in Acta Informatica

[Vau 87] Vautherin, J.: Parallel specification with coloured Petri nets and algebraic data types. In Proc. of the 7th Europ. Workshop on Appl. and Theory of Petri Nets, 5-23, Oxford, 1987