Algebraic System Specification and Development: Survey and Annotated Bibliography

- Second Edition -

— Draft (19 August 1997) —

Maura Cerioli, Martin Gogolla, Hélène Kirchner
Bernd Krieg-Brückner, Zhenyu Qian, Markus Wolf (Eds.)

Editors:
Maura Cerioli, Università degli Studi di Genova
Martin Gogolla, Universität Bremen
Hélène Kirchner, CRIN-CNRS and INRIA-Lorraine
Bernd Krieg-Brückner, Universität Bremen
Zhenyu Qian, Universität Bremen
Markus Wolf, Universität des Saarlandes

Contact address:
Prof. Dr. Bernd Krieg-Brückner
Bremer Institute for Safe Systems
FB3 Mathematik und Informatik
Universität Bremen
Postfach 330 440
D-28334 Bremen, Germany

Contents

Preface

Methods for the algebraic specification of abstract data types were proposed in the early seventies in the USA and Canada and became a major research issue in Europe shortly afterwards. Since then the algebraic approach has come to play a central role in research on formal program specification and development, as its range of applications was extended to the specification of complete software systems, to the formal description of the program development process, and to the uniform definition of syntax and semantics of programming languages. Today this approach extends beyond just software to the development of integrated hardware and software systems. These flourishing activities in the area of algebraic specifications have led to an abundance of approaches, theories and concepts, which have universal algebra, category theory and logic as a common mathematical basis.

The present volume is an annotated bibliography which attempts to provide an up-to-date overview of past and present work on algebraic specification. No attempt is made to provide a coherent introduction to the topic for beginners; the intention is rather to provide a guide to the current literature for researchers in algebraic specification and neighbouring fields. Some indications of how the different approaches are related are included.

This volume arose out of the work of the COMPASS Basic Research Working Group, funded from 1989 to 1996 by the European Community under the Basic Research Action programme, ref. no. 3264 and 6112 (see [584, 589, 586]) and coordinated by Bernd Krieg-Brückner of Universität Bremen. The name COMPASS stands for "a COMPrehensive Algebraic approach to System Specification and development". The introductory chapter in this volume includes a brief summary of the history of COMPASS.

An early version of this document was included in the original COMPASS project proposal as a review of the state of the art [591]. As a result of suggestions for improvements from the participants in the COMPASS Working Group, it was edited into a more or less coherent form and appeared as [114]. The current version is an update of this Annotated Bibliography that reflects the considerable volume of publications in the last five years.

Since the COMPASS Working Group includes most of the leading European experts in algebraic specification, the result should be a relatively comprehensive overview of the main work in the field. But because of the likely (sometimes perhaps even subconscious) focus on the COMPASS Working Group, there are inevitable gaps and inaccuracies. We apologize for these in advance, and in particular to anybody whose work has inadvertently been omitted. In spite of its deficiencies, we hope that it nevertheless represents a useful snapshot of the current state of the art.

Acknowledgements

We thank those members of the COMPASS Working Group who contributed to this document and the group of so-called Scientific Correspondents who added to the success of the Working Group, their contribution should become apparent from the publications. Special thanks go to Detlef Plump and Zhenyu Qian of Universität Bremen for their efforts in administrating the COMPASS Working Group.

This Book

This book is an annotated bibliography which attempts to provide an up-to-date overview of most past and present work on algebraic specification for researchers in algebraic specification and neighbouring fields. It extends the previous annotated bibliography by Bidoit, Kreowski, Lescanne, Orejas and Sannella [114]. A review of the main topics of current and past research is given, with some indication of how the different approaches are related and pointers to relevant papers.

The term "algebraic specification" has been used by different people at different times to refer to different things. Originally it referred to the use of (many-sorted) algebras to model programs, and the use of equational axioms to write specifications. This was sufficient to handle simple specifications of total first-order functions, but not much more. Nowadays, more complicated notions of algebra are used to take account of partial functions, higher-order functions, etc., and more sophisticated logical systems are used to write axioms describing such algebras. Although the use of the label "algebraic" no longer seems appropriate in this context, it is still widely used. Probably the feature which unifies all approaches under this banner is the use of algebra-like models of some kind in order to concentrate on the functional behaviour of software systems and the representation of data. Specifications describe such models via the use of axioms in some logical system which normally includes equality as a primitive.

No attempt is made here to provide an introduction to algebraic specifications. For such an introduction, the reader is referred to any one of the textbooks and monographs which have appeared during the past few years. In order of publication, these include [557], [328], [813], [748], [291], [506], [329] and [763]. Several collections of papers on the topic have been published, including [573, 847, 317, 99, 331, 37, 469, 945, 68]. A early annotated bibliography of papers on algebraic specification is [599].

The field of algebraic specification is divided here into five main topics. Any such division is difficult; there is inevitably significant overlap between parts.

Chapter 1: Basic Concepts

A discussion which arose early in the history of work on algebraic specifications concerned the choice of semantics of a specification. In the context of purely equational (or conditional equational) specifications, the obvious choice of taking the class of all algebras which are models of the axioms is ruled out since this will always admit undesirable trivial models. An early suggestion was to restrict attention to the *initial* object(s) in this category of models; another possibility is to restrict attention to the *terminal* object(s) in a certain sub-category of models; other choices are also possible.

Any approach to algebraic specification must be based on some notion of algebra and some choice of logical system in which to write axioms. Early approaches were based on many-sorted algebras and equational axioms. Soon it became clear that this is not always flexible enough. Some data types and functions can only be described in an awkward fashion using equational logic, and there are some cases where equations are not powerful enough. It is apparent that more complicated logical systems and more complicated notions of algebra are needed to cope with "awkward" aspects of programs such as: types, partial and non-strict functions; higher-order functions and infinite objects; non-deterministic functions and concurrent systems. A great deal of work has been devoted to enriching the basic algebraic specification framework to deal with these complications. It turns out that much of the work on algebraic specifications that has been done is actually independent of the notion of algebra and the logic used to write axioms: it is possible to formulate many results in the context of an arbitrary *institution*.

Chapter 2: Structuring Concepts

As soon as non-trivial examples are considered, it becomes obvious that large specifications are unmanageable unless they are built in a structured fashion from smaller specifications using *specification-building operations* of various kinds and mechanisms for *parameterization* and *modularization* of specifications. The decision to structure specifications also makes it necessary to combine several logics in a common framework.

Chapter 3: Specific Paradigms

There are relationships between work on algebraic specification and work on functional, logic and object-oriented programming, states and concurrent and reactive systems. First, programming languages can be considered as executable specifications; second, algebraic specifications may need to be eventually implemented by programs; finally, algebraic specification methods can be used to specify the semantics of programming languages, provided that algebras are used which are capable of successfully modelling the features present in the programming language at hand (non-termination, concurrency, etc. The chapter concludes with a review of specification languages.

Chapter 4: Methodology

The interest in specifications stems largely from their intended role in the formal program development process whereby programs are evolved from specifications by means of successive refinement or *implementation* steps. With many different choices of specification approach (initial/final models, with/without parameterization, with/without constraints, etc.) the question of the relative expressive power of the different approaches arises.

Work on algebraic specifications aims to provide conceptual and methodological support for all aspects of the software development process. An early phase of the software life-cycle is the construction of the requirement specification, which lists the properties which the desired system is to satisfy. An important problem is how to translate the often vague wishes of the customer into a formal requirement specification: this is the topic of *requirements engineering*. Once a specification is available, various methods are available for *transforming* it into an equivalent specification or into an executable program which is correct by construction. There are relations between these methods and program synthesis and the “proofs as programs” paradigm. One of the advantages of formally developing programs from specifications is the possibility for correct *reuse* of previously-developed program components based on their specifications.

Chapter 5: Tools

The use of algebraic specifications in practical software development demands a range of machine-based tools to support various software development activities. A number of integrated specification *support systems* have been produced, aiding the construction of specifications and checking of their properties, and covering certain aspects of the development of programs from specifications. When specifications include equational or conditional equational axioms only, it is possible under certain conditions to use them directly as high-level *prototypes*, running test cases in order to check that the specification describes what is intended. An obviously important activity in connection with formal specifications of any kind is *proving theorems* in the context of a specification. Important sub-topics concern *proofs of termination* of rewrite systems using sophisticated orderings on the set of terms, and Knuth-Bendix-style *completion* of term rewriting systems, constraint solving, etc.

Chapter 1

Seven Years of COMPASS

(An earlier version of this chapter has appeared in [586].)

1.1 Historical Development

The systematic investigation of algebraic specification methods in the context of computer science started in the early seventies in the USA and Canada. In the late seventies, the topic also became a major research issue in Europe. Since then, the number of researchers in this area has grown considerably, and the algebraic approach has come to play a central role in research on formal program specification and development, as its range of applications was extended to the specification of complete software systems, to the formal description of the program development process, and to the uniform definition of syntax and semantics of programming languages. Meanwhile this approach extends beyond just software to the development of integrated hardware and software systems.

Today the field of algebraic specification has grown into one of the major areas of research in theoretical computer science. The flourishing activities in this area in the past twenty years have led to an abundance of technically different approaches, theories, and concepts. This seeming diversity is only superficial; all approaches share a common framework and mathematical basis.

1.1.1 The Working Group

COMPASS (a COMPrehensive Algebraic approach to System Specification and development) was funded as an ESPRIT Basic Research Working Group 3264 from April 1989 (COMPASS 1), and, as a continuation, as Working Group 6112 until March 1996 (COMPASS 2).

The COMPASS Working Group originally consisted of 11 Partner organisations with 18 team leaders and about 90 scientists; in the second phase it was extended to 19 Partners, see Fig. 1.1 (only team leaders are listed; the numbers indicate partnership in COMPASS 1 and 2, resp.). In addition to members of the partner organisations, a varying number of so-called *Scientific Correspondents* received funding for travel to COMPASS Workshops, joint meetings and visits.

In effect, the COMPASS Working Group has brought together most European scientists in the area of algebraic specification methods in an effort to consolidate and integrate the theoretical basis and apply it to software technology.

1.2 Aims and Objectives

1.2.1 Original Aims

The potential effects and ramifications of the originally stated aims (quoted from [591, 584] in this section) are as important now as they were at the beginning of COMPASS:

- to provide a comprehensive algebraic approach to the specification of systems and their components,
- to provide a formal basis for their correct development,

AARHUS UNIVERSITET, DK (2)
Peter Mosses
UNIVERSITAT POLITECNICA DE CATALUNYA, BARCELONA, E (1, 2)
Fernando Orejas
TECHNISCHE UNIVERSITÄT BERLIN, D (1, 2)
Hartmut Ehrig, Peter Pepper
TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG, D (1, 2)
Hans-Dieter Ehrich, Martin Gogolla [a]
UNIVERSITÄT BREMEN, D (1, 2)
Bernd Krieg-Brückner (Coordinator), Hans-Jörg Kreowski
TECHNISCHE UNIVERSITÄT DRESDEN, D (2)
Horst Reichel
UNIVERSITY OF EDINBURGH, UK (1, 2)
Rod Burstall, Don Sannella
DIP. MAT., UNIVERSITÁ DI GENOVA, I (1, 2)
Egidio Astesiano
UNIVERSITÁ DI ROMA, I (2) [b]
Francesco Parisi-Presicce
INESC, LISBOA, P (2)
Amilcar Sernadas
TECHNISCHE UNIVERSITÄT MÜNCHEN, D (1, 2) [c]
Manfred Broy
CRIN, NANCY, F (1, 2)
Pierre Lescanne, Hélène Kirchner
B-FAC., UNIVERSITY OF NIJMEGEN, NL (1)
Helmut Partsch
IFI, UNIVERSITY OF OSLO, N (2)
Ole-Johan Dahl, Olaf Owe, Sigurd Meldal [d]
OXFORD UNIVERSITY, UK (2)
Joseph Goguen
LRI-URA 410, UNIVERSITÉ DE PARIS SUD, F (1, 2)
Marie-Claude Gaudel, Jean-Pierre Jouannaud
LIENS, ÉCOLE NORMALE SUPÉRIEURE, PARIS, F (2)
Michel Bidoit
FORWISS, LMU MÜNCHEN, D (1, 2) [c]
Martin Wirsing
MPI INFORMATIK, SAARBRÜCKEN, D (1, 2) [e]
Harald Ganzinger
INST. OF COMPUTER SCIENCE, PAS, WARSAW, PL (2) [f]
Andrzej Tarlecki

[a] now at Universität Bremen
[b] formerly Universitá degli Studi de Aquila
[c] formerly Universität Passau
[d] University of Bergen, local partner site in Norway
[e] formerly Universität Dortmund
[f] associated

Figure 1.1: The COMPASS Working Group

- to consolidate the theoretical background,
- to lay the basis for an increased power of support tools,
- to encompass new programming methodologies and application areas, and
- to make progress towards the development of a uniform mathematical framework for logic and semantics within computer science.

An Industry of Reusable and Interchangeable Software Components will have significant advantages (and is perhaps only possible) with formal methods, i.e. if the meaning of a component is precisely specified and the implementation can be relied on to be correct. It should be possible to understand and use a system component only on the basis of its interface, making it reusable. For the implementor, it is equally important to be able to hide the body (e.g. for proprietary reasons), and to replace it when the interface is not affected. Generic reusable system components avoid duplication of development efforts, ease prototyping and speed up system construction.

Formal Specification and Development. It is common practice in software technology that only the syntactic aspects of interfaces are specified, i.e. the data types and operations that the component provides. User and implementor have to rely on natural language comments for information about their meaning. This poses a severe danger of misinterpretation, inconsistency and incompleteness; there is no way to increase confidence by any means of verification other than inspection of code. This violation of the information hiding principle makes system improvements and evolution extremely cumbersome and error-prone, if not impossible.

The algebraic approach supports the *precise specification* of the semantic aspects of interfaces, providing the conceptual basis for their use and the theoretical foundations for the formal development of *correct* system components from their specifications. Thus it covers the whole software development process from the specification of requirements to the finished system.

Loose Specification of Requirements. It is highly desirable to be able to specify only the *necessary* properties of a component or system in a requirements specification. This is the interface seen by the user that serves as a guideline for the implementor. It should be as precise as necessary, but also as *loose* as possible: this avoids over-specification of details, leaves *freedom for different design decisions* by the implementor, and is therefore likely to decrease cost and increase efficiency.

Independence of Design. The particular design implementing the interface should remain hidden. The hiding principle eases replacement of components at the development stage, for example of a prototype component by a final implementation, and during system evolution when requirements change.

Purely model-oriented approaches, specifying the meaning of a component by giving a very abstract implementation (for example by sets, or by lists in a functional programming language) may give away more detail about the particular design than desirable since they specify a particular model. Transition to another design may be very difficult since a specification of the abstract properties common to both is not available; in fact, such a *property-oriented specification,* which is loose enough to characterize a class of distinct models, has to be constructed in this situation.

Stepwise Development. Thus a high level of abstraction from subsequently introduced design details can be achieved. Perhaps most importantly, the refinement relations between a specification and a sequence of more detailed designs, that is the *correctness* of the implementation with respect to the interface, can be *formally verified.*

The COMPASS Working Group investigates the stepwise development of more detailed and more efficient versions, supported by semi-automatic tools, such that the overall correctness proof is broken down into the verification of correctness at each step. It is a challenge to formalize the development process itself as a basis for a library of reusable development methods.

Impact and Potential. Government institutions and private industry are starting to require adherence to formal development methodologies in security and safety critical areas. The economic impact of increased *reliability*, *productivity* through reuse, and the potential for early prototyping is increasingly recognized.

1.2.2 Original Objectives

The main goals of the COMPASS Working Group are the further investigation and consolidation of the algebraic approach to system specification and development in a comprehensive way. There are considerable needs for clarification, unification, extension and integration of other programming and specification paradigms, in particular, the relationship between the algebraic approach and other formal approaches. The hope is to bridge the gap between the state of the art of the algebraic approach and the practical needs of the specification and formal development of system components.

The COMPASS Working Group brings together the leading experts in the area of (algebraic) system specification and development in Europe. It is expected that its members will continue to have a major influence in the area of formal methods on ESPRIT projects, ESPRIT Basic Research projects and working groups, and national projects. The increased cooperation will lead to further cohesion, consolidation of approaches, and, in some cases, perhaps even some form of quasi-standardisation.

1.3 Actions

Apart from many individual joint meetings on specialized topics and visits, there have been business meetings adjoining major conferences at Barcelona, E (1989), Brighton, UK (1991), Orsay, F (1993), Aarhus, DK (1995) and Oxford, UK (1996), COMPASS Workshops at Bremen, D (1989), Nancy, F (1990), Bastei, D (1993), Sintra, P (1994), and International Workshops on Abstract Data Types, jointly organized with the COMPASS WG, at Wusterhausen, D (1990, [317]), Dourdan, F (1991, [99]), Caldes de Malavella, E (1992, [331]), Sta. Margherita, I (1994, [37]), and Oslo, N (1995, [469]).

The primary areas of research were:

1. Foundations
1.1. Models and Logics
1.2. Structure and refinement
2. Development Concepts
2.1. Methodology
2.2. Specification Languages
2.3. Concepts for Environments and Tools
2.4. Theorem Proving
2.5. Term Rewriting
3. Applications and Connections to Other Areas
3.1. Software Development Process
3.2. Programming Languages
3.3. Special Applications

Object orientation and concurrency, and structuring of specifications and proofs, received a certain prominence.

1.4 Results

Algebraic techniques, stimulated by COMPASS members, have already played a dominant role in several national and international research projects, among them various ESPRIT projects such as DRAGON, FOR-ME-TOO, GRASPIN, LOTOSPHERE, METEOR, PEACOCK, PROSPECTRA, RAISE, and SEDOS.

During the first phase of COMPASS [589], over 300 research papers and an *Annotated Bibliography* [114] were published by group members (not counting those of associated Scientific Correspondents), more than 20% by joint authorship from different teams. More are expected for the second phase; the final count is available from the update of the Annotated Bibliography [589] in this volume. A continuing source of information is the series of *ADT/*COMPASS *Workshop Proceedings* in Springer Lecture Notes in Computer Science [317, 99, 331, 37, 469].

Another major outcome of COMPASS will be a comprehensive volume on *Algebraic Foundations of System Specification* [27] (with 14 chapters and a total of about 500 pages), written jointly by 20 authors from the COMPASS WG (with 2 exceptions) and developed jointly with IFIP WG 14.3. Indeed, this latter international Working Group ("Foundations of System Specification") has been started by members from COMPASS, includes many of the team leaders and will provide some form of continuation of COMPASS.

Consolidation of different approaches and extensions into new areas have been attempted in several focal areas, with emphasis on the structure of systems (modularisation, reusability), structure of developments and proofs, and concurrency. Other fruitful areas of work include logical foundations, development concepts (methodology, specification languages), environments and tools (prototyping, theorem proving, term rewriting), and applications.

The significant efforts to move away from individualistic research efforts (occasionally even contradicting each other) to a coherent basis, but with a lot of diversity, deserve particular emphasis. The survey papers written by COMPASS members, often by co-autors from several groups (cf. those invited at WADT/COMPASS workshops in [317, 99, 331, 37, 469]), have dramatically clarified the state of the art. During the life of COMPASS, new cooperations between members were established, producing indeed much more coherent views on some topics (e.g. observability). In many such areas, nobody in the field today will consider the topic without starting from the results that have been obtained in cooperation, hence COMPASS was successful in clarification and extension of several existing directions of research.

Group members have also significantly contributed to the development of specification languages and methodologies for development, and also semi-automatic tools and development environments based on algebraic specifications, e.g. the ESPRIT project PROSPECTRA, in which 6 COMPASS members have been involved; the ASSPEGIQUE system has been used in METEOR; the ACT set of tools is integrated into the systems developed in SEDOS and LOTOSPHERE.

It should be emphasized that work under the auspices and cooperation of the COMPASS community has provided a very productive forum for the work of numerous younger scientists, in particular PhD students; the educational aspect should not be underestimated.

1.5 Common Framework Initiative

In the original proposal [591] it was envisaged to have an R&D project in the third phase of ESPRIT for a Uniform European Algebraic Language and Environment for System Specification and Development with an associated formal semantics and development methodology.

In fact, such a collaborative effort, the so-called Common Framework Initiative (CoFI[1]), was started some time ago by the COMPASS WG and IFIP WG 14.3 and has been in full swing since Autumn 1995, with the goal of having a first result in Autumn 1996. To have most impact, it is intended as an open international effort, driven by the European COMPASS community but involving people beyond it. Some 25 leading researchers in algebraic specification have already agreed to participate, and others are being invited to join.

The Common Framework is perceived as a major chance for integration and standardization, indeed the very basis for relating existing methods and tools to overcome the apparent gap between theoryand industrial application that still exists. For the COMPASS community, there is agreement that only with a common framework, where language and logics serve as a vehicle for communication, will new integrated environments be developed in which specialized methods and tools can coexist; only with such tools and environments, first as public domain prototypes, later in industrial-level versions, will industry be convinced to apply formal methods on a larger scale, whether in-house or with outside help.

1.5.1 Aims and Scope of CoFI

Quoting from the "Charter" (see [238]) of the *Common Framework Initiative for Algebraic Specification:*

The rationale behind this initiative is that the lack of such a framework greatly hinders the dissemination and application of research results. In particular, the proliferation of specification languages, some differing in only quite minor ways from each other, is a considerable obstacle for the use of algebraic methods in industrial contexts, making it difficult to exploit standard examples, case studies and training material. A common framework with widespread support throughout the research community is urgently needed.

The current aim is to base the common framework as much as possible on a critical selection of features that have already been explored in various contexts. The collective experience and expertise of the participants provides a unique opportunity to achieve this aim within a short time-span.

The common framework will provide a family of languages at different levels (cf. fig. 2).

The Overall Aims of the initiative are:

[1] CoFI may be pronounced like 'coffee'

```
*-*-*  Extension Languages
 \|/
  *    The Common Language
 /|\
*-*-*  Sub-Languages
```

Figure 1.2: The CoFI Family of Languages

- A Common Framework (CoFI) for algebraic specification and software development is to be designed, developed, and disseminated.
- The production of CoFI is to be a collaborative effort, involving a large number of experts (30-50) from many different groups (20-30) working on algebraic specifications.
- In the short term (e.g., by 1997) CoFI is to become accepted as an appropriate basis for a significant proportion of the research and development in algebraic specification.
- CoFI is to include a uniform family of related specification languages, all with a consistent, user-friendly syntax and straightforward semantics. It is expected that some of the CoFI languages could replace existing specification languages.
- CoFI is to be supported by concise reference manuals, users' guides, libraries of specifications, tools, and educational materials.
- In the longer term, CoFI is to be made attractive for use in industrial contexts.
- CoFI is to be available free of charge, both to academic institutions and to industrial companies. It is to be protected against appropriation.

Functionality. CoFI is to allow and be useful for:

- Algebraic specification of the functional requirements of software systems, for some significant class of software systems.
- Formal development of design specifications from requirements specifications, using some particular methods.
- Documenting the relation between informal statements of requirements and formal specifications.
- Verification of correctness of development steps from (formal) requirements to design specifications.
- Documenting the relation between design specifications and implementations in software.
- Exploration of the (logical) consequences of specifications: e.g., rewriting, theorem-proving, prototyping.
- Reuse of parts of specifications.
- Adjustment of specifications and developments to changes in requirements.
- Providing a library of useful specification modules.
- Providing a workbench of tools supporting the above.

1.5.2 The CoFI Family of Specification Languages

Quite important for the success of CoFI is the coverage of most concepts of existing specification languages on the one hand, but, more importantly, uniformity and simplicity. How can these seemingly conflicting goals both be achieved? The key idea here (and the challenge for the design) is not to have one huge specification language, but to impose some structure such that a family of specification languages emerges. It is to consist of three main levels (cf. Fig. 1.2):

The Common Language (CL). This general-purpose specification language is to be the most prominent of the CoFI family: the other languages are to be obtained by restriction or extension of it. CL should be at least as expressive as various existing languages. Its intended applicability is for specifying abstract data types, and requirements and design for programs (at least for programs which do not react with their environment during execution). It is also to be the main topic of the documentation (reference manual, user's guide, text book).

Some Sublanguages (SLs). These are to be obtained from CL by imposing syntactic and/or semantic restrictions. The sublanguages need not have a common kernel. Their intended applicability is in connection with particular kinds of tools (e.g., a first-order equational SL might support tools for term rewriting and completion).

Some Extended Languages (XLs). These are to be obtained from CL (or perhaps from SLs) by syntactic and/or semantic extensions. The extended languages need not have a common super-language. Their intended applicability is to include advanced features well-adapted to specifying requirements and design of systems involving various programming paradigms (e.g., reactive systems, object-oriented systems).

1.5.3 CoFI Task Groups

Various CoFI Task Groups have been defined, with the following coordinators:

- Coordination: Peter D. Mosses (Aarhus)
- Language: Bernd Krieg-Brückner (Bremen)
- Semantics: Don Sannella (Edinburgh)
- Methodology: Marie-Claude Gaudel (Paris) / Andrzej Tarlecki (Warsaw)
- Tools: Michel Bidoit (Paris)
- Reactive Systems: Egidio Astesiano (Genova)

Language Design and Formal Semantics. The design of the Common Language of the language family will proceed as fast as possible to keep up the momentum. It will be followed by a formal semantics (an absolute necessity for formal methods); here also, the challenge is going to be, how the semantics might be structured, and perhaps how simpler semantics for sublanguages can be derived.

A checklist of language concepts and, based on this, a catalogue of existing specification languages for comparison and easy cross-referencing is being compiled [237]. Language design study notes are being written and the design is well under way.

Methodology. The language design is influenced by methodological considerations. Among the issues to be considered are the role of specifications in the development process (relationship between specifications and programs, formal vs. semi-formal specifications), an abstract view of the development process (refinement and proof obligations, structuring issues, reusability, flexibility and evolution), and techniques for writing specifications of individual system modules.

Introductory material, documentation of development strategies and proof strategies, description of case studies, and libraries of specifications and developments designed for re-use will have to be developed eventually.

Tools. To be used, the language family must be supported by a minimal but widely available set of tools (syntax and static semantics checkers, library support, etc.); to some extent, such considerations influence the design of the language family. A most important aspect is the potential for reuse of an abundance of existing tools. Many of these are specialised, that is only applicable to a particular sublanguage and its associated logic. This raises the issue of multiple logics (see below) and suitable embedding translations, but also of "interchange formats" and means for exchanging libraries of specifications.

Reactive Systems. After the design of the Common Language is in good shape, the design of Extended Languages will proceed in full force. Very prominent among the concerns are language extensions to support state change, reactive systems, and object-orientation.

1.6 Multiple Formalisms and Logics

It has already been mentioned that the issue of integrating tools in a common support environment raises theoretical problems of combining and embedding different logics. The general issue of coexistence, transition between and combination of different paradigms and formalisms is an even deeper research issue:

- for *heterogeneous systems* comprising components developed with different paradigms, e.g. components encapsulating complex data and algorithms, and components for concurrent control and real-time behaviour,

- at *different stages* of the development, e.g. from a property-oriented specification to an executable program, from temporal logic to a CSP program,

- in a variety of *specialised tools,* e.g. for validation prototyping, verification, analysis, or refinement by transformation.

In the particular context of Formal Methods guaranteeing correct software, one universal language or development methodology to cover all problems would lead to unrealistic complexity. It is therefore mandatory to tailor languages, semantics, methodology and tools to specific problem domains to be effective. Then the question is, how to decompose problems and how to combine partial solutions.

A formal framework should be developed, in which various specific paradigms, formalisms, and tools, each as simple as possible for the task at hand, can coexist and be combined in a systematic way. To be able to combine existing software components from a library, to develop heterogeneous systems or to move from one paradigm or tool to another, fundamental problems of semantic consistency and integration, compositionality and interoperability have to be solved.

Logic provides the key for the solution of the semantic aspects of these problems. The semantic core of each specific formalism is mapped to a specific logic. Thus by establishing a firm theoretical basis for relating and combining different logics, a basis for relating and combining formalisms is provided. Specific logics should be related in a Logic Graph. The Logic Graph can be supported by a tool which would enable even non-specialists to naturally navigate to find the formal framework best suiting their needs, e.g. for assembling elementary components or tailoring to the needs of a specialised tool.

The above concerns, both theoretical and practical, have prompted COMPASS members several times over the years to submit various research proposals.

1.7 Conclusion

The COMPASS community is now closer-knit than ever. The theoretical basis has been consolidated (cf. [27]), differences between technically different approaches have been bridged, approaches (and people) have matured.

The funding of COMPASS as a European Working Group has come to an end with the termination of COMPASS 2. Efforts to keep the COMPASS community together, to maintain the immensely fruitful cooperation, have to be reinforced; it would be just too sad to see it disintegrate.

I am personally very happy that initiatives such as the Common Framework Initiative have finally materialized and are taken very seriously by COMPASS members (and others); the network of close contacts and friends that has developed over the years in the COMPASS community is probably an essential ingredient here, not only to convince people that such an effort is essential for the future of the research area, its perception in public, its potential for impact on industrial application and its financial support, but also to actually make things happen. In effect, many colleagues have set aside old (and now almost forgotten) technical differences and are willing to melt their pet specification languages into the family pot of CoFI while significantly capitalizing on their experiences, of course. Under the general coordination of Peter Mosses, international support for the effort is growing. Let us hope the effort will be successful, and that some support from the EU will eventually come to bear.

Similarly, funding in the area of Multiple Formalisms and Logics is needed and would have a very beneficial effect on the community in relating their technical work.

Much more work needs to be done by individual groups and in cooperation to make the vision of integrated and industrially useful support environments based on CoFI a reality; these can then gradually be supplemented by individual tools. One effort in this direction is the UniForM Workbench (Universal Formal Methods Workbench) funded by the German ministry of research BMBF [588].

Tony Hoare, in his lecture given at this last COMPASS meeting/11th ADT conference in Oslo, was kind enough to pat our egos by suggesting that we stick to theory (that we do best, apparently), driven by wonderment, cultivating our genuine curiosity, generating results elegant enough to enjoy and admire; he was exceedingly generous to extend such attributes to some of the results from COMPASS. However, many of us are sufficiently ambitious to attempt to participate in the engineering science of developing languages, methodologies and tools. I wish that we would all get our hands genuinely dirty in the future and have an effect, producing something that is really widely used in practice – and still come out with something simple, elegant and beautiful.

Chapter 2

Basic Concepts

2.1 Specification Semantics

The term “algebraic specification” originally referred to the use of many-sorted algebras to model programs and the use of equational axioms to write specifications with initial algebra semantics. See the pioneering paper [420, 418] by the ADJ group and [392] for a full account of the early days of ADJ. Nowadays, several complicated notions of algebra and sophisticated logical systems are used as well. However, the basic intuition is still the same: some kind of algebra-like models, classified by the language used to describe them, are used as semantics of data types and programs. Moreover they are specified in terms of their properties, in order to concentrate on the functional behavior of software systems, abstracting from the representation of data.

Therefore an algebraic specification framework consists of the collections of symbols used to describe the data types, the possible models giving semantics to those symbols and the axioms, together with a notion of validity relating the possible models to the axioms. Specifications in such a framework describe the intended models via the use of axioms and of an extra-logical notion, stating which (non-necessarily proper) subclass of the algebras satisfying the axioms has to be taken in account.

A number of survey of the field of algebraic specifications have been produced, e.g. [328, 402, 842, 942] In [841], Sannella and Tarlecki provide a summary overview of the model-theoretic foundations of program specification and formal software development, with emphasis on the basic motivations behind the technicalities presented elsewhere rather than on providing new results.

Loose Semantics

The *loose semantics* of a specification is the whole class of models satisfying its axioms. Loose semantics is appropriate for specification and program development [59, 163, 848, 739].

Indeed, in the initial phase of a specification design, many details have to be fixed and only some requirements are already known. Thus, it is interesting to consider the whole class of models, because it represents the possible implementations of the decisions still to be taken and further steps of refinement will restrict the model class.

Moreover, in most frameworks (see e.g. [411, 646, 171, 17, 20]) there are systems for logical deduction that are complete at least for atomic formulae and implications w.r.t. the whole class of models. This, in turn, allows to reason about the properties satisfied by all possible implementations of a requirement specification.

However, if the language is not powerful enough, then among all the models satisfying a given specification there will probably be many non-interesting or degenerate ones that, because of the lack of power of the language, cannot be ruled out. For instance, this is the case when using pure equational, or even conditional equational logic. Thus, to axiomatically exclude at least part of the non-interesting models, in the case of algebraic specifications of relevant data types we need inequations in the premises of conditional equations. [623] investigates the existence of free functor semantics for specifications of first-order total algebras with positive/negative conditional equations. The corresponding results carry over straightforwardly to specifications of first-order partial algebras with positive/negative conditional existence equations.

Most logics used in algebraic settings include an equality symbol that is interpreted, usually, as the identity relation. But allowing it to be interpreted as any congruence, *ultra-loose* semantics (see e.g. [174]) captures all possible implementations as models of a specification.

Initial semantics

The models of interest are taken to be the initial objects in the category of models. The basic reference is [419], which develops the basic theory for total algebras; more recent surveys of the field are [328, 671] and [577] is a tutorial introducing the basic features of algebraic specification with initial semantics, including parameterization and instantiation.

An initial many-sorted total algebra for an equational specification SP is characterized (see [181, 182]) by the fact that all elements of its carriers are the interpretation of a term without variables (*no-junk*) and that it satisfies only those ground equations which hold in all models of SP (*no-confusion*). Because of this characterization, initial algebra semantics is closely related to rewriting (see Section 6.2) and therefore well-suited for rapid prototyping of specifications. For instance term rewriting provides an operational semantics for the constructs supporting initial semantics in algebraic specification languages such as OBJ [422].

Therefore the so-called *no-junk* and *no-confusion* properties are usually preserved also for more exotic algebraic framework, by the choice of homomorphisms between models, aiming at sorting out in a clean categorical way the model (up to isomorphism, that is up to data representation) corresponding to recursive definitions of functions (programs).

Indeed, in the case of total equational specifications initial semantics represents primitive recursion, and in case of equational partial specifications the whole range of recursion.

A very abstract treatment of initial semantics is given in [399], based on work of Diaconescu [277, 278]. These papers present a categorical treatment of equational logic based on forgetful functors from models to 'domains', which generalizes equational logic to cover Horn clause logic and constraint logic programming. Initiality properties are used to prove a Herbrand theorem at a very high level of abstraction and generality.

Final semantics

In the case of final semantics [376, 933], the interesting models are considered to be the terminal objects of a certain sub-category of the category of all the models. Usually such subcategory is chosen in such a way that final algebra semantics represents the "mathematical" semantics of a specification by which the identity of a data value is determined by its behavior in all contexts.

While initiality corresponds to minimal fix-point semantics, final semantics represents maximal fixed-point semantics and allows the use of co-inductive techniques.

The relationships between final semantics and the kind of operational semantics embodied in such structures as transition systems provide an interesting area for future research; [640, 644] give preliminary accounts, and [217, 642] develop these ideas for the object paradigm.

Term-generated Algebra Semantics

The term-generated algebra semantics of a specification [59] (for an overview, see [942]) is considered to be the class of all algebras satisfying the specification and containing no "junk", i.e. every value in the algebra is the result of evaluating a ground term. Initial and final algebras are special "extremal" models of this semantics [153, 159].

Observational and Behavioral Semantics

The basic intuition is that the data types of the models are distinguished in *visible* (or *observable*) and *hidden* (representing internal states) and that the equality between terms of invisible sort cannot be directly stated/checked, but it is deducible from their behavior in all observable contexts.

In the literature one can distinguish two main approaches to the definition of observational semantics for algebraic specifications.

- On one hand the satisfaction relation is modified, in order to get that (given some notion of observation) a formula is satisfied by a model iff all its observable consequences are satisfied in the standard sense (see e.g. [812, 727, 74, 474]).

This notion originates in the work of Reichel [811], primarily in the context of final semantics, and has recently been incorporated in the institution of *hidden sorted algebra*. The hidden sorted approach is prefigured in the 'abstract machines' of [412], and shown to form an institution in [394]. An abstract approach to hidden sorts in arbitrary institutions is given in [180]. The relevance of hidden sorted algebra to the object paradigm is set out in [394, 408], and this work is extended to the order sorted case in [401, 643].

In [644] very general results are stated on the existence of final algebras and a radical new treatment of nondeterminism is proposed. Moreover, a general approach to concurrency via the concurrent interconnection and the relationship with process algebra are illustrated as well, together with the use of coinduction to make proofs.

The relationship between behavioral equivalence, final semantics, and abstract machines is investigated in [640].

- On the other hand the observational semantics of a specification can be defined as the class of all structures that are equivalent to some model of the specification with respect to an observational equivalence relation between algebras, that is usually identifying algebras that are indistinguishable by observable formulae (see e.g. [811, 727, 845, 941, 858]).

In [107, 113] an analysis and a comparative study of the two different approaches is presented in a more general framework which subsumes the observational case. Thereby the distinction between the different observational concepts is reflected by the notions of behavioral specification and abstractor specification. [113] provides necessary and sufficient conditions for the semantical equivalence of both kinds of specifications and shows that behavioral specifications can be characterized by an abstractor construction and vice versa. These results are generalized to higher-order logic in [501] and to an even more general setting based on institutions in [115].

Constraints

In some approaches, restrictions to the class of models of a given specification are given explicitly using so-called *constraints*. A constraint is a semantic construction "fixing" or "restricting" the interpretation of a specification contained within a larger one. As a consequence, the semantics of this larger specification is also restricted accordingly.

Constraints were first defined and used by Reichel [810] for giving semantics to *canons*. In his approach, a constraint designates a part of a specification that is to be interpreted initially (or to be more accurate, freely). This kind of constraint has been called a *data* or *free generating* constraint. They were the basis of the semantic definition of CLEAR [184] and of other specification languages [334].

More recently, in [815] the combined use of initial and terminal constraints for the specification of data types on one hand and of behavior classes on the other hand has been suggested. Till now terminal and initial constraints have been used alternatively within algebraic specifications. But in the algebraically more general setting of the simultaneous use of algebras and co–algebras, which may be subsumed by Di–algebras [452], initial constraints and terminal constraints supplementary and allow in a uniform manner the algebraic specification of data types, of behavior classes and of dynamic systems. The application of the combined use of initial and terminal constraints on Di–algebras is subject of future research in Dresden.

In [850], Sannella and Wirsing proposed a different kind of constraint, called *hierarchy* constraints, related to loose semantics in the same way as data constraints are related to initial semantics. Similar constraints are also introduced in the PLUSS specification language [93].

Wagner and Ehrig in [923] defined a language for the definition of constraints and showed that any constraint in this language may be expressed in a canonical form in terms of three basic operations for constraint definition: *generation*, *translation* and *reflection*.

In [226], constraints are used to work with incomplete specifications and a new inheritance relation between such specifications is defined to express the concept of "adding detail". It is shown that parameterization is not needed when dealing with this kind of relation. However, some work has to be done concerning the foundations of this approach. In particular, results have to be obtained which characterize the correctness conditions under which the constructions work "properly".

In [477] the use of equations for constraining behaviors has been discussed.

In [406, 407] a general and flexible notion of constraint is introduced for the institution metaframework, capturing for instance initiality and term-generatedness conditions.

Combining Algebraic and Set-Theoretic Specifications

There are two main schools of thought regarding the formal specification of abstract data types: the model-oriented [2, 886, 265], and the property-oriented [418, 328, 421, 942]. Specification frameworks such as B and Z provide power sets and cartesian products as built-in type constructors, and employ a rich notation for defining (among other things) abstract data types using formulae of predicate logic and lambda-notation. The underlying logic for reasoning about such a specification is a powerful higher-order logic, e.g., based on ZF set theory. In contrast, the so-called algebraic specification frameworks often limit the type structure to sort constants and first-order functionalities, and restrict formulae to (conditional) equations.

An intermediate framework is proposed in [487] where algebraic specifications are enriched with a set-theoretic type structure, but formulae remain in the logic of equational Horn clauses. Types include abstract types (of individuals) as well as product, power set, and function types; operations may be higher-order and partial; the only built-in relations are equality and set membership. Formulae are restricted to Horn-clauses (no disjunction, no explicit negation, variables are universally quantified); models have quite straightforward set-theoretic foundations; specifications have initial models (when consistent) and are amenable to prototyping and reasoning using rewriting and saturation techniques.

Ideas developed in [487] are derived from R^n and G^n logics presented in [488] and are a refinement of G^n-logics with an extended type system. Kinds are added, that did not exist in G^n-logics, and further refinements of the type structure are provided. The deduction rules are similar to those given for G^n-logics but they include three additional axioms. Power algebras proposed and compared to unified algebras in [692], can be encoded in this framework. Compared with higher-order functional languages, the framework provides some higher-order features, since function graphs are specified as set constants, which can be passed to other functions as higher-order arguments. It differs from universal algebras with higher-order types developed in [661] in the fact that the formal basis is set-theoretic rather than purely algebraic and provides a uniform treatment of typing and higher-order functions. Compared with λ-calculi, the abstraction operator is deliberately omitted in order to minimize the functions in the initial model, which may be crucial for limiting the search space for automated theorem proving.

2.2 Expressive Power of Specification Approaches

In order to understand the expressive power and the limitations of the different specification methods, it is interesting to analyze their computability aspects. The main clarifications are due to Bergstra and Tucker who undertook a systematic study of the computability of abstract data types in a rich series of papers (see e.g. [70]); for an overview see [671] or [71]).

An early question was the *hidden function* problem [639, 900]; Bergstra and Tucker proved that (finite equational) specifications without hidden functions are not expressive enough to specify all recursive data structures [71]. On the other hand, they showed that initial algebra specifications with hidden functions describe exactly all semicomputable algebras whereas conditional equational terminal algebra specifications with hidden functions describe exactly all co-semicomputable algebras [70]. A still open question is whether the latter result also holds for specifications with purely equational axioms.

The computability of parameterized abstract data types is studied by Bergstra and Klop [69]. In [66] characterization theorems for hierarchical specifications and partial algebra specifications are given, where it is shown that the use of constraints in full generality increases the power of specifications so much that models of algebraic theories defined by constrained specifications may lie outside the arithmetic hierarchy. Partial algebra specifications are also analyzed by [536]. First results on the expressive power of specification-building operations can be found in [941]. The computational complexity of abstract data types is studied in [630, 327, 44].

Researchers have introduced various syntactic features (such as polymorphism, exception handling, definedness predicates, Horn clauses, ordered sorts, based specifications etc.), generalizing and supplementing the traditional concepts of equational specifications. On the semantical level, a variety of different data type constructions (such as the initial algebra construction, image factorization, restriction, forgetting, deriving, etc.) correspond to such specification techniques. The expressive power of these techniques — in particular when they are combined with each other — has not yet been studied systematically.

2.3 Types

Types are entities that classify collections of individual values (objects). They are used primarily for specifying the functionalities (domains and ranges) of operations, and for restricting the ranges of variables in axioms. The use and nature of types in a wide variety of algebraic specification languages are discussed and illustrated in [696].

The most common approach to capturing type information in algebraic specification is to use *sorts*. At a higher level, it can also be useful to consider types as *theories*; an advantage of this approach is that it supports the use of parameterized specifications to capture higher order features of programs [393]. This approach is treated in detail in [394], which also discusses the use of theories to represent object types, or 'object class templates'.

Subtyping and Order-Sorted

In the order-sorted approach a subsort relation is imposed on the set of sorts with $s' \leq s$ requiring the carrier for sort s' to be a subset of the carrier for sort s, so that a function $f : s \to \bar{s}$ is automatically also applicable to values of sort s'. Thus, the inductive construction of terms is reacher and some syntactical restrictions on the overloading capability have to be imposed to avoid inconsistencies.

In the so called universal order-sorted algebra approach (see e.g. [378, 380, 788, 883]), a strong set-theoretic notion of subsorting is presented. Indeed, if the same function name is declared twice, taking arguments of sort s and s' respectively, then the functions interpreting the two declarations must coincide on the intersection of the carriers of sort s and s'.

On the converse in the so called overloaded order-sorted algebra approach (see e.g. [415, 413]), a weaker condition is required, in order to get many-sorted as a special case. Indeed, if the same function name is declared twice, taking arguments of sort s and s' respectively, then the functions interpreting the two declarations must coincide only on common subsorts of s and s', if any.

All the basic results of equational logic generalize to the many-sorted case. However, order-sorted deduction is more subtle and its correspondence with concepts like replacement of equals for equals and term rewriting needs a careful analysis (see Section 6.2).

In [579] the notion of relation-sorted algebraic specification with built-in coercers is introduced, generalizing the order-sorted approach. The existence of initial algebras as a sound and complete deduction system is presented. A relation-sorted specification SPEC consists of a signature, a set of equations and an arbitrary relation $\to$ on the sorts. If $s \to s'$ for two sorts s, s', then an operator $c : A_s \to A_{s'}$ is assumed in any relation-sorted SPEC-algebra A.

Unified and Classified Algebras

In this approaches, models are homogeneous algebras in which sorts are treated as values of the algebra. This permits the use of sort construction operations such as union and intersection and makes it possible to deal with partiality and parameterization in a very smooth way.

In unified algebras (see e.g. [692, 693, 696]) types may be used as the arguments and results of operations. Individual values are treated as singleton types – this 'unification' of types and values is what gave the framework its name. Signatures are extremely simple (ranked alphabets). Models are distributive lattices, with a distinguished set of individuals, all elements representing sorts, join and meet correspond respectively to union and intersection. All functions are required to be monotone – which seems to preclude the specification of a type constructor corr. to higher-order function spaces. Axioms are Horn-clauses, the atoms being not only equations between values but also equations and inclusions between types, which are semantic and ordered by inclusion. The property of being an individual is expressed by a variant of type inclusion atoms. The empty type is used to represent the lack of a value, i.e., undefinedness, giving something like the effect of partial operations. Type union and intersection are provided.

The entire structure of unified algebras is captured by a simple Horn-clause logic (with equality), allowing easy derivation of results concerning initial models. The definition of 'bounded' data constraints is less easy, because of having to simulate the effects of 'forgetful' functors.

Using sets of equations between types, that are elements of the carrier of unified algebras, context-free grammars, and hence abstract syntax, can be specified (see e.g. [698]). Moreover, considerable conciseness is obtained by the use the Kleene star and other standard operations on types.

Adopting ideas from [692, 693], a compromise between unified algebras, which are one-sorted, and many-sorted algebras has been proposed in [759] where each element of a given set of sorts is viewed as a maximal element in a coalesced sum of disjoint unified algebras, each representing the "constructible" subsets of the appropriate sort.

In [11] a novel axiomatization and rewriting theory for the unified algebra of regular events are developed, see Section 6.2.

A similar approach, called *equational type logic*, is developed in [645, 646]. In this logic, models are standard algebras equipped with a (binary) typing relation, and formulae are positive conditionals with equations and type assignments.

Equational, membership and existence formulas can be expressed in the algebraic framework proposed by [660]. G-Algebras are homogeneous partial algebras where sorts are interpreted as membership predicates.

R^n and G^n logics

A simple, set-theoretic framework is proposed in [488], and provides expressive typing, higher-order functions and initial models at the same time. Building upon Russell's ramified theory of types, the theory of R^n and G^n logics is developed. The parameter n, which is a natural number ≥ 0, gives a bound on the nesting depth of the sets used in interpretations. The difference from Russell's simple theory of types is the consideration of non-term-generated models. R^n logics are axiomatisable by an order-sorted equational Horn logic with a membership predicate. G^n logics provide in addition partial functions. The latter are therefore more adapted to the use in the program specification domain, while sharing interesting properties, like existence of an initial model, with R^n logics. Operational semantics of R^n and G^n logics presentations is obtained through order-sorted conditional rewriting.

ETL [646] is a fragment of R^n logics. On the contrary, unified algebras [693, 692] cannot be encoded in R^n or G^n logics without adding the empty set and singletons. Concerning higher-order features, R^n or G^n logics allow to specify function graphs as set constants, which can be passed to other functions as higher-order arguments. The resulting logic is close to λ-calculi without the abstraction operator. This kind of logic also covers the equational theories obtained by lambda-lifting applied to ML-style functional programs [517].

Polymorphism

Polymorphism is another very important and related notion. So far, there are four major kinds of polymorphism: overloading, coercion, subtype polymorphism and parametric polymorphism (see e.g. [188]). The relation of all these kinds of polymorphism to order-sortedness has been discussed in [415]. It is clear that order-sorted algebra is very suitable for subtype polymorphism. It also provides a solution to the other kinds of polymorphism. But some problems are still open, e.g. the relation between parametric polymorphism and parameterized algebraic specifications.

The specification language SPECTRUM uses shallow predicative polymorphism with type classes [437, 436]. Type classes allow to abstract from concrete sorts by using sort variables and to restrict the range of the sort variables to a given set of types. Technically, each type class is a sort and each type is an element of an order sorted algebra of type classes. For this kind of polymorphism, type inference is decidable and a type inference algorithm was given which computes the principle type for each term.

An extension of type classes to predicated (or qualified) types was investigated in [712]. Qualified types allow to qualify the sort variables by predicates. For this kind of polymorphism, type inference is semi-decidable.

In order to deal with inheritance and subtyping [433] develops a polymorphic type discipline with conditional types and a restricted form of union and intersection types. This typing system is shown to be decidable and a type inference algorithm is given which computes the least type of a term.

Within the Rapide project [680] [545] [546] the interaction of object-oriented formalisms with polymorphic type systems has been explored, resulting in a simple and clean framework for specifying the interfaces of software modules. The type system has made explicit the distinction between subtyping (a relationship between interfaces) and inheritance (a relationship between implementations). The Rapide family of languages has extended the type system to the domain of architecture specifications, and recognized the need for a flexible (though precisely stated) notion of *conformance* [662].

2.3.1 Type theory

Continuation-based control operators can be used as a way to incorporate concurrent and imperative features into type theory, and to give a complete axiomatization of an encoding of these operators (see e.g. [494]). In [497] extensionality principles and quotient types in intensional type theory and the relationship between extensional type theory and intensional type theory, with applications of extensional concepts including constructions on streams and ideas on formalized category theory are studied. Results on the semantics and metatheory of dependent types contained in [497] have been published separately [503, 498, 495, 499, 496].

Tools have been developed for dealing with undecidability in type theory. The work arose in the context of expressing model-theoretic semantics for constructive logic computationally. The techniques include the use of co-inductive types in conjunction with a computational interpretation of classical reasoning, in particular, the typing of non-local control operators with classical logical axioms. This work appears in [905] and [904].

A new type-theoretic interpretation of the Standard ML modules language is currently being investigated on. The emphasis is on the reduction of the modules language to well-understood constructs borrowed from type theory, but taking a different approach from [460] and related work. The ultimate goal is to remove some of the complexity from the formal definition of Standard ML, argue that the separation between core and modules language is unnecessary, and provide practical and well-behaved type inference algorithms for modular programming in the spirit of the core language. It has already been formally shown that the Standard ML notion of type *generativity*, in which each type declaration generates a new type distinguished from all others by an internal "stamp", is equivalent to existential quantification over types, a concept borrowed from type theory [828]. Starting from this observation, a higher-order modules system with a suitable notion of type "generativity" is under study. This work will appear in the forthcoming PhD thesis [829].

[835, 14, 16] present the formulation and metatheory of inference systems for proving properties of algebraic specifications and programs, and for proving that programs meet specifications, with special emphasis on parameterization of programs and specifications. This work draws inspiration from type theories and type systems for programming languages featuring subtyping (which corresponds to specification refinement) [13, 15].

In [4], a method for specifying and implementing abstract data types using constructive type theory is developed. This approach uses the proofs-as-programs paradigm, so that an automated reasoning system can be used to ensure that a proposed implementation meets the specification. A detailed example appears in [906].

The relationship between algebraic methods for program development and the "proofs-as-programs" paradigm of type theory, with a particular emphasis on the various approaches to behavioral equivalence, are studied in [907]

2.4 Partiality

Nowadays it seems quite clear that if algebraic specifications are to be used as a powerful and realistic tool for the development of complex systems they should permit the specification of partial functions. This gives a lot of flexibility and is natural since in computer science one has to deal with the fact that certain algorithms and recursive functions do not always terminate.

There are essentially two ways of specifying partial functions. The first possibility is to explicitly give the domain of definition of each function (for example, by means of a definedness predicate [171] or as the set of solutions of a system of equations [813]). This point of view give rises to the theory of partial algebras, as in Section 2.4.1 below.

The other way is to model partial algebras using total algebras where each carrier set is partitioned into a set of acceptable values and a set of unacceptable ones (the latter set sometimes contains a single element representing "undefined"). A function is considered to be defined for a given argument if it delivers an acceptable value for that argument. Partiality introduced by means of error specifications is of this second type. In this case, special rules have to be given to deal with the application of functions to the undefined element (resp. the unacceptable elements). If one assumes that all functions are *strict*, which means they produce undefined whenever one of the arguments is undefined, one stays in the classical framework of partial functions and just uses a total representation of partial functions.

For an analysis of the relationships among different specification style for partial functions see [21, 194, 690, 686]

2.4.1 Partial Algebras

Partial algebras are an extension of standard many-sorted algebras where function symbols are interpreted as possibly partial functions. Since the interpretations of function symbols are partial, the evaluation of expressions may be undefined; thus, in particular, equalities may be interpreted as existential, holding iff both sides denote the same value, or as strong, holding iff both sides denote the same value or both sides do not denote any value.

Algebraic specifications with partial functions have been investigated for many years. The Munich CIP Group [59] investigated from the very beginning of the idea of algebraic data types the possibility of working with algebraic specifications that specify partial algebras [171], using the conditional fragment of partial logic based on existential equality. Further work has been done on giving an algebraic theory based on the concept of partial algebras for specifying the semantics of programming languages [769, 160, 164, 166, 949, 159].

Partial algebras are used not only to specify in a natural way inherently partial functions (like possibly non-terminating algorithms), but also to describe functions with predictable errors in the requirement phase, postponing the treatment of undefined situations to the level of implementation where the partial functions will correspond to their total implementation([813, 178, 953]).

Partial conditional specifications consist of conditional axioms, with equalities in the (possibly infinite set of) premises and in the consequence, which are interpreted in partial algebras. The use of strong equalities as premises of conditional axioms is motivated, for instance, by the extensionality axiom, needed if partial functional data types have to be described.

In [17, 23] the problem of existence of free and initial models is addressed, giving necessary and sufficient conditions, first from a model theoretical and then from a logical deduction viewpoint. Moreover a deduction system which is complete w.r.t. strong equalities between open terms is proposed, handling also the case of infinitary conjunctions as premises of the axioms.

A sound and relatively complete formal logic for partial functions has been identified, so called WS logic. It is identical to classical sequent calculus with only two exceptions: trivial sequents and variable instantiation require a well-definedness condition [740, 742, 337].

The books [178] and [813] lay a solid foundation for the mathematical treatment of partial algebras. Recently partial algebras have been generalized to partial first-order structures, endowing them with predicates. The chapter [198] gives an overview over different approaches to the treatment of partial functions, presenting a full first-order theory for partial first-order structures. Both model theory and proof theory are discussed, and a reduction of proof theory to proof theory for logics with total functions is sketched.

For relating, comparing, and classifying different specification frameworks a categorical presentation in sense of [601] is very appropriate. [224] provides such a categorical presentation for specifications of partial algebras with conditional existence equations and develops the corresponding structural theory of parameterized specifications. In [432] this work is extended to 2-categories, such that rewriting between terms can be made explicit.

2.4.2 Tamed partiality

In some cases it is also possible to specify partial functions by means of total frameworks, possibly endowed with *ad hoc* features. Basically there are two possibilities: the domain of any partial function, is described as a sort of the signature, or the elements of the carrier are classified in “correct” and “incorrect”.

Explicit Domain

The domain of any partial function f from s into s', is described as a sort $\bar{s}$ of the signature. This approach is able to deal only with recursive domains and introduces the problem of the relationship between $\bar{s}$ and s. In the case of unary functions, the relationship can be naturally expressed in the framework of order-sorted algebras (see e.g. [416]), but if multiple arguments are expected, then the treatment becomes more complex.

Order-sorted Algebras The total algebra approach can be extended to allow partial operations and error handling by ordering sorts with a notion of subsort (see Section 2.3). A detailed account is given in [416], together with a basic results supporting initial semantics for order sorted specification.

Order-sortedness allows partiality to be handled by making functions total and well-defined on the right subsort. This is not always possible, due to computational considerations, because recursively enumerable but not recursive domains cannot be statically constructed by total terms. Moreover, the pure order-sorted approach is too restrictive in term formation, because a function symbol expecting an argument of sort s, subsort of s', cannot accept a term of sort s' even if this can be deduced equal to a term of sort s. For instance, in the standard specification of stacks two consecutive application of the pop are forbidden, even on stacks containing more than two elements. Thus, a *retract* mechanism, inherently based on partial functions, is added.

The connections between order-sorted specifications and partiality have been explored in [415, 884] and in [415, 387, 386] order-sorted specifications are used to represent exception handling.

Labeling

The applications of partial functions to elements outside their domain result in elements that are someway labeled as incorrect. Thus, such elements can be used as inputs for other functions, creating yet more "junk", and must be taken in account in the design of the axioms. But, on the other side, error-recovery, non-strictness and exception handling can be easily described, as the errors are elements of the model carriers.

The first attempt at distinguishing erroneous from correct elements has been proposed, within the equational many-sorted approach, by the pioneering [419]. The resulting specifications are difficult to understand and too complex, due to the limitation of the logic used to express them. To avoid this drawback, *ad hoc* framework have been proposed, where the distinction can be stated in a more compact way, as in error specifications (see e.g. [403]) or, more recently, in guarded algebras (see e.g. [470]), where the operations may be equipped with guards, and the sorts come with data constraints.

An alternative point of view, providing tools for a distinction between errors and exceptions, is that terms instead of elements should be labeled (see e.g. [75, 79, 80])

Although focusing on exception handling, also the theory presented in [379, 381, 386]) can be classified within this approach to partiality.

In the so called based specification (see e.g. [574]), elements are considered as correct if built by some total constructors (the base of the specifications) and functions are partial if their applications do not reduce to such elements.

In the approaches based on explicit typing, like unified algebras, equational type logic, G-algebras (see Section 2.3), partial functions are implemented as total functions whose erroneous computations cannot be typed. However, such computations belong to the carrier of the models and must be taken into account, for instance when giving axioms.

Moreover in unified algebras (see Section 2.3), as the carriers of the models can be thought to be a sort of power sets, the empty set can be naturally used as result of erroneous computations for partial functions. But this approach can only deal with partial functions whose undefined results can be axiomatized.

2.5 Non Strictness

As soon as partial functions are incorporated there is also the question of whether non-strict functions (see e.g. [173]) should be incorporated, too. Non-strict functions are important in the framework of infinite objects and special constructs in functional programming such as `if_then_else`. In many cases additional requirements are assumed for non-strict functions such as *monotonicity*, *continuity* or *regularity*.

In [18] an algebraic framework where function symbols are interpreted as monotonic non-strict functions is introduced and studied from a categorical and logical point of view. In particular, since monotony implicitly introduces disjunctions, non-strict conditional specifications are equivalent to disjunctive specifications.

Non-strict don't care algebras are more deeply studied in [24], where, using the *institution* approach, the relationship between monotonic non-strict functions and their totalizations with the appropriate use of elements representing undefinedness, is fully investigated, showing that the totalization is satisfactory only at set-theoretic level, but that a much more subtle totalization is needed for the correspondence to be between *category* of models and that the conditional fragment is insufficient for axiomatize this relationship.

Although monotonic non-strict functions are essential to represent commands of imperative (conditional choice of functional) languages, they are not sufficient to deal with error-recovery issues. Most approaches for error-recovery, are based on a total interpretation of function symbols, plus some way of labeling the values in correct/errors. This point of view makes models full of junk and the axiomatic description of data types very tricky.

In [78], the labeling is proposed on *terms* instead than on values, in order to capture the intuition that the same value can be denoted by both a correct and an exceptional term. Going a step further in the direction of formalizing the intuition that errors are due to term evaluation, in [195] is introduced a rigorous way to represent *simplification*. The technical counterpart is endowing any (partial) algebra with a congruence on terms (built from the elements of the algebra as variables) representing the shortcuts that can be made during the evaluation process. Correspondingly the logic used to specify such data types is a Horn logic built starting from equality between values (i.e. existential equality as standard in partial types) and an equality between *terms* whose semantics in a model is given by the term congruence of the model itself. The usual algebraic machinery and results apply to this paradigm too.

2.6 Higher-Order Functions and Infinite Objects

Higher-order functions are a concept which occurs in many important applications in computer science. For instance, if a programming language with assignments is treated at a functional semantic level, commands are functions from states to states. The language constructs then correspond to mappings between functions from states to states. If one wants to study recursion, one also has to solve fixpoint equations between commands. Another example are pointer structures which can be very elegantly modeled by higher-order functions. Furthermore, program schemes can be considered as a special form of higher-order functions.

In recent years some attempts have been made to incorporate higher-order functions into algebraic specifications. Möller [682, 683] makes an attempt to incorporate higher-order functions in the framework of partially-ordered algebras with monotonic operators.

[898] uses an infinitary least upper bound operator. [714] gives a rather domain-oriented approach to algebraic specifications including functional domains see also [138]). [144] tries to incorporate functional sorts into algebraic specifications and, in a similar way to Möller, does not associate the complete function space with a functional sort but rather a subset of nameable (term-generated) functions. These decisions influence the interpretation of quantification. The foundations of a theory of higher-order hierarchical specifications is given in [684].

The joint work between Aarhus and Nancy on *Combining Algebraic and Set-Theoretic Specifications* [487] provides in effect a generalization of unified algebras to higher-order, at the same time removing the restriction to monotone functions. However, due to the set-theoretic interpretation there, individuals there are no longer treated as types: the conventional distinction between a singleton and its only element is observed.

Higher-order partial conditional specifications has been investigated in [19, 20]. There, via the usual first-order reduction, the problem of the existence of free and initial models for partial higher-order specifications of term-generated extensional models, can be solved using the theory of strong-conditional partial first-order specifications (see eg [19]).

The investigation on the extensional model class of partial higher-order specifications, that is the class of those models where the functional carriers satisfy the extensionality condition, is carried on in [19] and in [20], where an equationally-complete inference system is given, borrowed from that in [23] for the first-order case via a simulation, based on a skolemization process, using a technique introduced in [21] and then generalized in [196].

There is a relation to work on typed λ-calculus, such as the theory of constructions [239] and the theory of polymorphism [188]; in [136, 135, 137] it is shown how equational axioms can be consistently combined with higher-order λ-calculus.

In [790, 791], order-sorted algebras are extended with higher-order functions for functional programming, data type specification and program transformation. The main results are a new subtype relation, a new semantics for higher-order order-sorted types and a new complete equational deduction calculus.

Infinite objects occur in many contexts. For example, the interpretation of circular pointer structures leads to infinite objects. In connection with interactive programs in general, infinite objects are considered. Functions can also be understood as a special form of infinite objects. Infinite objects allow an elegant treatment of certain algorithms using lazy evaluation [117].

Infinite objects arise in connection with domain theory when complete partially-ordered domains are considered where certain chains converge to infinite objects. Domain theory provides a thorough and extensive theory for domains with infinite objects [862]. A comprehensive study of infinite objects in algebraic specifications appears in [681]. An infinite object can be seen either as a result of taking the limit of certain chains or as a directed set of finite objects approximating this infinite object. This is an order-theoretic and in a certain sense non-algebraic view of infinite objects. But infinite objects can also be understood as arising from the solution of certain fixpoint equations, which is an algebraic way of specifying objects. In [172, 144] there are attempts to give a purely algebraic non-order-theoretic approach to algebraic specifications including infinite objects. The construction of infinite objects is closely related to questions of non-strictness (see Section 2.5). Infinite objects arise only if constructor functions are considered for heterogeneous algebras that are non-strict.

2.7 Non-determinism

Non-determinism in algebraic specification may be studied from two different points of view.

On one hand, non-determinism may arise during intermediate steps in the specification design process. In that case, the final specification obtained as a consequence of further refinements would be deterministic. Thus this problematic has usually been dealt with by means of loose semantics (the only alternative way of handling this problem seems to be [481]).

On the other hand, some systems have inherently non-deterministic behavior (for instance, systems of concurrent processes). The need for such nondeterministic operators has been motivated, for instance, by a series of examples in [926, 929].

The most common approach for dealing with this latter kind of non-determinism in algebraic specification [170, 893, 723, 40] consists of considering non-deterministic functions as relations, i.e. boolean-valued functions. From a practical point of view, it seems natural that term evaluation should be done just by rewriting as usual (see Section 6.2), although such rewriting systems would obviously be non-confluent. The different normal forms obtained when rewriting a given term will correspond to all the possible solutions obtained by a non-deterministic computation. However, when using predicates to describe non-deterministic functions, the evaluation of a term consists in finding the values that satisfy a given relation. This is, in general, more complicated than needed, since it may require the use of narrowing.

A different approach is taken in [606], in which non-deterministic functions are converted into functions whose arguments and results are sets of values. In fact all operations, in the presence of a non-deterministic function, are converted into this kind of operation. The objectives of that paper are to characterize the class of axioms which do not present problems with respect to this kind of transformation.

The main drawbacks of dealing with sets of values are that either we generalize signatures so that operations may take as operands sets of values, and then we may find the kind of problems studied in [606] (problems that may be avoided with order-sorted specifications — see Section 2.3), or else the signature becomes too rigid, since it would not be allowed to apply a deterministic operation to the result of a non-deterministic term because of type conflicts.

An interesting approach taking care of these problematics is the multialgebraic semantics - extending conservatively the traditional algebraic semantics - that has been developed in [927, 928, 929] and related to alternative semantics in [927, 924, 925, 664]. Variants of the multialgebraic semantics have been studied with the restriction to generated models [928] and for the plural (run-time choice) semantics of parameters [931], [929].

The above works focus also on the logic of nondeterminism addressing in particular the problem of restricted substitutivity. Sound and complete reasoning systems (in the natural deduction style) have been introduced for all variants of the semantics. The work on more efficient reasoning tools is in progress (see Section 6.2, also [783]).

Another approach following the idea that non-deterministic functions are functions on sets of values is developed in the unified algebras framework (see e.g. [692]), where a type that includes particular individuals can be regarding as representing a nondeterministic choice. (In fact unified algebras originated from the idea of using nondeterministic choices as types.) The restriction to monotone functions is particularly well-motivated in this context.

An alternative approach based on explicitly considering a choice operator is presented in [537]. The main problem of this paper is that the category of models used seems too complicated.

A survey of algebraic approaches to specification of nondeterminism is given in [930] and submitted to Computing Surveys.

2.8 Metaformalisms

Since "algebraic frameworks" are in a sense variations on one theme, many theoretical results, as well as practical tools, can be developed independently from the particular setting chosen to express them. For purposes of generality, it is best to avoid choosing any particular logical system on which to base a specification approach. This leads to results and tools which can be reused in many different logical systems.

Therefore a bunch of meta-formalisms has been presented in the last decade to describe and classify the relevant aspects of algebraic logical systems. Such metaformalisms answer mainly to three aims:

- the development of theories and tools independently from the underlying concrete formalisms, so that they can be *reused* in different contexts.
- the definition of rigorous comparison techniques for the expressive power of different specification formalisms;
- the modular definition of specification formalisms and the reuse of specifications developed in one formalism into another one.

Depending on the main purpose intended by the authors, several notions of meta-formalisms has been presented, starting from the *institutions* introduced at the end of the 80's by Goguen and Burstall in [406] (see e.g. [407] for a comprehensive introduction).

An institution consists of a collection of signatures together with, for any signature Σ, a set of Σ-sentences, a collection of Σ-models and a satisfaction relation between Σ-models and Σ-sentences. The only "semantic" requirement is that when we change signatures, the induced translations of sentences and models preserve the satisfaction relation.

[956] provides a unified formal description of four relevant specification frameworks based on the notion of institution.

Institutions were introduced to define the semantics of specification languages independently from the underlying algebraic formalism (examples are CLEAR [184], ASL [837], OBSCURE [604] and PLUSS [93]). It is also possible to discuss fundamental concepts such as free constructions [896, 897], observational equivalence [845], refinement [62, 846] and composability of implementation steps [859] in this context. Defining a wide-spectrum language in this way (see for example Extended ML [844]) leads to a program-development framework which accommodates not only different styles of specification but also different target languages.

Analogously, in [955] the semantically compatible embedding of an algebraic specification language and of an algebraic programming language into the general-purpose specification language SPECTRUM (see e.g. [154]) is investigated as relationships between institutions.

Institutions have also been *generalized*, allowing sentences to be evaluated in a generic value, instead than in a boolean result (see [397]). A similar notion is that of *galleries*, introduced in [658], corresponding to generalized institutions, where evaluation of sentences is required to be preserved by model homomorphisms.

For both institutions and generalized institutions, has been introduced the notion of *parchment*, that is a uniform way of presenting the sentences of an institution as terms of boolean sorts over given standard many-sorted sentences and accordingly its models as signature morphisms from each individual signature into a *universal* signature, intuitively consisting of all sets and functions available in the universe of interest. Parchments (see e.g. [406, 407]) allow not only to avoid the proof that the satisfaction condition holds, deriving it from the properties of standard many-sorted validity, but also to combine institutions in a more meaningful way, getting sentences with mixed features (see e.g. [690]).

Foundations were introduced in [789] and are a refined categorical specialization of generalized institution, were sentences form a fibration, corresponding to the intuition that they are based on the notion of term.

Pre-institutions were introduced in [831] and are a relaxed version of institutions, where truth is not required to be invariant under change of notation. They are intended to accommodate a larger class of formalisms and offer a technique to show that a framework actually is an institution, consisting of two steps: first the framework is proved to be a pre-institution and then the satisfaction relation, that is often the tricky part, is proved as well, by coding the pre-institution into an already known institution.

The notion of *specification frame*, introduced in [307], captures the model part of an algebraic framework. Indeed a specification frame consists of a collection of specifications with specification morphisms between them. Further there is for any specification a collection of models of this specification and for any specification morphism there is a reduction of models of the bigger specification to models of the smaller one. Thus technically specification frames correspond to institutions, using specifications as signatures and without sentences, that is with an empty set of sentences for each signature.

Based on this notion [316] develops a unified structural theory of various kinds of algebraic specifications. Moreover [240] covers the abstract investigation of module specifications in arbitrary specification frames, including an extended instantiation by the behavior specification frame.

Specification frames were also used in connection with *amalgamation*, that is a technique for modular construction of data types in [309, 306], for parameterization and parameter passing in specification languages in [308, 318] and for structuring mechanisms of specification languages in [219].

Especially for describing the interconnections of different specification frameworks, [954] proposes a further formal notion of logical systems located between the notions of institution and specification frame. *Institutional frames* keep the concept of signatures but abstract from single sentences and single models.

π-*Institutions*, introduced in [348], cover the entailment part of a framework. They consist of signatures (and signature translations) and, for any signature Σ, of a set of Σ-sentences and of a consequence relation between set of Σ-sentences and Σ-sentences. The consequence relations are required to be reflexive, transitive, monotonic and compact; moreover changing signatures, the induced translations of sentences and models preserve the consequence relation.

Logical Systems are a modular construction of meta-formalisms, having institutions as model part, entailment systems (that are π-institutions where the consequence relation, here called entailment, is not required to be compact), as inference part, and Proof Calculi as effective deduction. They were introduced by [667].

In [672, 194], an overview of the use of meta-formalisms is presented.

A number of categories having institutions as objects, but with different notions of arrows, have been introduced and used, mostly to relate and structure specifications in a multi-logical world (see Section 3.4 for more details on application).

The first definition of arrows between institutions introduces institution morphisms (see [406, 407]), mapping signatures and models covariantly and sentences backwards. Institution morphisms were used mainly to allow constraints described by an institution to be applied to another one (*duplex and multiplex* institutions) an to verify the soundness of deduction systems designed for an institution when applied to (the translation of) sentences from another one.

Another possibility is to translate signatures and sentences covariantly and models backwards. This corresponds to the intuition of representing an institution using another one. Then, if the model component is required to be surjective, then we get *simulations* (see e.g. [21, 194]) while if the domain of the model component is required to be described by a set of sentences, then we get *maps of institutions* (see e.g. [667]), that are the model theoretic part of *maps of logical systems*.

In [21] it is shown how simulations can be used to investigate the relationships between frames, distinguishing three levels, corresponding to different kinds of simulations: "set-theoretic", where the individual models of different frames are related disregarding their categorical and logical interconnection, "categorical", where the relation is between the categories of models, and "logical", where the relation is between specifications.

More complex variations can be given, relating sets of sentences (models) instead of individuals, like in *pre-institution transformations* in [831].

Chapter 3

Structuring concepts

Early work on algebraic specification focused on the means to specify an abstract data type itself, e.g. an algebraic specification was simply given by a signature together with a set of axioms. This approach is feasible only for specifying relatively small systems, but as in programming, it is more convenient to build up specifications of complex systems in a structured way by combining smaller specifications. Structuring large specifications into smaller component specifications that can be treated separately facilitates their construction, understanding, analysis and implementation. Furthermore, it enhances reusability and, if structuring concepts are independent from a particular logic within a suitable logical framework, heterogeneous development of specifications is possible.

Structuring concepts have been incorporated in virtually all algebraic specification languages since CLEAR [183]. Their theoretical foundations have been investigated intensively (see for example [328, 329, 220, 221, 840, 942, 619]).

Concepts which were developed to support structured specifications include a wide range of different *specification-building operations* (Section 3.2), and various different approaches to *parameterization* and *modularization* (Section 3.3).

Most algebraic specification languages (e.g. CLEAR [184], ACT ONE [311, 222], ACT TWO [346, 347], ASL [851], OBJ3 [421, 422], Extended ML [844, 534], CIP-L [54], PLUSS [367], LARCH [445, 446], OBSCURE [604, 605], PAnndA-S [491], SPECTRAL [590], SPECTRUM [154, 434]) offer a full range of operators for building specifications.

3.1 Structuring elements

The structuring elements investigated can be roughly divided into concepts supporting the organization of complex specifications, concepts of structured development and concepts providing for a certain independence from logical formalisms.

3.1.1 Structured specifications

The need to build up large specifications by combining, extending or refining smaller ones was recognized very early on. In [443] specifications were built by successive extension of previously existing ones (see also [949]); moreover, the notions of *hierarchical consistency* and *sufficient completeness*, as the conditions to be achieved by extensions in order for them to be considered correct, were defined (see also [72]). A more sophisticated approach is the notion of two-dimensional specification design [405]. The *horizontal structure* of such a hierarchical organization corresponds to enrichments or extensions and the *vertical structure* to specification refinement.

Parameterization is another important mechanism for structuring specifications or algebras. It allows to exhibit genericity in specifications and programs which supports their reuse in different situations. In [835] it is argued that there is a clear distinction between parameterized specification and the specification of parameterized programs.

3.1.2 Structured development

According to the formal methods approach to software development, programs are the result of a *refinement process* starting from a high-level specification and ending in a low-level program. If each refinement step can be proved correct then the resulting program is guaranteed to satisfy the original specification.

In order to make this process precise, an adequate formal notion of refinement step is needed and its "compatibility" with horizontal structuring has to be studied. Among the many definitions in the literature three different ways for a specification sp_1 to *refine* or *implement* a specification sp_2 can be singled out:

1. The most straightforward idea is to demand that the model class of sp_1 is contained in the model class of sp_2, i.e. this formalizes the intuitive idea that more details (design decisions) have been fixed in sp_1. This notion has been developed by [851, 845, 859]. Also, the subclass relation that may be found in [226, 739] may be seen as a special class of such refinements.

2. The language of sp_1 is "less abstract" than the language of sp_2. For instance in [58] the relation between explicit and implicit definition in algebraic specification is studied. Also, in [734] and in [544] this is done for equational specifications and recursive definitions.

3. The specification sp_2 somehow "simulates" sp_1. This is the classical notion of implementation that goes back to the pioneering paper of Hoare [489]. This notion has been developed by [418, 448, 321, 322, 288, 320, 412, 850, 319, 362, 76, 158, 859] among others. Probably the most influential of these is [320].

An approach which unified and generalized all previous approaches is [838]. This definition clarifies the role of certain properties (both vertical and horizontal composability [405]) which many notions lack [732, 733, 735]. *Vertical composability* is essential for stepwise program development since it allows successive correct refinements to be composed into a single correct refinement step. *Horizontal composability* is necessary for modular program development since it allows correct refinements of separate modules of a program to be combined into a single correct refinement of the whole program. A survey of implementation approaches is in [737].

3.1.3 Independence from the underlying logic

Various logical formalisms have been used for specification. Examples include first-order logic, higher-order logic, infinitary logic and temporal logic. Furthermore, many of the logical formalisms come in different variants, supporting partial functions or not, with only an equality predicate or with arbitrary predicates. Corresponding to the logic there usually are different notions of signature (syntax) and model (semantics). Since none of the logical formalisms is strictly superior to the others it is a matter of taste or appropriateness to the situations which formalism to choose.

Independence from fixing a logical formalism for specification yields the enormous advantage of using the formalism most suitable for the specification at hand. Defining mechanisms for structuring specifications independently from logical formalisms and finding mechanisms for "switching between logics" allows the system designer to choose the logical formalism most appropriate to the component at hand and leads to heterogeneous specification and development. Of course, a suitable meta-formalism encompassing a great number of logical formalisms is needed for this idea, the best known being *institutions* (cf. Section 2.8).

The notion of institution provides a useful framework in which to study composability of specification modules, both at the level of building systems from components, and at the level of translating one logical theory into another. The level of building systems from components is appropriate for studying the horizontal structure of systems; while the level of translation between logical theories is useful for, among other things, studying the vertical structure of systems, in particular relationships of refinement and implementation between specifications. These relationships are well-understood at the level of individual specifications; however, it is important that refinement and implementation should behave coherently across the horizontal structure of systems. In other words, it should be possible to refine system specifications by refining their individual components. These topics are addressed in [279].

3.2 Basic constructs

A structured specification is given by an abstract syntax of *specification expressions* generated by a set of *specification-building operations*. Although a lot of specification-building operations have been proposed most of them are similar. A major difference often is the semantics given to structured specifications. Two different approaches [942] are usually considered (cf. e.g. [311, 328] and [851, 941]):

1. the *presentation oriented* semantics associates to the specification expressions some sort of normal form of specification, e.g. a pair consisting of a signature and a set of formulas;

2. the *model oriented* semantics associates a class of models with each specification expression.

A discussion of structuring concepts is in [942, 619]. A comparison of several specification languages and their specification-building operations is in [944].

The most trivial and straightforward concept for structuring a specification is naming specification expressions or even parts of the text of a specification and keeping them in a global environment. This mechanism can then be used to split the text of a specification into several texts that are then "imported" into the final specification text. Most specification languages support this form of input splitting or abbreviation.

Union is the combination of two or more specifications (which possibly have common sub-specifications) to form a new specification. In its simplest form it corresponds to a set-theoretic union of the presentations or disjunction of the model classes. In the presence of common sub-specifications this can lead to inconsistencies, for example when there are operations with the same syntax but different semantics. Hence a disjoint union or elaborate conditions on the union operation are needed (see e.g. [605]). A refined approach is the *amalgamated union* [328, 306] that forms a disjoint union while identifying the semantics of all symbols occurring in a shared sub-specifications. This shared sub-specification has to be given as an additional argument to the union construct. Amalgamated union has a categorical foundation and corresponds to pushout constructions.

In [471] the pushout construction is investigated in the category of order-sorted algebraic specifications. Since the existence of pushouts of order-sorted specifications depends on the existence of pushouts of the underlying signatures, conditions on signature morphisms are established which guarantee that desired properties of order-sorted signatures such as regularity or locally upward filteredness (see [416]) are preserved by pushouts.

Another essential kind of operation which may be found in all specification languages is *renaming*. This is used to avoid name clashes and to increase the readability of specifications.

Often it is necessary to *hide* sorts or operations, either for conceptual reasons, i.e. the sort or operation was only needed temporarily or as an auxiliary construct, or to reuse a specification, "plugging" it into an interface that demands less sorts or operations.

For a hierarchical design *enrichment* of specifications is paramount. Among the specification languages mentioned at the start of this section there are various enrichment primitives that reflect various combinations of hierarchical conditions such as sufficient completeness and hierarchical consistency (e.g. the enrichment primitives *using*, *extending* and *protecting* in OBJ 3, or the enrichment primitives *assumes*, *imports* and *includes* in LARCH). In the PLUSS specification language, completed specification components and specifications under development are distinguished not only at the level of the enrichment specification-building primitives but also by a (syntactic) distinction between the specification modules themselves. Thus, appropriate syntactic rules prevent enriching completed specification components using overly permissive enrichment primitives, while this is allowed for specifications under development.

A specification obtained via enrichment from another specification can be considered to be a *sub-specification*, it usually has a restricted class of models. In many cases it should be admissible to insert a sub-specification whenever the (super-)specification is required. The connection between a specification and a "sub-specification" is in general determined by a *morphism* from the specification to its sub-specification. In a general situation the morphism may express additional renaming or sharing of components. First techniques for treating morphisms as "first-class citizens" of a specification language and a calculus for determining the sub-specification relationship have been described in [86].

Being abstract requirements for implementations, specifications should always admit all models that are "similar" to the given one. The usual notion of similarity corresponds to the notion of *isomorphism* and most specification formalisms require model classes to be closed under isomorphism. Often this notion is not general enough and it seems desirable to include all models that *behave* like a given model with respect to some set of *observable* sorts or operations, e.g. sets and lists behave in the same way with respect to the operation "$\in$". This lead to the study of notions of *behavioural abstraction*, *behavioural equivalence* and *behavioural satisfaction* [811, 727, 860, 307, 474, 77, 113, 240] and the introduction of specification operators *abstract* or *behave* that form a closure under behavioural equivalence.

Hidden sorted algebra, because it is based on the notion of equality *up to observability*, allows refinements that make use of subtle changes of representation (implementing stacks with arrays and pointers is the classic example). A powerful proof technique for proving correctness of refinement is given in [401, 643]. This technique is based on splitting signatures into generators and derived functions (or into methods and attributes for the object paradigm), and reduces the need for complicated inductive proofs. In [644], it is shown that this proof technique is an example of coinduction (cf. the comments on final and behavioural semantics in Section 2.1).

Hofmann and Sannella [502, 500] have studied behavioural abstraction and behavioural satisfaction in the context of specifications with higher-order logical formulae as axioms. The central result is a generalization of [113] to this context, where the role of equality in the context of behavioural semantics is revealed as special case of a much more general notion. The fact that behavioural satisfaction is expressible in terms of ordinary satisfaction in higher-order logic leads to techniques for proof in specifications with behavioural semantics.

An interesting principle that emerges from the study of institutions is that syntax is composed by colimits, while semantics is composed by limits [407, 394]. It turns out that the basic constructions that support horizontal and vertical structure in specifications also support distributed structure in systems. In [408] a construction is given for parallel connection which allows distributed systems to be specified by composing their parts in such a way that communication is achieved through sharing subcomponents. This construction has a universal property, and recent results show that it is an instance of a colimit construction in a suitable category of hidden sorted specifications [644]. Moreover, models (i.e. implementations) of component parts can be composed by a limit construction to give a model of the composite system. At the semantics level, this is an example of the 'Behaviour is Limit' slogan from [395], which provides the basis for recent work on algebraic approaches to interconnections of abstract machines: [641] gives a sheaf theoretic account of systems built as limits of diagrams of object specifications, an approach further developed in [642, 217] to provide a 'distributed' semantics for an object-oriented programming language.

[219] describes mechanisms supporting application oriented aspects like *sharing* and *structural equivalence* extending the work of Burstall and Goguen on *based objects* [184]. The structuring operations are described by colimit constructions on comma categories. These constructions are identified as the basis mechanism needed for tools handling structured specifications. [957] shows the possibility to reflect colimit-based semantic definitions directly by the tool design for a specification language without violating efficiency constraints.

The Rapide family of languages [680, 545, 546] has extended the type system to the domain of architecture specifications, and recognized the need for a flexible (though precisely stated) notion of *conformance* [662]. In particular it has formalized the notions of *required* as well as provided features of a module interface, and introduced a means of grouping interface features analogously to the notion of plugs and sockets in hardware.

Recurrence relations play an important role in the solution of many problems, such as those in Mathematical Physics and Operations Research. Modular software construction for solving recurrences has been studied in [466, 912, 251, 252], contrasting a functional programming approach, *Sapphire* [466], with structuring of software libraries for more conventional programming languages, the *Structural Blanks* approach [252].

The module specification language presented in [693] is used with unified algebras [692] (see Section 2.1) and in action semantics [694]. It allows modules to be nested and mutually-dependent, and it provides a novel treatment of instantiation of generic modules, using subtype inclusion axioms instead of signature morphisms.

In order to reap the advantages of structuring other important activities, as for example proving properties of specifications or proving the correctness of refinement steps, should be "compatible" with the structuring.

In [943] a sound and complete infinitary proof system is presented for verifying first-order properties of structured specifications which are built using ASL-like specification building primitives. In [475] and [476] this proof system is adapted and extended to also include observability operators. In all these papers proof systems for verifying implementation relations between specifications are provided.

Behavioural proof techniques are needed for proving the correctness of abstract implementations in an observational framework. In [112] an elaborated study of behavioural theories and a concrete method for proving observational consequences of specifications is presented which extends and integrates previous work on this topics (cf. [105, 108, 109]). The main interest of the proof method is to avoid the use of explicit context induction (cf. [472]) and to generate (first-order) proof obligations under which proofs of observational properties can be reduced to standard (first-order) proofs. Hence standard proof techniques (such as those implemented in the Larch Prover) can be used to establish observational consequences of specifications. This technique is the basis for applying the proof rules for behavioural implementations provided in [111].

Methods for proving the correctness of behavioural implementations were also studied in [643].

3.3 Parameterization

The use of parameterization concepts has become a powerful tool for exhibiting genericity.

Parameterized specifications were first introduced in [183, 900, 184, 288]. In [900] the semantics of a parameterized specification was defined in terms of free functors, while in [183, 184, 288], working at the specification level, parameter passing was defined in terms of pushouts (a set-theoretic version of this is given in [833]). In [323] the two approaches were combined. In this framework, correctness of parameterized specifications was defined in terms of the *actual parameter protection* and *passing compatibility*, which were proved equivalent to persistency of the associated free functor [323, 736]. In [304] this framework is extended to deal with parameterized specifications with requirements and [708, 746, 747] generalize the basic results to deal with parameterized specifications with boolean constraints.

Many of the underlying constructions are best understood in the context of institutions [407, 279]. Some subtle examples of parameterized programming and theorem proving, using 'parameterized views' between parameterized specifications, are given in [410]. This style of parameterized programming has been adapted to provide a module facility for the object-oriented specification language FOOPS [417]

Ganzinger [362] defined parameterized specifications within the tradition of final semantics and also provided a characterization of persistency in terms of hierarchy consistency and sufficient completeness properties. Similar results have been obtained by several authors for different approaches to parameterized specifications with behavioural semantics [412, 811, 812, 726, 708] and with term generated semantics [946].

In [709] a proof-theoretical characterization of the property of "no confusion" was provided. This was an old open problem since Ganzinger provided the first characterization of persistency. According to Ganzinger's result, persistency was shown to be equivalent to two proof-theoretic properties, namely sufficient completeness and hierarchy consistency. In particular, the property of "no junk" was shown to be equivalent to sufficient completeness and "no confusion" was shown to be equivalent to hierarchy consistency for sufficiently complete specifications. The characterization shows that "no confusion" is equivalent to a variation of the property of hierarchy consistency that makes use of conditional equations.

The parameterization for order-sorted algebraic specification is studied systematically in [793] by following the notion of order-sorted algebraic specification by Smolka, Nutt, Goguen and Meseguer [884]. Some notions of parameterized specification and parameter passing are introduced and justified in this context. Especially, a notion of coordinate parameter passing is introduced, which relates different actualizations of one parameterized specification in the very spirit of the order-sorted algebraic specifications. The semantics of coordinate parameter passing is proved to protect the semantics of each single actualization.

A completely different approach is taken in ASL [851], in which parameterized specifications are presented as computable transformations of specifications, in the style of λ–calculus.

These parameterization concepts deal with *parameterized specifications* and the parameterization is not intended to reflect the parameterization of the programs that are the result of a development process, i.e. they are not intended for the *specification of parameterized programs*. The difference is a conceptual and a semantical one. A parameterized specification denotes a map between the class of possible "argument models" and the class of possible "result models" whereas a specification of a parameterized program denotes the class of all maps between argument and result models. This distinction was first clarified in [835] where a specification language based on ASL was introduced that provides constructs for parameterized specifications and specifications of parameterized programs.

Kahrs showed a shortcoming of the module system of Standard ML and related languages, namely a limited capability to abstract types from modules which restricts the way that polymorphism and parameterized modules (functors) can be combined. The problem and its solution is presented in [527].

The design of a modular system with given interfaces had been reduced to the generation of a specification in an algebraic specification grammar. In order to obtain more general interconnections [333] introduces a different notion of derivation, where standard derivation steps can be preceded and/or followed by reduction steps using specification morphisms. Motivated by recent results on canonical derivations in High Level Replacement Systems, the correspondence between designing and rewriting can be used to check the equivalence of modular systems by reducing them to the normal form obtained from the canonical derivation. To make the search space more manageable, criteria for avoiding the application of certain productions, which would produce specifications already obtained, have been developped. These criteria are static and can be checked when the library of interfaces is constructed/modified. The search through the reduced tree can be guided by a notion of similarity [215] which maximizes at each step the elements needed in a predefined goal specification. The applicability of a production is determined by the existence of a specification morphism: unfortunately [216] it has been shown that such a problem is NP-complete.

The same paper also advocated the use of *higher-order parameterization*. Higher-order parameterization can be used to structure a specification more flexibly and it can be used in the implementation process. An example of this is to provide a higher-order composition constructor that builds up a specification of a parameterized program from two specifications of parameterized programs with "matching interfaces". The two specifications can then be implemented separately.

In [192] an ASL kernel specification language over any institution is presented which supports programming in the large by two different means: specification-building operators and parameterization over higher-order variables. Specification-building operators allow the construction or combination of specifications, whose semantics is loosely defined as the collection of all models satisfying the inherited axioms. Parameterization abstracts just in the same way function definition does in traditional programming languages and is based on the simply typed λ-calculus (see [942]). In this context, function application maps specification expressions into specification expressions, this means, parametric specifications are neither interpreted as specification morphisms nor define parametric algebras or structures (cf. [835]). A contextual proof system is provided which extends the structured proof system of [943] and allows the derivation of formulas valid in a specification term under certain assumptions for its free variables. The system is correct, and also complete if the underlying logic satisfies interpolation (cf. [67]).

The work presented in [190] revisits the refinement implementation relation. This partial order relation defined by signature equality and model class inclusion extends to parametric specifications in the standard way. A system for the derivation of this extended relation is presented, which is based on the contextual proof system, that is, derivations are relative to a context of value assumptions. It extends the one presented in [943], and in the same way as there it generates proof obligations. The system is correct and complete –if the interpolation property is satisfied– w.r.t. denotable assumptions.

These results enable the introduction of semantic conditions to be fulfilled by the argument passed to a parametric specification. That is, an actual parameter may need not only to have some syntactic characteristics but also to exhibit a particular behaviour in order to be a valid argument of a parametric specification. In a simple case, the satisfaction of formulas may be expressed using the implementation relation, that is, an actual parameter is a valid argument if it proves a set of formulas. This typing mechanism generalizes the one presented in [340]. In the same way as in that work, a step of β-reduction is possible only if the conditions can be derived, otherwise the application is undefined.

All these results are compiled in [189]. In this thesis, moreover, the ultra-loose approach to specification semantics is investigated. The basic idea is that universally quantified formulas are tested just in the denotable part of the corresponding universe; we speak of "relativized universal quantification." In this way, the ultra-loose semantics allows the reasoning with constructors. Its main impact, and the motivation for this definition, is the resulting refinement relation, which approximates much better the intuitive implementation process a programmer has in mind.

In its first definition, the ultra-loose logic does not fulfill the so-called satisfaction condition and thus fails to be an institution. It can nevertheless be adjusted by explicitly declaring w.r.t. which signature the universal quantification has to be restricted. In its turn, and given that this signature may be unknown by the time the formula or axiom is written (e.g. if a reference is made to the elements of a sort to be supplied by an actual parameter), the universal quantification gives rise to formula terms –and not just to plain formulas– with possible occurrences of signature variables. This combined with the contextual proof system demands just one additional inference rule.

Classical approaches to parameterization in axiomatic specification languages require the user to explicitly handle specification instantiation. This often makes specifications less readable and manageable. The use of type classes in *SPECTRUM* allows a simple parameterization mechanism with implicit instantiation. When combined with classical parameterization, type classes allow to achieve both elegance and power [435].

In [685, 689], Mossakowski examines four specification methods with increasing expressiveness. *Parameterized recursion theory* allows to characterize the power of parameterization in the methods, using a computational model based on Moschovakis' search computability. The four specification methods can be characterized by four different notions of semicomputable parameterized abstract data types, which differ in the availability of the parameter algebra and of nondeterminism. These characterizations further lead to different algebraic properties of specifiable parameterized abstract data types. Together with example parameterized abstract data types, they enable the proof of a hierarchy theorem.

Given a sample parameterized abstract data type, the algebraic properties help to find out the lowest position (i.e. the most restricted method) in the hierarchy usable to specify it. This is important because the available tools may become weaker, if we choose a too general method.

Module specifications may be seen as a generalization of parameterized specifications. Several approaches with different aims have been developed.

In [120, 335, 314, 936] a module concept is defined to serve as basis for modular specification development. According to this approach, a module specification consists of a realization specification, a formal parameter specification and explicit import and export interfaces. The theory of module specifications combining initial and loose semantics [305] and their interconnections (horizontal structuring) is well-developed in the basic algebraic case with persistent (as well as conservative) free functors. The basic horizontal structuring operators for such module specifications are *renaming*, *union* and *actualization*. Others are *extension* [120], *recursion* [751], *product* and *iteration* [752]. There are first ideas how to extend the concepts to algebraic specifications with constraints [336, 315] and how to handle the development of specifications (vertical development) [312, 329]. This concept of module is the basis of the specification language ACT TWO [345, 346].

The module system of the Standard ML programming language [462] is adopted by the Extended ML specification language [836, 844, 848, 839, 534]. Its aims are to serve as the basis for the modular development of (ML) programs from specifications. The module system of Standard ML includes facilities designed to allow large programs to be structured into self-contained program units with explicitly specified interfaces. In this system, interfaces (signatures) and their implementations (structures) are defined separately. Structures may be built on top of existing structures, so each one is actually a hierarchy of structures. Functors are "parameterized" structures; the application of a functor to a structure yields a structure. A functor has both an input signature describing structures to which it may be applied, and an output signature describing the result of an application. The development of programs from specifications is simplified by the fact that the mechanisms used for structuring specifications and programs are identical.

A related module system is Pebble [185], which abandons the distinction between type and value. This gives more power and simplifies matters but as a consequence it sacrifices guaranteed termination for type checking. A similar powerful system based on type theory with decidable type checking is ECC [626, 628]. The application of ECC and related systems to problems of specification and verification has been demonstrated in [659, 627]. [817] demonstrates how modular specifications can be expressed in type theory and investigates the validity of the module algebra axioms for these translated specifications.

A further module concept similar to that of Extende ML is presented in [619]/

Notions of implementation and refinement may be extended to deal with module specifications. For instance, module refinements in Extended ML are studied in [848]. In [859] a language-independent theory of module refinement and implementation is given that allows the horizontal and vertical composition of implementation steps. Also the theory of algebraic module specifications [120] with horizontal structuring operations is extended in [312, 329] by vertical development steps leading from abstract module specifications to more concrete versions. The basic concept of refinement is given by specification morphisms between corresponding parameter, export and import interface specifications. It is shown that refinement is compatible with the horizontal structuring operations composition, actualization and union. This allows modular vertical development. Other kinds of vertical steps between module specifications are *simulation* and *transformation*.

A theory of specification modules and system specifications is developed in [106] which, based on the stratified loose semantics approach, studies the impact of modularity on the semantics and on the implementation of software specifications in a general logical framework. The approach meets the following requirements: the independent construction of implementations for the single constituent parts (modules) of a system specification and the encapsulated development of each implementation part using the principle of stepwise refinement.

3.4 Multiple logics

There is a variety of specification languages for the formal specification and development of correct software systems. They differ in purpose, expressiveness, level of abstraction (requirement, design, implementation), notation, available tools etc. It has been argued that one should not construct a single universal all-purpose specification language (which would be very clumsy), but rather relate different languages based on different logical frameworks. Thus there is a need for a meta-notion of logical framework and a notion of map between logical frameworks in order to be able to *compare* them.

With the help of this, one specification can contain parts written in different logical frameworks using a multiparadigm specification language [22]. Moreover, tools like theorem provers can be borrowed along embeddings of logical frameworks in a way that soundness and completeness is preserved [196].

A first study of different structuring concepts in an institution independent fashion is [837].

[318, 516] builds over previous work by Sannella, Sokołowski and Tarlecki. In particular, compositional semantic definitions are given to different kinds of parameterization and parameter passing mechanisms for the basic first-order case in an institution independent framework.

[575] points out that colimit constructions provide a class of parameterized data types with variable formal parameter. This observation gives some hint how the concept of parameterization should be generalized in such a way that the formal parameter part becomes more flexible.

Theories with hidden sorts provide a setting to study the idea of behaviour and behavioural equivalence of elements. But there are variants on the notion of theory: many sorted algebras, order sorted algebras and so on; the theory of institutions can be used to develop ideas of some generality. Burstall and Diaconescu [180] formulate the notion of behavioural equivalence in a more abstract and categorical way, and give a general explication of “hiding” in an institution. This is used to show that both hidden many sorted algebras and hidden order sorted algebras yield institutions.

[22] is a first attempt at a formal foundation of specification languages allowing their basic modules to be defined in several formalisms. More precisely a rigorous notion of a compositional tool for importing/exporting specifications between two instances of one specification metalanguage on different basic algebraic frameworks is proposed.

Adopting the notion of institution as a synonym for formalism, the simulation of basic and structured specifications is discussed, introducing the concept of *simulation independent metalanguage*, a generalization of institution independent languages, which allows “putting together theories *from different formalisms* to make specifications”. Since simulation generalizes the notion of implementation and allows relating implementations in different formalisms, a third dimension is added to the well known horizontal and vertical compositions of specifications, typical of Clear and ASL.

[196] proposes a technique to borrow the missing components (as well as their associated tools if they exist) from a logic that has them for use in another logic. The relevant structure is transported using maps that only involve a limited aspect of the two logics in question—for example, their model theory. The constructions accomplishing this kind of borrowing of logical structure are very general and simple. They only depend upon a few abstract properties that hold under very general conditions given a pair of categories linked by adjoint functors.

This “borrowing” is further investigated in [197], showing more practical applications, and related to the cartesian construction associated with a fibration.

Based on the work in [196] some guidelines for the introduction of new types of logical framework, which keep things neatly together and comparable in order to achieve a neat structuring principle in the metatheory, are set up in [687].

According to this, in [578] and [687] several types of “map between institutions” are introduced and examined, driven by a systematic study of different logical frameworks of partial and order-sorted algebras in [686] and [688].

In [710, 711] a logic-independent formal framework for studying the correctness problem of modular software systems is provided. An abstract framework is introduced that allows to represent, as specific instances, most concrete modular frameworks. In particular, the framework presented is "parameterized" by the specification and programming language formalisms, by the semantic constructs associated to modules and by the behavioural equivalence relation used to define module refinement. In this framework, it is shown that, to achieve modular correctness, it is sufficient that the given programming language satisfies a property of "stability" with respect to the given behavioural equivalence relation. In particular, modular correctness is shown to be independent of the satisfaction (or not) of the horizontal and vertical composition properties at the specification level. The framework and the results shown in this paper can be seen as a continuation of some of the ideas that were present in previous work by Schoett, Sannella and Tarlecki and in [737] and its expanded version ([738]) where, in addition, a survey of the various implementation concepts in the literature was presented.

Using institutions to represent specification frameworks, the problem of modular construction of formal software specification methodologies can be rephrased in terms of operations among institutions.

In [200], two operations on institutions, that are originated by abstracting from the definition of several institutions for very-abstract specifications in the field of concurrency, are presented, as a first step in building a library of formalism enrichments and assemblings.

But many enrichments of basic frameworks with new features, tailored for special purposes have meaning only for those institutions that represent "algebraic formalisms". Thus, in order to give sound foundations for the treatment of such operations, a preliminary step is the formal definition of which institutions correspond to algebraic frameworks. In [199] the definition of *algebraic-oriented* institutions is introduced and motivated by the formal definition of some operations adding features to basic algebraic frameworks, that on any algebraic-oriented institution yield an algebraic-oriented institution, too; so that the result can be used as input for other operations, modularly building a formalism as rich as needed by the application. Technically algebraic-oriented institutions are described by (standard) algebraic specifications, so that both theoretical and software tools are at hand to help in the building process; moreover algebraic specification users already have the know-how to understand and manipulate metaoperations building algebraic formalisms.

Another topic is the systematic *combination* of logical frameworks. It turns out that this cannot be done successfully with institutions – parchments have to be used instead. In [690], the category of parchments and parchment morphisms is shown to be cocomplete, and this is applied to the combination of several features of logical frameworks with partiality. Compared to the approaches of Diskin [283], who incorporates algebraic logic methods into institutions, and Hilberdink [483], who *generates* term and formula syntax and the semantical evaluation of terms and formulas from model theory, parchments allow to write *specifications* of these in a well-structured way amenable to combinations.

Chapter 4

Specific Paradigms

4.1 Functional Programming

4.1.1 Functional Programming and Algebraic Specification

Functional programming and algebraic specifications are closely related. In algebraic specifications, families of functions are often introduced and specified by means of conditional equations. In functional programs, functions are defined by a combination of language constructs based on a number of built-in functions which are traditionally combined using if-then-else, function abstraction, and a fixed point operator. If a sufficiently powerful algebraic specification language is used, functional programming can be seen as a sub-style of algebraic specifications. More recently, definitional (conditional) equations with constructor terms (below the leading function symbol on the left-hand side) further close the gap between functional programming languages and algebraic specification.

A great deal of work has been done in the area of functional programming in recent years, including the development of powerful functional programming languages, such as FP [47], Standard ML [462], Miranda, Haskell or Gofer. A term rewriting treatment of functional programs is available. Implementation techniques for functional programs have been carefully considered, and special machine architectures have been developed for the implementation of functional programming languages. Therefore, a functional programming language provides an excellent target language for the development of programs from algebraic specifications.

Functionals (or higher-order functions often reduce the effort for program development significantly and serve as a kind of "program generator". The technique of homomorphic extension functionals (and their specialisation) is important in this context (cf. e.g. [583]). Algebraic axioms for higher-order functions are a powerful specification method (cf. e.g. the theory of lists in [118]) and provide a basis for formal reasoning about functional programs.

4.1.2 Language Developments

Opal. The development of the *Algebraic Programming Language* Opal [343, 282] is an ongoing activity at TU Berlin. The aim of this work is to provide an algebraic high–level language which is equipped by a compilation system for generating efficient code. The Algebraic Programming Language Opal (with initial emphasis on functional programming) has been extended by a property language, so every part of a software development from the requirements specification to the final implementation can now be expressed within Opal. We hope for an even better integration in the successor to Opal, called Opal2 [281]. The language design took place in 1995 and first tools are now being worked on. We plan to reuse as much of the current Opal backend as possible, so we expect to inherit the efficiency of the Opal compilation system for a future Opal2 system.

Sapphire. The purely functional programming language Sapphire [466] (also in [912, 251]), developed At University of Bergen, is a highly parameterized language, which includes algebraic specifications as an integrated part of the programs. The underlying semantics is based on guarded algebras [470] and special graph structures which allow the translation of recursive programs into non-recursive loops. Sapphire programming has been explored in [9, 176] as a means of using algebraic technology in the programming of parallel computers. The approach was sketched and compared with other parallel programming models in [464, 468, 465], and [624] relates Sapphire to Standard ML. Program development in SAPPHIRE supports the methodology advocated in [842]. A sequential prototype compiler now exists for the language, and a parallellising version is expected to be available in 1997.

4.1.3 Theoretical Background

Termination of rewrite systems. Proving termination of rewrite systems is an important issue. A promising method is based on interpretation. This generalizes the idea that, for proving termination of a process, one associates a number with each state and one proves that each step decreases this number. Since one works with open terms (i.e. terms with variables), one associates with each term a function. Usually people use polynomials. Lescanne has proposed a new method for proving termination based on elementary interpretations, i.e. functions built with polynomials and exponentials. This method has been implemented in CAML and is available in ORME [610].

work needed: work at Bremen on termination

Substitution in λ-calculus. In λ-calculus, substitution is usually described as a mechanism external to the calculus itself. If one can internalize this operation, one may expect to have a better understanding of the underlying computation mechanism which would be very useful for a better implementation of functional languages. Starting from principles of abstract data types developed in COMPASS, Lescanne has given a systematic description of several calculi of explicit substitutions. These systems are orthogonal and have easy proofs of termination of their substitution calculus. The last system, called $\lambda\upsilon$, entails a very simple environment machine for strong normalization of λ-terms [609]. A presentation of $\lambda\upsilon$ can be found in [611] and a complete study in [63]. Other references on explicit substitutions are [1, 249, 457, 612, 141, 285, 535, 609, 665]

Higher-order unification with subtypes. In [799, 725], reduction, equality and unification are studied for a family of simply typed lambda calculi with subtypes. The subtype relation is required to relate base types only to base types and to satisfy some order-theoretic conditions. Constants are required to have a least type, i.e. "no overloading". We define the usual beta- and a subtype-dependent eta-reduction. These are related to a typed equality relation and shown to be confluent in a certain sense. We present a generic algorithm for pre-unification modulo beta-eta-conversion and an arbitrary subtype relation. Furthermore it is shown that unification with respect to any subtype relation is universal.

Matching. The standard second-order matching algorithm by Huet may be expansive in matching a flexible-rigid pair. On one hand, many fresh free variables may need to be introduced; on the other hand, attempts are made to match the heading free variable on the flexible side with every "top layer" on the rigid side and every argument of the heading free variable with every subterm covered by the "top layer". Curien, Qian and Shi [250] propose a new second-order matching algorithm, which introduces no fresh free variables and just considers some selected "top layers", arguments of the heading free variable and subterms covered by the corresponding "top layers". A first implementation shows that the new algorithm is more efficient both in time and space than the standard one for a great number of matching problems.

Matching combinators. Work on extended matching was originally motivated by its application to transformation, but its potential use goes much beyond to any kind of symbolic manipulation or other formal systems where matching is a central operation.

In the classic concept of constructor-based matching, the left-hand side of an equation (interpreted as a rewrite rule) may only contain constructor terms, as arguments of a single function symbol in functional languages. As an extension, Shi introduced so-called *matching combinators*: they are equationally defined as recursive functions (with some restrictions). Furthermore, the framework is extended to *higher-order* and a notion of constraints ([879, 878, 876, 877]). Prominent examples of matching combinators are list concatenation, map, permutation, substitution, etc. This gives rise to a notion of *extended higher-order abstract syntax* where combinators such as check for subexpression or free or bound variables may be used.

In order to solve the resulting class of equational matching problems, a new kind of equation solving, *structural narrowing*, has been introduced, which is easier to implement than some existing narrowing strategies and has been proved to be a complete lazy narrowing strategy unlike outermost narrowing. The higher-order lifting is promising for efficient higher- order equation solving in practice. For decidability, *proper non-decreasing functions* have been defined.

4.2 Logic Programming

4.2.1 Development of Modular Prolog Programs

As mentioned in Section 2.8, a wide-spectrum language with an institution-independent semantics provides the basis for a program-development framework which accommodates not only different styles of specification but also different target languages. Such a language is Extended ML [836, 844, 848], an extension to Standard ML [462] whereby axioms are allowed in module interface declarations and in place of code in module definitions. In this framework, programs are viewed as axioms of a special form. Thus, if the usual institution is replaced by one in which axioms include at least Horn clauses, the result is a framework for stepwise formal development of modular Prolog programs.

A module system for Prolog based on the one in Extended ML has already been developed [849]. The semantics was obtained directly from the institution-independent semantics of Extended ML as indicated above, although for the sake of readability the general category-theoretic constructions in the semantics were replaced by simple set-theoretic ones. There seems to be no reason why work on formal program development in the Extended ML context cannot be directly translated to Prolog in a similar fashion.

4.2.2 Combination of Logic and Functional Programming

Higher-order equational unification. Qian and Wang present an algorithm consisting of three transformation rules for pre-unification of simply typed λ-terms w.r.t. α, β and η conversions and an arbitrary first-order equational theory E [800, 800]. The algorithm is parameterized by a first-order E-unification algorithm that admits free function symbols. It is proved that the algorithm is sound and complete if so is the first-order E-unification algorithm. The result is relevant to implementations of higher-order proof systems and higher-order logic programming systems. It improves the previous result by Nipkow and Qian [724].

Higher-order patterns are simply typed λ-terms in η-long form where free variables F only occur in the form $F(x_1, \cdots, x_k)$ with $x_1, \cdots, x_k$ being distinct bound variables. It has been proved by Miller that in the simply typed λ-calculus unification of higher-order patterns modulo α, β and η reductions is decidable and unifiable higher-order patterns have a most general unifier. Qian presented a unification algorithm for higher-order patterns, whose time and space complexities are proved to be linear in the size of the input [792, 795].

Higher-order equational logic programming. Higher-order equational logic programming is a paradigm which combines first-order equational and higher-order logic programming, where higher-order logic programming is based on a subclass of simply typed λ-terms, called higher-order patterns. Central to the notion of higher-order equational logic programming is the so-called higher-order equational unification. In [794], Qian extends several important classes of first-order equational unification algorithms to the higher-order setting: only problems of the extensions are discussed and first-order equational unifications are viewed as black boxes whenever possible. The work first extends narrowing and shows that the completeness of many higher-order narrowing strategies reduces to that of their underlying first-order counterparts. Then an algorithm for higher-order equational unification of free higher-order patterns in an arbitrary equational theory is proposed. Finally a general approach to extend first-order unification combination algorithms is sketched informally. The termination property of the above higher-order extensions is considered in a uniform way.

Since there have been many first-order narrowing strategies in the literature, it would be desirable to have a general approach, which can extend these strategies to the setting of higher-order patterns in a uniform way. Liu and Qian propose such an approach based on the syntactic completeness of most existing first-order narrowing strategies [617]. The abstract formulation of the approach clearly shows where and why the approach can or cannot be applied to first-order equational unification algorithms and even general first-order constraint solving algorithms.

The above method may be used to solve goals of higher-order patterns. But the method itself is not a narrowing process. Liu and Qian report other methods to lift first-order narrowing to the setting of higher-order setting, which result in special higher-order pattern narrowing strategies [616].

In [801], Qian and Wang consider the unification of patterns in the presence of some associative commutative (short: AC) function symbols. A pattern is called *AC-pure* if it can be written in the form $\lambda x_1 \cdots \lambda x_k.a_1 + \cdots + a_n$ with $+$ being an AC symbol and each a_i a (free or bound) variable or a constant symbol. A pattern is called *free* if it contains no AC function symbols. They first propose a unification algorithm of AC-pure patterns, which uses a given first-order AC1-unification algorithm. They prove that the algorithm is complete whenever the given AC1-unification algorithm is. A unification algorithm for all patterns in the presence of AC function symbols is then presented as a combination of a unification algorithm of AC-pure patterns and a unification algorithm of *free* patterns. The completeness of the combination is implied by that of its two component unification algorithms, the termination by that of its first-order counterpart.

Extensions of λ-calculus by constraints. In [618] an extension of lambda calculus by logical features and constraints is presented, which yields a minimal core language for constraint-functional-logic programming. A denotational semantics for this language is based on the continuation passing style. The operational semantics of the language is given as a set of reduction rules. In [703] the above mentioned language is used as a target language to compile functional-logic programs. For the compilation scheme soundness and completeness is proved with respect to the narrowing calculus, which serves as an operational semantics of functional-logic programming languages. An abstract machine to execute target language expressions is derived from the continuation semantics.

The aim of the work presented in [648] is the integration of functional programming with constraint programming. The untyped constrained λ-calculus (see [247]) is an extension and generalization of the traditional λ-calculus which consists in the introduction of terms of the form {C}M, where {C} denotes a constraint and M an arbitrary (possibly constrained) λ-term. The operational semantics of the calculus is given by an extension of the usual reduction rules of λ-calculus. Additional rules describe the interaction between constraints and λ-terms. The denotational semantics of this calculus is given (see [647]) and the semantic correctness of the reduction rules is demonstrated. In [248]) the calculus is further extended with existential quantifiers. Existentially quantified variables are only computed by the constraint solver whereas the variables under the scope of a λ-abstractor are computed inside the λ-theory. For this calculus an abstract machine was developed and implemented (see [560]). The last enhancement studied is the disjunction of terms. Using a new operator multiple solutions of constraints can be manipulated. New reduction rules are incorporated in order to handle disjunctive terms. These rules are semantically correct and syntactically confluent (see [191]). It is shown that the calculi are confluent.

4.3 Object-Oriented Programming and Data Bases

4.3.1 Object-Oriented Programming

Object-oriented programming is a programming style which emphasizes the description of data structures instead of the description of algorithms. This style originated with SIMULA-67 [258], in which most of the features of later object-oriented languages can be found. A later well-known object-oriented language is SMALLTALK-80 [423].

Object-oriented programming can be defined as a style in which a system is described as a collection of *objects*. The internal data of an object cannot be accessed directly by other objects; instead, objects interact by calling *methods* (in Smalltalk terminology), i.e. sending messages to each other. Objects are dynamic entities which can be created during the execution of a program, and which can have an internal activity of their own (in parallel with the activities of other objects).

An object is not directly defined but is an instance of a *class*; objects in the same class all have the same set of variables and execute the same methods in response to messages. An important relation between classes is the *inheritance* relation; a class inherits from another its variables and its methods; new variables and new methods can be added, possibly overriding existing variables and methods. In pure object-oriented languages as Smalltalk-80, classes are also seen as objects, which react to messages by creating an instance of themselves.

After a very application-oriented phase during which several software systems have been produced (first the Smalltalk-80 programming environment, which then inspired a whole generation of environments sharing the common idea of a visual interactive object-oriented user interface — SunWindows, the Macintosh Toolbox, Microsoft Windows, etc.), the scientific community has recognized the need for a formal basis for this approach. Work on applications continues, both on the development of object-oriented languages (such as e.g. Eiffel [674], whose aim is to combine object-oriented concepts with a more traditional view of software engineering) and on the analysis of key concepts (e.g. inheritance vs. delegation [613]). From the theoretical point of view, many recent attempts have been made at giving a formalization of object-oriented languages; these include a formal semantics of Smalltalk-80 [951] and attempts [414] at unifying functional, logic and object-oriented programming.

4.3.2 Databases and Knowledge Bases

In the design of databases, information systems and knowledge bases, the conceptual modeling of information to be stored plays an important role. Some earlier work has been done to support conceptual modeling by algebraic tools or other formal methods: the applicative approach of [324, 325, 344], [170] and the modal systems of algebras of [424, 549].

The importance of formal semantics for programming languages was recognized already in the 1960's. Since then, formal methods have had an important influence on the design of new programming constructs and modern languages. In the field of database languages (usually equipped with at least a schema and a query part) it is an exception [892] when a language is defined in mathematically precise terms. More common is the definition of "semantics" by means of examples. The state of the art is more or less intuitive and seldom yet reached the level of precision achieved in the area of programming languages. One of the reasons for the lack of formal descriptions of (especially) relational database languages is probably the fact that classical formalisms like relational algebra or relational calculus are not powerful enough to explain certain features of query languages, e.g. aggregate functions. Therefore a more comprehensive approach is needed.

Translations between database languages have been considered to some extent during the past few years, but only from a concrete implementation viewpoint [177]. For example, the central part of some languages is defined by translation into the relational algebra which is often taken as an implementation basis [193]. But due to the lack of formal semantics and due to the weakness of relational algebra, no systematic approach was possible. It is also often the case that a "user-friendly" language is considered to be only a front end for a more expressive and "harder to use" language, e.g. QBE is often offered as a front end for SQL-systems.

In [871, 865, 302, 301], several aspects of an object-oriented approach to information system specification and design are analysed, covering both structural and dynamic aspects, and the kernel for an algebraic/categorical theory of abstract object specification is provided. These research activities are supported by the ESPRIT BRA Working Group ISCORE (Information Systems: Correctness and Reusability). This project aims at the development of logical and semantic foundations, methodology, languages, and software tools. The semantic domain is provided by a category of objects in the sense of object-oriented programming, where morphisms describe interaction between objects and colimits describe their parallel composition.

Both database (resp. knowledge base) schema and the associated transactions have to be described by rigorous algebraic semantics [867, 868, 348].

4.3.3 Semantic Foundations of Object Orientation

Mathematical models for objects, object relationships, and object constructions have been defined and investigated. The first semantic domain for object-oriented languages and systems outlined in [292] was an amalgamation of two approaches: the objects-as-sheaves approach of Goguen published nearly two decades ago, and the more recent ideas of Ehrich and Sernadas on objects as processes endowed with observation. The basic insight is that objects in the latter sense correspond to object morphisms in the former sense. After an informal introduction into both approaches, first the sheaf model is elaborated, using the term "behavior" for objects in this sense, and avoiding concepts from topology. Behaviors and behavior morphisms are shown to form a complete category where parallel composition is reflected by limits. Objects are defined to be behavior morphisms, giving rise to a cocomplete category of objects where colimits reflect object aggregation. Object morphisms reflect different forms of inheritance, and also object reification (implementing objects over objects) is conveniently expressed in this framework.

The initial work [871] proposes an algebraic approach to object-orientation. Further theoretical investigations about object-oriented concepts in general and especially for information system design have been made [864, 866, 875]. The principal behavior model was a set of life cycles, i.e., of sequences of situations [301, 302, 246, 296]. [298] discussed another model, the menu model. Object relationships have been expressed by appropriate morphisms between life cycles and life cycle sets. Among the object constructions considered were inheritance and interaction, specialization and generalization, aggregation and synchronization [296, 295]. [245] presents a semantic domain for processes with initiative and discusses the relevance for object semantics. [246] presents a categorial semantics for process communication. Subsequent papers ([243, 244]) present further results concerning inheritance even in the presence of overriding. [350] proposes a canonical semantics of temporal object specifications. [294] studies dynamic specialization, i.e., object roles and phases. An institution for a simple logic of behaviour using a cofibration from a category of transition systems into the envisaged category of signatures is presented in [863].

A survey of concepts for system specification and design based on the viewpoint that a system is a community of objects is given in [290]. Objects are self-contained units of structure and behavior capable of operating independently and cooperating concurrently. The approach integrates concepts from semantic data modeling [291] and concurrent processes, adopting structuring principles partly developed in the framework of object-orientation and partly in the area of temporal logics. The approach was presented to scientists working in other fields [296, 295], and to applicants in physics [286].

More recently, a more general object model based on event structures has been investigated [299, 289]. This model captures distribution and full concurrency. The typical example of distributed networks of sequential machines is captured by locally sequential prime event structures.

Temporal Logic. The basic logic adopted for object specification was linear temporal logic [865, 290, 870, 122]. [186] discusses relationship between operational and denotational semantics of temporal logic specification of object behaviour. In [231, 232, 869, 872] the issue of certifying object-oriented specifications has been addressed. [231] introduces a basic calculus for expressing and proving properties of synchronously interacting objects. The objects considered have dynamic behavior and are organized into object communities in a hierarchical way. Syntax and semantics of the calculus are presented in detail. Moreover, a set of basic inference rules is given. Essential properties of the calculus are discussed, especially the question of compositionality. The calculus developed constitutes a basis for investigating issues of verifying properties of objects and object communities. In the paper [234] an elementary deontic action logic for information systems as systems of dynamic objects is sketched. The ingredients which should be included in a logic for abstract object specifications are informally presented. Afterwards a calculus which realizes a basic part of such a logic is proposed. For this calculus a simple proof system is provided which permits to gain experience in verifying object properties. [233] addressed the issue of temporal specification of objects, it provided an approach to compositionality and reusability allowing next operators.

More recently, a distributed temporal logic based on ideas from n-agent logic was defined and investigated [300, 299, 293]. Its interpretation structures are locally sequential prime event structures. [300] studied elements of a specification approach based on distributed temporal logic and a locality principle.

Reification. Refinement techniques have been developed to break down the software design process in pieces with reduced complexity. Reification of object-oriented specification for information systems is connected to different refinement notions such as abstract data type refinement, action refinement and the notion of transactions in data base theory.

First investigations concerning reification have been made in [297, 830, 873, 874]. There a semantic foundation of correct implementation between an abstract object and a community of base objects is studied. An object is viewed as an observed process. Objects and morphisms constitute a category OB in which colimits reflect object aggregation and interaction between objects. The concept of implementation allows for composition, i.e. by composing any number of correct implementation steps, a correct entire implementation is obtained. Two specifc kinds of implementation, extension and encapsulation, are studied in more detail and their close relationship to object morphisms is investigated. The main technical result is a normal form theorem saying that any regular implementation, i.e. one composed of any number of extensions and encapsulations, in any order, can be done in just two steps: first an extension, and then an encapsulation.

Transactions have been introduced in TROLL a temporal logic has been proposed that incorporates the concept of transactions to formalize such specifications, and an appropriate interpretation of specifications that incorporate transactions based on event structures has been given [270, 268, 271, 269].

Recently, refinement notions and results in the theory of process algebra were investigated and applied to the object-oriented framework [510].

Another approach to refinement based on non-sequential automata is given in [666].

Reasoning about object-oriented systems. Object-oriented technology featuring the concepts of object identity, instance creation, encapsulation, dynamic binding and inheritance are increasingly popular when designing systems, but there is a lack of formalisation tools. The common way to formalise object-oriented models is by translating from object-oriented models to models which are supported by existing calculi, i.e., functional models (lambda calculus) or models of communicating processes (e.g., =BC-calculus). Such translations are not ideal as it may not be intuitive how definitions and proofs done in these calculi map to object-oriented models. [728] presents a calculus which is based on object-oriented concepts. Reasoning about object-oriented systems can then be done without changing paradigm.

4.3.4 Object Oriented Specification

TROLL **and TROLL** *light* Investigations have been made concerning database design for specification environments. A conceptual model for the specification language ACT ONE and its environment has been proposed [223]. This has been further investigated in [382]. The paper gives a formal description of suitable database support for an environment supporting the interactive development of ACT ONE specifications. The object-oriented data model of TROLL *light* [235] is used to present the design of such an environment and its dynamic behavior.

In [235, 478] TROLL *light* is introduced as a language for conceptual modeling of information systems. The work about the object oriented specification language TROLL [522] has been a starting point. TROLL is based on OBLOG [349]. It allows for descriptive modelling of complex systems and incorporates concepts from object-orientation, conceptual data modelling, and process modelling. TROLL *light* is a reduced version of TROLL without an inheritance mechanism. It has been developed especially for the purpose of validation and verification of systems specifications. TROLL *light* is designed to describe the Universe of Discourse (UoD) as a system of concurrently existing and interacting objects. TROLL *light* objects have observable properties modeled by attributes, and the behavior of objects is described by events. Possible object observations may be restricted by constraints, whereas event occurrences may be restricted to specified life cycles. TROLL *light* objects are organized in an object hierarchy established by component relationships. Communication between objects is supported by event calling. Apart from introducing various possibilities for the syntactical description of objects, the aim is to describe how the state of an object community is changes with the occurrences of events.

Ideas towards a special formal semantics of TROLL *light* were developed in [267]. The state of a TROLL *light* system is given by the states of all objects, i.e., their attribute values and their process state, which are currently part of the system. System evolution is caused by a set of events which are to be executed concurrently and give rise to changes of several objects. Assuming this interpretation of systems and their evolution an approach by Meseguer is adopted which deals with rewriting systems and concurrency. This approach provides a more general interpretation of conditional rewriting logic, in the sense of a logic of change and action, to come up with a model of concurrency. The approach covers a lot of already known models, and especially it can be applied to the object-oriented subject.

The paper [385] outlines a simplified computational model for TROLL *light*. After introducing signatures for collections of object descriptions (or templates as they are called in TROLL *light*) it is explained how single states of an object community are constructed. By parallel occurrence of a finite set of events the states of object communities change. The object community itself is regarded as a graph where the nodes are the object community states reachable from an initial state and the edges represent transitions between states.

In [390] Entity-Relationship (ER) schemas are translated into the object-oriented specification language TROLL *light*. TROLL *light* incorporates two essential aspects, structure and behavior, in one formalism. By doing the translation from ER to TROLL *light*, the visual advantages of the former is preserved and a formalism is obtained through the latter which can be mapped to an adequate object-oriented database system. Proceeding this way the proposal for transforming ER schemas into TROLL *light* specifications provides a valuable link between structural and dynamic modeling.

[479] outlines a SQL-like query calculus for object-oriented database systems.

An implementation of TROLL *light* is studied in [480]. The techniques for the development of reliable information systems on the basis of their formal specification are investigated. The specification language comes along with an integrated, open development environment [383, 384, 920, 919]. The task of this environment is to give support for the creation of correct information systems. One important ingredient of the environment to be described here in more detail is the TROLL *light* animator.

4.3.5 Inheritance and Reuse

An algebraic characterization of sequential object oriented systems, with main empasis on the notions of inheritance, clientship and subtyping, data abstraction as a structuring mechanism and questions of dynamic creation of objects is given in [139].

Within the Rapide project [680] [545] [546] the interaction of object-oriented formalisms with polymorphic type systems has been explored, resulting in a simple and clean framework for specifying the interfaces of software modules. The type system has made explicit the distinction between subtyping (a relationship between interfaces) and inheritance (a relationship between implementations). The Rapide family of languages has extended the type system to the domain of architecture specifications, and recognized the need for a flexible (though precisely stated) notion of *conformance* [662].

In [428] basic features of object-oriented programming and system design are modelled by a combination of algebraic and projection specifications (= algebraic/topological process specifications). Its primitives are the declaration and state manipulation of objects. As class definitions, algebraic module specifications with import and export interface, parameter and body are used, with module interconnection mechanisms like union and extension to model strict inheritance.

An algebraic formalization of class, general enough to include most object oriented languages, has been investigated to analyze different notions related to inheritance. It is possible to formally distinguish among reuse inheritance, specialization inheritance, clientship and genericity. The last two notions give rise to two basic interconnection mechanisms between two classes [750, 755]. The inheritance relation has been introduced in order to model both monotonic and non–monotonic derivations. The latter have been shown to be simulable by the former via a virtual class. The interconnection mechanisms parameter and import passing have been shown to imply a particular form of specialization and reuse, respectively [754]. An inheritance operator is formally defined [757, 753] which, applied to a class, generates a new class which inherits by specialization from the old one. Basic properties of such an operator form the basis for a reuse methodology for entire class hierarchies, focusing on design reuse rather than code reuse [756]. The hierarchies represent design stages and are a formalization of the knowledge acquired during the analysis of a specific application domain. Being able to reuse whole hierarchies turns out to be important for defining a theory of design reuse [755].

The public interface of a class describes the static properties of objects. The dynamic properties are modelled as configurations of objects, by many-sorted algebras, which evolve through non homomorphic transformations. The structure of the corresponding object transition system reflects the architecture of the interconnections of the classes constituting the software system [758].

Further work on the state dependency aspect of specification techniques for object-oriented systems [780] is discussed in Section 4.4.

4.3.6 Combination of Functional and Object-oriented Programming

Object-oriented programming languages are suitable for describing real-world objects, and functional programming languages for algebraic values. Qian and Krieg- Brückner [797, 798] propose an object-oriented functional programming language, called TOFL, which combines many desired properties of the two paradigms. In particular, TOFL unifies object classes, inheritance, (method) redefinitions and late binding as in object-oriented languages, algebraic data types, higher-order functions, type classes and type inference as in functional languages. TOFL is translated into a stratified and explicitly typed lambda calculus T with overloaded functions, where redefinitions and late binding become late binding of overloaded functions. The operational semantics of T gives a semantics and a simple prototyping implementation of TOFL. In addition, Qian and Krieg-Brückner also studied the type reconstruction problem for a restricted case of TOFL [796].

4.3.7 Combination of Functional and Concurrent Object-oriented Programming

[637] presents an algebraic approach to specifying object-oriented, reactive systems.

An implicitly typed lambda-calculus for objects is developed in [433]. This calculus views objects as extendible case-functions (message interpreters) rather than as extendible records. This view allows to unify the concepts of function, object and process into one concept, that of a functional entity which is self contained and provided with a uniform communication protocol. The calculus is used to give a formal foundation for both sequential and concurrent object oriented languages. In the latter case, objects are viewed as functions communicating asynchronously over unbounded channels.

Maude. In recent years several proposals have been made to extend abstract data type techniques in order to cover the dynamic aspects of object-oriented systems, see Section 4.5.2. A prominent approach is Meseguer's Rewriting Logic as realized in the Maude language [668, 670]. In Maude the functional aspects of a concurrent object-oriented system are described in the algebraic specification language OBJ, whereas the dynamic aspects are modelled by nondeterministic rewriting.

A Maude-case study is presented in [602], where an airport system is modelled by an abstract and a concrete specification, and two different notions of behavioural refinement in Maude are proposed to relate these specifications. Based on Maude and motivated by pragmatic software development techniques an extension of object-oriented nondeterministic rewriting by an algebra of message expressions is given in [948].

In [272] the language Maude is applied to TROLL *light*. Because of the very similar features offered by both languages, a translation from TROLL *light* concepts into the Maude language is presented in order to compare the languages. Apart from presenting the translation, the languages are briefly described and illustrated by examples.

Dynamic data types. The dynamic data-type approach presented in Section 4.5.3 is also suitable for modelling object-oriented systems. In particular, in [7] the model is applied to give a classification of inheritance relations, and a family of object composition operators are proposed in [8], intended to be a first attempt at developing a theory of modules with state. Some other work concerned with applying algebraic or logical techniques to object oriented programming is [668, 750, 3].

The entity specifications of [805] are a formalism which can be used to specify object systems.

4.4 States

The extension of the usual algebraic approach to the case with state is geard towards imperative and object oriented languages, information systems, and databases. There are several ways of dealing with the concept of state using algebraic specifications.

Explicit state approach. In the explicit state approach the state of the system, a value of a state sort, is an additional argument to the operations of the system. An operation changing the state returns a value of the state sort. To model encapsulation, the state of a system should be hidden and only accessible by observer operations yielding values of non state sort [412]. This implies that two states are identified if and only if there is no observation context built from observer operations that can distinguish between both states. there. In general the set of contexts is infinite, which makes it difficult to reason about two states being observably equal. Using context induction of Hennicker [472] is one way to approach this problem. Another approach is to use only finite sets of observation contexts. In [110] Bidoit and Hennicker give sufficient conditions for when an infinite set of observation contexts can be replaced by a finite set.

Implicit state approach. In the implicit state approach the state of a system is an algebra. There are several methods [361, 440, 442, 339, 938, 262, 330, 42, 60, 61, 429] based on this view. These methods differ in the kind of algebra used for modeling the state, for example, total versus partial algebra, and how the operations on algebras are defined.

For example, the evolving algebra approach by Gurevich [442] uses unsorted algebra for the state space and defines a transformation of an algebra A to an algebra B by a set of guarded local function updates of the form **if** b **then** $f(\vec{t}) := t_0$. The semantics is that if b evaluates to true in A then the interpretation of f in B is $f_B(x) = (t_0)_A$, if $x = t_A$, and $f_B(x) = f_A(x)$ otherwise. The interpretation of all other functions remains unchanged.

In the implicit state approach by Dauchy and Gaudel [262] the state space consists of many-sorted algebras. The operations are functors on algebras built from elementary modifiers $\mu\text{-}ac(\vec{p}) = t$ by sequential and indifferent composition and using conditionals.

COLD-K [339] uses many-sorted partial algebras for specifying the state space and Harel's dynamic logic [458] to specify the operations, while Wieringa [938] uses order-sorted algebra defined by equations and equational dynamic logic for the operations.

In Baumeister's approach [60], states are structures in an arbitrary institution and the operations are abstract data-types based on this institution. This allows to define operations using methods for the specification of abstract data-types as, for example, found in [837]. One has the free choice of a specification language for defining the state space. The choice may depend, for example, on the availability of tools for the language. Then the same specification language can be used to specify also the operations. In [61] Baumeister shows how to use this technique to write specifications in a style typically associated with model-oriented specification languages like Z [887], using an algebraic specification language (the Larch Shared Language [446]).

See also *dynamic data-types* in Section 4.5.3.

4.5 Concurrent and Reactive Systems

In this section we consider the use of algebraic specifications of Abstract Data Types (ADTs) within formalisms for the specification of concurrent and reactive systems.

In the literature the specification formalisms for concurrent and reactive systems using, including or integrating algebraic specifications follow three main approaches listed below; for a survey on this topic see [32].

Separate specification of data An existing formalism for the specification of concurrent and reactive systems is extended or integrated with the possibility to specify the data used inside such systems by means of algebraic specifications of ADTs.

Specification of dynamic data Concurrent and reactive systems are considered as dynamic data specified by means of particular kinds of algebraic specifications of ADTs.

Dynamic Abstract Data Types A concurrent and reactive system is modelled by means of an ADT that is dynamically changing, and specified by using appropriate extensions of algebraic specifications.

4.5.1 Separate Specification of Data

LOTOS and PSF are two formalisms following the approach of *separate specification of data* by choosing a process calculus as the formalism for concurrency. The differences between them are in the formalism for the algebraic specification part (ACT ONE [311] for LOTOS and ASF [64] for PSF) and in the combinators of the chosen process calculus (inspired by those of CCS for LOTOS and by ACP for PSF).

PSF (see [657]) is the process specification formalism developed by Mauw and Veltink as a basis for a set of tools to support the process algebra of Bergstra, Klop et al' (see e.g. [65, 48]). The basic specification formalism is equational logic with total algebras. The theory and language of ASF is adopted for handling modular and parameterized specifications.

LOTOS has probably been the first internationally known algebraic specification formalism for concurrency (see [187, 143]); most importantly, it is an official ISO language specification for open distributed systems, a qualification that alone would rank it high in an ideal value scale of possible important applications.

LOTOS uses ACT ONE ([311]) for the ADT specifications and a process description formalism based on an extension of CCS with several derived combinators (e.g. input/output of structured values, sequential composition with possible value passing, enabling/disabling operators). The basic specification formalism (equational logic with total algebras) is the same as in PSF and also the bisimulation semantics for processes.

All these years, LOTOS has been used in several practical applications. Moreover, a toolset for helping to write correct LOTOS specifications has been developed (see e.g. the ESPRIT project LOTOSPHERE [911]). Recently a revised version of LOTOS (E-LOTOS, for Enhanced LOTOS) is under way, see [620], taking into account the needs raised during its application in industry.

Algebraic Petri nets are particular nets where the tokens are given by an algebraic specification of an ADT and where arcs are labelled by terms with free variables and transitions by formulae built on the signature. There are several proposals for algebraic Petri nets, among them Reisig's work [816], the work of Bettaz and others [89, 90, 91], and the work of the Milan group on OBJSA nets [52]

4.5.2 Algebraic Specification of Dynamic Data

There are two main instances of this approach: the work of the Genova group based on the idea of defining labelled transition systems by means of algebraic specifications of ADTs, and Meseguer's Rewriting Logic.

Algebraic specification of labelled transition systems. A labelled transition system $(S, L, \rightarrow)$ (where S and L are two sets and $\rightarrow \subseteq S \times L \times S$) can be viewed algebraically as an algebra with two sorts s and l and a predicate symbol $\rightarrow : s\ l\ s$; such algebras are called *dynamic algebras*. Notice that now the states and the labels are data structures modelled by algebras instead of simple sets.

Various results and techniques developed in the field of algebraic specifications can be specialized to the dynamic case, where dynamic algebras are used instead of algebras, getting conditional algebraic dynamic specifications with initial semantics, first order algebraic dynamic specifications with loose semantics, a notion of correct implementation between specifications, and so on. [26] presents a form of observational semantics for algebraic specifications apt to express bisimulation semantics for algebraic dynamic specifications; [36] is a complete presentation of algebraic dynamic specifications.

First order is not adequate to express all axioms for loose algebraic dynamic specifications (i.e. those concerning relevant properties of a dynamic activity, such as liveness/safety properties); so several extensions have been proposed: in [241, 242] the logic is extended with the combinators of the CTL branching time logic; in [805] with the possibility to express properties about the distribution structure of systems; finally, [31] presents a specification language for concurrent and reactive systems incorporating the above extensions.

In [806, 33] the logic has been extended instead with the possibility to express properties of activities in terms of abstract events, i.e. partial activities that are not just simple transitions; using this logic it is possible to formalize properties of reactive systems without fixing the grain of the atomic transitions.

SMoLCS. Algebraic dynamic specifications provide the foundations for a specification method for reactive and concurrent systems called SMoLCS, documented in [807, 760] [882, 28]; while some of its applications to case studies are in [809, 808] [34].

Rewriting Logic. In [669, 654] and, as application to object systems in [668, 670], Meseguer presents a formalism called "Conditional Rewriting Logic". Rewriting Logic specifications correspond to a subclass of algebraic dynamic specifications with initial semantics, but where the transitions are not labelled; the specification of the static and of the dynamic properties are completely separated and axioms for propagating transitions by congruence are assumed ("rewriting" comes in here). However, there is no provision for observational semantics and it is difficult to see how it can be achieved. Indeed, the absence of labels and the propagation of transitions by congruence are a considerable simplification giving Rewriting Logic a pleasant look, but are a major drawback with respect to modularity and the definition of sensible observational semantics. It has been remarked by many people that rewriting logic only seems to work well for closed systems and in connection with its support language Maude [673], which is well structured and enjoys a very efficient implementation.

Timed rewriting logic. In [563], [564] timed rewriting logic is introduced, which is an extension of Meseguer's rewriting logic to cover real-time aspects of dynamic systems. Timed rewriting logic is used to define a timed variant of Maude called Timed Maude. Timed Maude has been validated in the case study of the Steamboiler Specification Problem [730].

4.5.3 Dynamic Abstract Data Types

A new semantic framework for dynamic systems has been developed that generalizes the imperative case, which was considered first. In [38, 39] an algebraic model (*left/right structures*) is presented that is suitable for modelling, in a clean and uniform way, structured data types in imperative languages. In [962] it is shown how to formulate the usual notion of implementation of data-types in the imperative context.

*Dynamic (abstract) data-types*is the terminology proposed, in general, for formal models of dynamic systems based on the *state-as-algebra* view of systems, in which states are algebras and dynamic evolution is given by transformation of algebras (see e.g. [261, 330, 42, 429, 60]). These models are to be distinguished from algebraic models of dynamic systems that remain, essentially, within the usual algebraic framework, modelling states as (possibly special) values (*state-as-term approach* or *dynamic-data types*, stressing that in this case data, i.e. values, may represent dynamic entities, while in the state-as-algebra approach it is the overall data-type that becomes dynamic). The state-as-algebra view is the same in *evolving algebras* (see e.g. [441]); however, these have been introduced for the sole purpose of supporting a method for defining an operational semantics of programming languages, without any effort in developing a clean theory of structures and constructions.

The model part of the semantic framework is introduced in [41, 42]. A parameterized institution of dynamic data-types is presented in [961]. In [43] it is shown that it is possible to extend the notion of term to the dynamic case in a natural way.

In [330] an informal proposal for the concept of Dynamic Abstract Data Type (DADT), related to the above, is provided. Such a concept is an extension of abstract data types towards the modeling of dynamic behavior of complex state dependent systems, e.g. object communities. A DADT specification consists of four layers. The first two are algebraic specifications with initial and loose semantics, respectively. They model the underlying static data types of a dynamic system (initial semantics of level one) and the class of admissable states or instant structures (loose semantics of level two, including constraints). The third level describes the dynamic operations whose semantics are state transitions, given by transformations of (level two–) algebras. The fourth level models aggregations of states and their creation and manipulation.

An application of this approach, intended to model object–oriented systems, has been presented in [780]. Objects are attached to values by a partial function that characterizes the state. State transitions are given by manipulations of this partial function that are induced by the operations of the underlying static data type. Further developments concern the description of object sharing and object communities.

In [429, 430, 431] a concrete version of the concept of dynamic abstract data types is worked out, based on partial algebras as static data types and states, and a constructive approach to state transitions specified by conditional parallel assignments. States correspond intuitively to environments, that is partial functions from dynamic entities to current values. The approach is completely algebraic: First syntax, semantics and correctness for all parts are worked out completely. Then it is shown that the specification framework defined in this way has initial semantics, a sound and complete deduction calculus, free functor semantics for parameterized specifications, and allows arbitrary composition of specifications via colimits with compositional semantics. Applicability of the developed theory is shown by example specifications of programs from existing programming languages that cover aspects of parallel and object oriented programming.

4.5.4 Special Approaches

Sheaf theory. [217, 642] develop the relationships between final semantics and operational semantics embodied in transition systems for the object paradigm, by representing systems as diagrams of transition systems with algebraic structure. This approach uses sheaf theory to give a topological approach to system structure. The relevance of sheaf theory to concurrency and the object paradigm is first set out in [395] and applied to the semantics of an object oriented language in [952].

State-based algebras The research in Dresden is focused on algebraic specifications of state based systems. The basic idea is the algebraic interpretation of maximal fixed points of recursive type equations in terms of co-algebras. Whereas minimal fixed points of type equations are represented by initial algebras, maximal fixed points yield terminal co-algebras. As demonstrated in [814] co-algebras may be seen as algebraic formalizations of automata, processes, transition systems or objects in the sense of object-oriented programing and terminal co-algebras as classes of behaviours of those structures, where behaviour is formalized as equivalence class modulo strong bisimulation.

This formal view of process behaviour permits the use of the very powerful tool of coinduction for the manipulation of behaviours. In the same way as initial algebras represent primitive recursion, and in case of equationally partial algebras the whole range of recursion, the resulting coinduction can be used to define operations that process behaviours. In contrast to CCS or CSP there is no need to fix the set of process building operations. This approach guarantees complete freedom in choosing the appropriate operations and gives a formal framework for analyzing the properties of the chosen operations on behaviours.

Causality. In [663] a number of optimization techniques are given for dealing with the notion of causality in reactive systems. For certain architectures, it is shown, how one can reduce the overhead (both operational and in the size of supporting data structures) required for keeping track of causal dependencies in reactive systems. This is a prerequisite for a successful run-time monitoring of reactive systems w.r.t. causality-oriented specifications.

In [903], the behavior of reactive systems is discussed in terms of a black-box, compositional approach to concurrent system specification. It is pointed out how the notion of causality becomes problematic as process models are introduced that cater for infinitely executing processes.

Focus. The research group at the TU München developed Focus, a framework for the systematic formal specification and development of distributed interactive systems and their components. Focus provides models, formalisms and verification calculi for the stepwise specification and development, transformation and verification of such systems. Focus aims at the modular development and implementation of distributed interactive systems through several abstraction levels by stepwise refinement. In the models of Focus, components and networks of components are represented by sets of stream processing functions. An informal introduction to Focus can be found in [152]. An annotated biblocgraphy of case- studies performed in Focus can be found in [151, 157]. Finally, a formal language definition is given in [853].

The formal foundation for Focus is discussed in several papers [145, 147, 146, 149, 889]. An assumption/commitment style is presented in [148, 891]. A relational specification style is given in [167, 168]. A formal foundation of a kernel sublanguge of SDL in Focus is given in [504, 890]. Hardware specification is treated in [357, 356]. A calculus for mobile data-flow networks is given in [433, 150]. A fully abstract, denotational model for mobile, timed, nondeterministic data-flow networks whose components communicate in a point-to-point fashion is given in [439]. In this model the stream processing functions are time guarded and generic. Time-guardedness assures unicity of fix-points and genericity assures mobility. The relation between Focus agents and automata is studied in [438].

Algebraic specifications and graph grammars. The algebraic approaches to graph transformation [621, 303] and the field of algebraic specification have a common mathematical background in algebra and category theory. From an applicational point of view, they share their orientation to system specification. Hence there are a number of interrelations, both on the mathematical and on the conceptual level.

Using graph transformation, a system is described as a collection of states, modeled by graphs, and state transformations, specified by graph rewrite rules. This approach is particularly well-suited to model the structural part of a system, and its development. In the framework of attributed graph transformation (see [622]) graphs are attributed with algebraically specified data types in order to store and manipulate data values. Applications for such a combination are specifications of concurrent and distributed systems, where data types and dynamic behavior are described within a single framework. In [922], for example, parts of the semantics of the object-based specification language TROLL *light* are described using attributed graph transformations based on partial algebras. In [562] attributed graph transformations are used as a model for concurrent object oriented systems, and a corresponding process semantics is developed. In [326] attributed graph transformation systems are presented as an instance of dynamic abstract data types [330], where the dynamic operations are defined by graph transformation rules.

Algebraic high-level (AHL) nets [332] are a combination of algebraic specifications and Petri nets. This integration allows the modeling of abstract data types with a dynamic behavior in cases where the state of a system is given by a multiset, rather than a graph. The horizontal and vertical structuring of AHL nets is described in terms of high-level replacement systems, a categorical generalization of graph transformation systems. The results of this integration can be found in in [332] and [818]. A survey is given in [749].

4.6 Semantics of Programming Languages

Algebraic specifications can be successfully applied to specify the semantics of programming languages. This is a worthwhile approach since it permits a proper description of the meaning of programming languages with all the advantages algebraic specifications have when they are used for the description of data structures. They allow a model-independent, property-oriented description of programming languages without forcing the understanding and knowledge of a particular mathematical model.

In the early years, work within the area of the semantic description of programming languages by algebraic techniques (apart from the field of concurrency already mentioned — see Section 4.5) was mainly done by Mosses under the keyword of *abstract semantic algebras* [691]. Further work has been done by the ADJ group in the course of an attempt to provide a foundation for the application of algebraic specifications to compiler construction [899], by Wand in giving algebraic descriptions of functional languages [934] and by Broy and Wirsing, and also by Pepper, in giving an algebraic theory based on the concept of partial algebras for a semantic specification of programming languages [769, 169, 160, 164]. This work has shown clearly that a purely algebraic description of programming languages is possible and helpful in analyzing programming languages and their semantics.

4.6.1 Semantics of Ada

In recent years a very challenging problem has been to define the semantics of languages with strong interference from sequential (imperative) and concurrent constructs, Ada being the most important representative in this class (see [25]). By using the algebraic techniques for the specification of concurrent systems developed in the SMoLCS approach (see Section 4.5.2), it was possible to give the first complete formal definition of Ada [25]. The foundations of this approach to semantics are in [29] and [30]. Two notable points are that the overall definition of Ada is just one (highly structured) algebraic specification, with an appropriate semantics; moreover the special meta-language devised for the definition can be translated into the kernel algebraic specification language ASL [851, 941].

Early attempts to extend current techniques to deal with object-oriented languages were [414] and [140]; cf. also Section 4.3.1.

4.6.2 Action Semantics

Action Semantics [694] is a framework for specifying (the semantics of) programming languages. It has good pragmatics, making specifications easily accessible to programmers. Thanks to its inherent modularity, action semantics scales up smoothly to describing practical, industrially-useful languages [455].

The relevance of action semantics here is that the data types used in action semantic descriptions are specified algebraically, in the framework of unified algebras; likewise for abstract syntax [698]. Moreover, the action notation used in action semantics provides a smooth combination of various paradigms of computation: functional, imperative, and reactive [695, 701]. The algebraic and operational semantics of action notation are also specified using unified algebras. The ASD Tools developed for action semantics can be used generally for specifications in the framework of unified algebras.

An up-to-date overview of the theory and practice of action semantics [700] and a tutorial on action semantics [699] are available. A workshop on action semantics [697] was partially supported by

COMPASS.

4.7 Specification Languages

4.7.1 Early Algebraic Specification Languages

Existing specification languages can be classified according to their underlying logic, semantics or structuring mechanisms. Particularly interesting examples of such languages are briefly described below.

CLEAR [183, 184, 833] was the first algebraic specification language and its influence can be seen in most of the languages developed later. It offered for the first time the now standard operations of combination, enrichment, derivation and parameterization. Its semantics was loose, model-oriented, defined by means of constraints. Its static semantics provided a notion of "environment" that offered a reasonable solution to scoping. CLEAR was the first institution-independent language (see Section 2.8).

LOOK [334], a language designed at IBM and in some sense an ancestor of ACT ONE, had two main aims: simplicity, and compatibility between the semantics at the model and the theory levels. Simplicity was achieved by providing powerful enrichment, combination and substitution operations that would allow explicit parameterization and parameter passing to be avoided. This idea can be found later in LARCH. The second aim, compatibility between the two levels of semantics, was not fully accomplished since no correctness conditions were obtained for the specification building operations. Recently, in [739] these conditions were finally obtained and exploited in the definition of the GSBL language [226].

ASL [851], [941] had as its main aim to provide very powerful specification building operations, in such a way that the language could be used as a *kernel* to design other languages. Because of its aims its semantics is model-oriented. In [843, 837] it was shown that operations similar to those in ASL can be provided in an institution-independent framework. Extended ML [836], and the meta-language for the semantics of Ada [25] are defined in terms of ASL.

ASF [68] is based on a pure initial algebra approach. It provides export, import and renaming as specification-building operations. The semantics of these operations is defined axiomatically using module algebra [67]. The specification part of the wide-spectrum language COLD developed within the ESPRIT project METEOR is strongly influenced by ASF [518].

CIP-L [54] is a wide spectrum language that was particularly designed for use within the transformational program development methodology [165, 771, 770]. It offers constructs for all levels of program development, ranging from specification constructs via applicative constructs to notations for procedural and machine-oriented programs. For the purpose of formal specification, CIP-L offers a sub-language for algebraic specifications, as well as particular constructs such as set comparison, quantification, choice, and a mechanism for formulating *descriptive* expressions. The characteristics of the CIP-L sub-language for algebraic specifications are (cf. [153, 55, 54, 59, 949]) that it provides parameterized, hierarchically structured specifications, partial operations, first-order axioms and a loose semantics.

PA^{nn}dA-S was the specification sub-language of the PROSPECTRA project [587, 581, 582, 585, 491, 138]. It has the same characteristics as CIP-L, extended by higher-order functions, non-strict functions for concurrency with streams, etc., and derives its modularization constructs from the target language Ada. The specification language is also used as a meta-language to develop efficient transformation algorithms and to formalise the development process [583, 585].

ACT ONE [311, 328, 218] is based on the concept of parameterized specifications which includes usual algebraic specification with initial and loose semantics as special cases. The structuring mechanisms for building up larger specifications from smaller pieces are extension, union, renaming, and actualization.

LOTOS has probably been the first internationally known algebraic specification formalism for concurrency (see [187, 143] and Section 4.5.1). LOTOS uses ACT ONE for the algebraic specification of data and a description formalism based on an extension of CCS for processes. A toolset (see e.g. the ESPRIT project LOTOSPHERE [911]) has been developed and a revised version of LOTOS (Enhanced LOTOS) is under way, see [620].

ACT TWO [345], the successor of ACT ONE, is mainly based on algebraic module specifications [120], but includes also basic units for algebraic specifications with loose semantics and constraints, called *requirement* specifications, and for parameterized specifications with initial (functorial) semantics. The structuring mechanisms in ACT TWO are similar to those of ACT ONE but are also defined for module and requirement specifications. The semantics of ACT ONE and of ACT TWO is totally defined on two levels: the first is that of theories, and the second is that of algebras and functors. Under suitable semantical context conditions, the semantics is compositional and global correctness is implied by local correctness.

LARCH [445] is not really a language, but a family of languages. The so-called LARCH *shared language* is what we may call an algebraic specification language. The main aim of this language is to provide operations for incremental construction of specifications. In order to do this it provides specification building operations similar to those in LOOK [334].

Extended ML [836, 844, 848] is not tailored to any particular logic. It adopts the modularization mechanisms of the Standard ML programming language [462], and its semantics is largely expressed in terms of the primitive specification-building operations of the ASL kernel specification language. Extended ML is a wide-spectrum language in the sense of CIP-L, as Standard ML function definitions are just axioms of a certain special (and executable) form.

PLUSS [366, 93] is the result of a broad range of experiments in writing large specifications and the treatment of specific institutions within PLUSS has been demonstrated in a series of papers [276, 98, 116], [98, 96, 214, 104]. The main originality of PLUSS is to maintain a careful distinction between *completed* specification components and specification components *under design*. A completed specification component has the property that its class of possible implementations is fixed. In practice this means that such a specification component is either already implemented or may be implemented without taking regard of its context (for instance, of the other components of a specification in which this completed specification component is used). The semantics of PLUSS is expressed in terms of the ASL kernel specification language.

OBJ3 [421] (cf. OBJ2 [358]) is a specification and logic programming language based on order-sorted algebras and logic [413, 884]. Within this framework, many of the problems with partiality and polymorphism can be solved. OBJ3 inherits its structuring mechanisms, parameterization and *views* from CLEAR.

OBSCURE [604] is based on module specifications. Each module is interpreted as a function mapping algebras into algebras. Apart from the classical structuring mechanisms, OBSCURE contains operations for the explicit construction of sub-algebras and quotient algebras.

LPG [83, 82] is a specification language combining algebraic and logic style of programming. Operators are defined by conditional rewrite rules and predicates are defined by Horn clauses with equality and disequality. Some specifications (properties) are loose, protecting imported types, and other ones (types and enrichments) are freely generated from the semantics of the parameter part (free functor semantics). Specification morphisms can be declared. They play an important rôle to generate the space of potential available specifications and to determine the sharing of sub-specifications [87]. The operational semantics is based on resolution and narrowing [84]. Among several case studies, LPG has been used for the specification and animation of standards [266].

4.7.2 Recent Algebraic Specification Languages

Maude is based on Meseguer's Rewriting Logic (see Section 4.5.2). In Maude ([668, 670], cf. Section 4.3.7) the functional aspects of a concurrent object-oriented system are described in the algebraic specification language OBJ, whereas the dynamic aspects are modelled by nondeterministic rewriting.

COLD-K [339] uses many-sorted partial algebras for specifying the state space and Harel's dynamic logic [458] to specify operations on the state space.

SPECTRAL [590] is a specification language design that tries to maintain compactness in spite of a number of orthogonal concepts. The design is based on Extended ML and PA^{nn}dA-S of the PROSPECTRA project [491], generalising and extending both approaches. Of particular concern are the means for structuring specifications and programs in-the-large and in-the-small. The language includes constructs for defining general higher-order functions yielding specifications or program modules. Concepts of subtyping and inheritance are included to support the specifi-cation development process and to enhance re-usability.

SPECTRUM is a specification language developed at TU München . It is based on algebraic /axiomatic specification techniques (with the concept of loose semantics) and is oriented towards functional programs. SPECTRUM includes the following features:

- full first-order predicate logic with induction principles,
- partial functions, definedness logic and fixed point theory,
- higher-order elements and typed lambda-abstraction,
- non-strict functions and infinite objects,
- predicative polymorphism with type classes,
- parameterization and modularization.

An informal introduction explaining the objectives and concepts of SPECTRUM by using simple, well-known examples is given in [155, 156]. The language definition, the semantics and the calculus are given in [434], [436] and [804], respectively.

SPECTRUM incorporates the entire notation for typed λ-terms. Therefore SPECTRUM supports partial and non-strict functions as well as higher-order functions. The models of SPECTRUM specifications are certain continous algebras [437, 436]. SPECTRUM uses three valued logic and also allows a restricted use of non-continous functions for specification purposes.

An extension of the semantics of SPECTRUM that supports higher-order non-continous functions was encoded in the higher-order logic theory HOL of the interactive theorem prover Isabelle [803].

ABEL has been developed at Oslo University. Distinctive features are:

- Parameterized types and subtypes (syntactic and semantic), also object oriented classes [253, 259, 745, 254, 255, 260, 50].
- Modules parameterized explicitly by types and implicitly by loose functions (renameable).
- Loose specifications by first order axioms.
- Constructive specifications are based on generator inductive definitions of types and functions. Definition of existing loose functions trigger proof obligations.
- Function definitions by terminating generator induction, TGI, give rise to convergent rewrite rules and to covariant profile families based on syntactic subtypes, syntactic totality checks, as well as mechanically defined well-definedness conditions [254, 260].
- Extension to higher order types and functions is planned [257].

4.7.3 The Common Framework Initiative.

A collaborative effort, the so-called *Common Framework Initiative for Algebraic Specification* (CoFI[1], see [702]), was started some time ago by the COMPASS WG and IFIP WG 14.3 (Foundations of System Specification) and has been in full swing since Autumn 1995, with the goal of having a first result in Autumn 1996. To have most impact, it is intended as an open international effort, driven by the European COMPASS community but involving people beyond it. Some 25 leading researchers in algebraic specification have already agreed to participate, and others are being invited to join.

The rationale behind this initiative is that the lack of such a framework greatly hinders the dissemination and application of research results. In particular, the proliferation of specification languages, some differing in only quite minor ways from each other, is a considerable obstacle for the use of algebraic methods in industrial contexts, making it difficult to exploit standard examples, case studies and training material. A common framework with widespread support throughout the research community is urgently needed.

Quite important for the success of CoFI is the coverage of most concepts of existing specification languages on the one hand, but, more importantly, uniformity and simplicity. The key idea here (and the challenge for the design) is not to have one huge specification language, but to impose some structure such that a family of specification languages emerges. It is to consist of three main levels:

The Common Language (CL). This general-purpose specification language is to be the most prominent of the CoFI family: the other languages are to be obtained by restriction or extension of it. CL should be at least as expressive as various existing languages. Its intended applicability is for specifying abstract data types, and requirements and design for programs (at least for programs which do not react with their environment during execution). It is also to be the main topic of the documentation (reference manual, user's guide, text book).

Some Sublanguages (SLs). These are to be obtained from CL by imposing syntactic and/or semantic restrictions. The sublanguages need not have a common kernel. Their intended applicability is in connection with particular kinds of tools (e.g., a first-order equational SL might support tools for term rewriting and completion).

Some Extended Languages (XLs). These are to be obtained from CL (or perhaps from SLs) by syntactic and/or semantic extensions. The extended languages need not have a common super-language. Their intended applicability is to include advanced features well-adapted to specifying requirements and design of systems involving various programming paradigms (e.g., reactive systems, object-oriented systems).

[1] CoFI may be pronounced like 'coffee'

Chapter 5

Methodology

The goal and the dream of the algebraic specification approach is the conceptual and methodological support of all levels and major aspects of the system development process as outlined, for example, in [310], [313], and [572]. In this respect, topics of interest concern development phases, formalisation of developments, development graph and reusability.

Work in [35] presents a two-rail approach to the development of formal specifications of reactive and concurrent systems. First an informal specification is given, consisting of natural language text structured in a precise way. From this a strictly corresponding formal specification is derived. Following this approach the task of giving the formal specification is made easier by splitting it in two phases. Moreover, the informal specification is a help to understand the formal one, especially for people not used to formalities. An application of this method to a case study can be found in [34].

5.1 Development Phases

It is widely recognized that formal development of correct programs cannot be undertaken without proper formal support. The algebraic approach to program development has good formal foundations and provides a rigorous approach to program development. Algebraic techniques may in particular serve as a formal support for the early phase of a software project (specification). Moreover, it is well-known that the decomposition of large systems into adequate modules is supported by the concept of abstract data type which is basic in algebraic specifications.

The algebraic framework provides formal criteria for the verification of development steps, and equational logic supplies powerful tools for machine-supported deduction and therefore for rapid prototyping of specifications, proofs and tests. Algebraic methods support program development from formal specification to efficient implementations: program transformations are usually based on algebraic techniques, efficiency oriented and control flow oriented concepts can be modeled in an algebraic framework and supported by algebraic techniques.

The algebraic approach has been widely used and intensively studied within the CIP project where a methodology for program development by transformations has been developed (cf. e.g. [56], [57], [161], [162], [946, 171], [54], [767]). The ESPRIT project PROSPECTRA is based on this work ([587], [581], [585]). It plays also a crucial role within the ESPRIT project METEOR where algebraic specifications form the basis of an object-oriented methodology of program development. In the specification language PLUSS [367], [94, 95], linguistic support is provided for the development of algebraic specifications.

The formal notion of implementation of algebraic specifications and the correctness criteria for such implementations form a basis for the stepwise construction of programs. A particular way of developing a program is to compile a specification of a certain form directly into efficient code [538].

The development of a methodology for "algebraic programming" [456] is needed on all levels of the software construction process: development of formal specifications from requirements; analysis and validation of formal specifications; transformation and implementation of specifications via specifications of a more concrete form for which the transition to efficient programs is possible. This methodological level should be supported by two other levels: a conceptual level on which the design of relevant structural properties of algebraic specifications such as hierarchical consistency or sufficient completeness are studied, and a technical level which provides verification techniques for algebraic specifications such as induction proofs, normal form proofs or Knuth-Bendix completion. All these points are addressed in Chapters 3 and 6.

The approach in [388] reports on successful application of algebraic ideas to the formal development of information systems. It describes a formalism, i.e., a language, for the specification of information systems [385, 389], a method for the construction of specifications in this language [388], and implemented and planned parts of a specification environment covering important phases of the software development process [383, 384].

Various authors have stressed the need for a formalization of the software development process: the need for an automatically generated transcript of a development "history" to allow re-play upon redevelopment when requirements have changed, containing goals of the development, design decisions taken, and alternatives discarded but relevant for re-development [939], [854]. A development script is thus a formal object that does not only represent a documentation of the past but is a plan for future developments. It can be used to abstract from a particular development to a class of similar developments — a development method, incorporating a certain strategy. Approaches to formalize development descriptions involve a form of development program [939], regular expressions over elementary steps [888], functional abstraction [342], and composition of logical inference rules [880], [514].

In the PROSPECTRA project [587], [581], [585], an elementary development step is the application of a single program transformation rule. A development script is then a sequence of rule applications. The question is how best to formalize rules and strategies. The approach taken in [583], [138] is to regard transformation rules as equations in an algebra of programs, to derive basic transformation operations from these rules, allowing composition and functional abstraction, and to regard development scripts as (compositions of) such transformation operations.

Using all the results from program development based on algebraic specifications, we can then reason about the development of transformation programs or development scripts in the same way as about programs. For example, we can reason about development goals as requirement specifications for transformation operations in the syntactic algebra and characterize them as structural normal forms; we can implement transformation operations in various ways and optimize them using algebraic properties; we can use composition and functional abstraction; in short, we can develop correct, efficient, complex transformation operations from elementary rules stated as algebraic equations. Moreover, we can regard development scripts as formal objects, i.e. as (compositions of) such transformation operations. We can specify development goals, implement them using available operations, simplify development terms, re-play developments by interpretation, and abstract to development methods, incorporating formalized development tactics and strategies. The abstraction from a concrete development to a method and the formalization of programming knowledge as "transformation rules + development methods" will be a challenge for the future.

5.1.1 Applications Within the Development Process

As previously discussed, algebraic specifications can be useful not only at the specification stage, but also at almost all stages of the usual software development process. Uses include prototyping (see Section 6.4) and proving correctness of programs. Algebraic specifications can also be used as a basis for functional testing of programs.

A theory of software testing, based on formal specifications, was proposed by Bougé, and then applied to algebraic specifications. The basic ideas are the following [124]:

- Given the axioms of an algebraic specification, which are equations (possibly positive conditional), the left-hand side and the right-hand side of each equation are evaluated using the program under test, for "interesting" values of the variables. The success or the failure of the test is decided by checking that both results are equivalent.

- the structure of the specifications is used to guide the choice of the test data (i.e. the "interesting" values above). For instance, a complexity measure can be defined on some sorts, every premise of each equation must be valid at least once, etc.

Recently, some coverage criteria for algebraic specifications have been defined [370]. These criteria induce incremental test strategies, where the limit of the testing process is a correctness proof of the program.

These testing strategies can be expressed formally. Thus, there is some hope that they can be automatized using theorem proving techniques. At this moment, experiments are performed using logic programming and its extensions (SLOG [354], RAP [372]).

In [206] an approach to integration testing with prototyping was proposed: starting with a hierarchical and modular executable specification of the system to be realized, implemented modules are progressively integrated in place of the corresponding specified modules. Performing an execution of such a system, where the integration is partial, requires mixing evaluation in the specification language and in the programming language used in the implemented modules.

5.1.2 Software Modularity

Modularization is one of the main principles in software development. The aim is to divide the task of developing a large system into modules which can be developed independently of each other.

In an abstract view, a module comprises three main components: its interface, its internal construction, and its externally-visible behaviour. These components of a module are given syntactically as well as semantically, and form a conceptual unit. Modular systems are comprised of modules which are composed in a particular way using interconnection mechanisms and combinators provided by the modularization framework in use. The design of a module specification, the choice of modularization, and the design and development of a modular system are tasks which can never be fully mechanized or formally solved [336]. Such tasks require skill, experience and imagination. But because of this it is important to have support in various ways.

The basic questions for a theory of specification of modules and modular systems concern mainly the following topics:

- Concept of module specifications and modular systems adequate for software system design and development;
- Correctness of module specifications;
- Compositionality of module operations;
- Equivalence of different modular structures of a system;
- Implementation and refinement of modules and modular systems.

Various module concepts, which are described in Section 3.3, have been designed in the field of algebraic specification to cover various phases of software development, fulfilling the above criteria to a greater or lesser extent.

5.2 Formalisation of Developments

In [662] *conceptual architectures* [885] are discussed, identifying the need for multiple perspectives on any software architecture, using conceptual decomposition to improve the clarity of specifications.

The role of formal, algebraic specifications in the development of knowledge based systems is discussed in [914, 915], using a large market-prediction knowledge based system as an example.

Refinement is the process of moving from one specification to another, more concrete, specification which displays the same behaviour, and fits into an incremental approach to moving from an initial specification to a final implemented product. The phrase 'more concrete' is generally understood to refer to a specification which can, in some sense, be more efficiently or more directly implemented. The requirement that the concrete specification display the same behaviour means that all models of the concrete specification are implementations, i.e., models, of the original, abstract specification. A powerful proof technique for proving correctness of refinement of hidden sorted specifications is given in [401, 643]. Because hidden sorted algebra is based on the notion of equality *up to observability*, it allows refinements that make use of subtle changes of representation, where correctness proofs can often be non-trivial. However, the proof techniques seem to lead to very simple and tractable proofs; for example, they have been used to prove the correctness of a rather sophisticated optimising compiler for OBJ3 [454, 453].

5.2.1 Proof obligations

In [834], the generation of proof obligations in EML-like formal development frameworks are studied. The paper gives an SOS-style semantics for formal development in a skeletal version of Extended ML, in which proof obligations associated with development steps appear explicitly in the rules. It is possible to prove that any *development script* which succeeds according to this semantics yields a modular system which is correct with respect to its initial specification. This formalizes earlier work on proof obligation generation in EML [848].

5.2.2 Development of Correct Software

Transformation is a theoretically well-founded (cf. [772]) and methodologically well-understood technique. It allows the reformulation of specifications into equivalent ones and the conversion of specifications into implementations that are correct by construction. Methods for the transformation of specifications are built on notions of refinement. Specification transformations for specific basic parameterized specifications (records, arrays, etc.) have been studied in [764], [600], [761], [763].

At the level of specification languages, basic laws for module expressions have been developed in [851] for ASL. A related concept is the *module algebra* of [67]. Laws for module expressions have various operational aspects. One is that of transforming (normalising) module expressions by applying the laws as rewrite rules. In [909] a variation on module algebra is proposed in which the basic modules can be imperatively implemented. Another operational approach is the $\lambda\pi$-calculus of [341]. The latter paper is also an application of these ideas to class structures in the sense of object-oriented programming.

Another goal is that of transforming a non-executable specification into an executable one. This goal can be formulated as follows: for a given input vector x satisfying an input condition C, find an output vector z (i.e. the output of some program $P(x)$) satisfying a given input/output relation $R(x, z)$. If R is a logical formula, then the above specification can be considered as a formula of the form

$$\forall x. C(x) \supset \exists z. R(x, z)$$

called the *specification theorem* (ST). In this case, one way to go from a non-executable specification to an executable one is by proving ST using mathematical induction and extracting a program from the obtained proof. This is known as the *deductive approach* to program synthesis from formal specifications [649].

In logic, there are well-known results about the links between an inductive proof of ST and the recursive definition of the Skolem function P extracted from this proof [861]. Sato [852] has shown that if we have a proof of ST, the extraction is performed within intuitionistic logic and the program obtained is correct. Many "program-extractors" were designed within already existing systems [236], or within new systems [426]. How to obtain an automatic or computer-assisted proof of ST becomes the main problem. Kreisel wrote several interesting studies concerning this general topic [567], [569], [568], [570].

A good deal of work was and is being done within artificial intelligence. Here there are two major lines of research and development:

- building theorem checkers (and program synthesizers) which are able to accept human guidance [236], [425]
- searching for "special-purpose" automatic theorem-proving methods.

See Section 6.3 for more on the topic of theorem proving.

The pioneering work on deductive program synthesis is that of Manna and Waldinger [649]. They built a formal system within which one can easily and in a natural way justify deductions necessary for proving specification theorems. It is up to the user, however, to decide which deductions must be considered.

Another interesting approach is pursued in the system LOPS [92]. Here the steps to be performed are well-defined, but it is up to the user to perform them in non-trivial cases. This is why, recently, a reimplementation of LOPS has begun based on the above-mentioned "human-guidance" principle.

The approach of Biundo [119] may be characterized as an intelligent search through the deduction tree. This search is based on a systematic use of matchings. There are limitations with respect to subproblem synthesis; moreover, only one existentially quantified variable may occur in the specification theorem.

Smith's approach [881] is based on the application of the *divide and conquer* strategy determined by the recursive construction of objects. Several well-known transformation techniques (e.g. split of post-condition to introduce an invariant, finite differencing) have been implemented in the PROSPECTRA transformation system [585].

Dershowitz [273], Kodratoff [561] and Perdrix [778] have advocated the use of completion techniques for the purpose of deriving executable specifications from both the specification of the problem domain and of the requirements for the solution. From the completeness properties of the completion process it follows that an executable specification can always be found, provided that it can be expressed as a canonical system with the given vocabulary and termination ordering. Unfortunately, in most cases auxiliary symbols are needed. It is not clear how to apply completion techniques to the synthesis of functional programs in rewrite rule form starting from requirement specifications of the kind mentioned above. In any case, a good ground completion procedure must be available in order to achieve better termination of such synthesis processes. See also Section 6.2.

All the systems discussed so far have in common that they ask the user to solve delicate problems. In contrast to this is the system PRECOMAS (PRoofs Educed by COnstructive MAtching for Synthesis) [352], [353], [351] which does not allow the user to interfere concerning the choice of the deductions to be performed. The method for automatic theorem proving of theorems with existential quantifiers upon which PRECOMAS relies is called *constructive matching* (CM). The main feature of the CM method is that it decomposes the problem of proving theorems into (conditional) equation-like sub-problems which are either directly solved or which can be represented as theorems. These new theorems are then in turn solved in the same way.

Among the systems which provide inductive theorem proving, one should not forget the Boyer/Moore theorem prover [133], [134]. It specializes in doing "clever" generalizations, but it does not prove theorems containing existential quantifiers unless the correct Skolem function is provided.

This topic has been studied in the wider context of a *methodology for the development of correct software* [774, 777]. This work provides a framework for the development process, in which each development step is *correctness-preserving* such that the correctness of the resulting moduar software system is correct with respect to the formal requirement specification. The development of software is formalized as a *development graph* representing the modular structure and development steps. This formalization serves as a basis for the design of a *development tool*.

The concepts described above have been studied in a medium sized case study [88]. The study presents the formal development of an efficient lexical scanner from a descriptive requirement specification. The formal development has been performed using SPECTRUM [154] as a specification language and OPAL [343, 282] as a high-level implementation language. It was presented at the COMPASS workshop in Dresden.

In another case study a universal formatting tool (UFO) was developed and implemented in OPAL. Experience from this study shows that errors are as likely to occur in the requirements specification as in the implementation. This case study was presented at the COMPASS workshop in Genova.

In [775] the aspect of developing realizations of data types is investigated. We discuss three approaches how sets may be realized by hash tables.

5.2.3 Parallel Programming

A methodology based on algebraic specification and transformations has also been developed in the context of *parallel programming*. Starting from specifications parallel functional programs are derived by transformation using *skeletons*, i.e. higher-order functions with associated parallel implementations [776, 773]. *Data distribution algebras* have been investigated as a more explicit way to represent data distribution — the main problem with regard to the programming of MIMD architectures [776]. Besides the investigation of the theoretical basis of these algebras using parameterized algebras and sheaf theory, the research has focused on the application of the methodology to numerical algorithms [894].

Furthermore, we have begun to integrate the individual results gained from the case studies to an overall methodology and to implement the methodology using the algebraic programming language OPAL.

5.2.4 Requirements Engineering

Requirements engineering characterizes that phase in the system development process where vague wishes about a desired software system are made precise by explicitly stating

- functional requirements ("what is the system to do") and
- non-functional requirements such as
  - quality attributes of the desired functions,
  - requirements for the implementation of the system,
  - requirements for validation, test, maintenance, and installation,
  - requirements for the development process.

Having primarily readability and understandability even for non-expert users in mind, traditional approaches in this area (e.g. SADT, PSL, RSL) incorporate only a few formal concepts and rely mainly on natural language for their semantics. Consequently, they all fail with respect to important aspects such as consistency, completeness, or integrated methodology. Therefore, approaches based on formal models (e.g. algebraic languages, relational languages, or net-based approaches) to requirements engineering have been suggested (cf. e.g. [505], [766], [762], [857]). Experience gained so far with these new approaches, and in particular with the use of algebraic specifications as a conceptual basis, is encouraging. The major idea in this context is based on the observation that in any case a "formalization step" has to take place, and, rather than using a semi-formal language for expressing the original requirements, the formalism for design specifications, viz. an algebraic formalism, should be used from the beginning. This implies a number of consequences, mainly concerning how to describe non-functional requirements and how to support the formalization step, but also how to "explain" the formal specification in terms of more "human" formalisms, such as natural language or graphics. With respect to the informal explanation of formal specifications, some experiences are already available [895], [287]. The crucial aspect here is the process of formalization: since requirements are informal, the transition to specifications cannot be done formally. So there exist only first attempts at methodological recommendations for how algebraic specification concepts can be used for certain purposes [910]. These methodological approaches are backed by a number of case studies [310], [765], [766], [762].

As far as requirements serve as a description of the data processing problem to be solved, one should be aware that such a problem may be specified by the class of all possible solutions (without necessarily knowing a single solution). Therefore, requirements may be formally defined by constraints (cf. Section 6.3) or by using specification languages with loose semantics (cf. Section 4.7). In both cases the semantics of a specification is given by a restricted class of data types, regarded as the acceptable solutions to the problem.

While algebraic specifications seem to cover functional requirements properly, their usefulness for non-functional requirements is less clear and obvious. In this respect, Guttag's, Horning's and Wing's work [447], [940] is quite promising.

Requirements Engineering is also the discipline for developing a complete consistent unambiguous specification describing what the software product will do. Formal software development methods such as CIP [54], PROSPECTRA [491], ACT [222] or KORSO [175, 777] propose a fully formal description of the functional requirements of the future system. On the other hand current pragmatic requirements engineering methods use a combination of diagrammatic notations (as for instance entity-relationship diagrams, data flow diagrams and finite state machines), but they do not give a precise mathematical semantics integrating these different views. To close the gap between the formal and the pragmatic world recently a number of proposals have been made to give a formal foundation to pragmatic methods and to integrate pragmatic and formal methods in a common semantic framework. In the following some approaches are mentioned, where the method of algebraic specification is chosen as an integrating formal framework: In [513], a formal foundation of the British quasi-standard method SSADM is given using the algebraic specification language SPECTRUM [154]. Approaches to a formal semantics of entity-relationship models based on algebraic specification techniques are presented in [482, 391, 225]. In [716, 715] a formal method for requirements engineering incorporating entity-relationship diagrams and data flow diagrams is proposed.

5.3 Development Graph

A formal framework for the design of modular software systems which incorporates the idea of reusing existing software components in new applications is presented in [365, 474]. Whereas in [474] the underlying specification language has a behavioural semantics, the specification language chosen in [365] is SPECTRUM [154] with its classical model class semantices. The method for the systematic reuse of components described in [365] is supported by the CSDM system: the software development process is represented as a graph and instantiates the notion of components as nodes and the notion of causal relations between these components as edges of the graph. The retrieval of reusable components from a database is based on the signature matching algorithm from [203].

5.4 Reusability

The demand for reusability originates from economic considerations and attempts towards standardization. Rather than always starting from scratch, the use of existing components is cheaper and also less error-prone. A central problem for the identification and the correct use of reusable components is the abstract description of such components. A formal specification is the only form of description that can serve as a basis for a correctness proof; it can be processed automatically and it establishes a degree of confidence in the functionality of the component that is particularly important if a component has to be modified before being reused.

Goguen [404] proposes the algebraic specification language OBJ as well-suited for the design of reusable software systems. A component's interface is specified as an abstract data type and may be parameterized by other components. Combination of components is possible by instantiation using appropriate fitting morphisms. A similar approach is used in Extended ML [836], [848].

In ACT TWO [315] components are modules, which consist of two interface specifications, i.e. an export specification and an import specification, and a body specification which implements the export specification in terms of the import specification. Similarly, in LARCH [445], a component consists of a pair, an "abstract" interface specification and an implementation. Here the implementation is written in a conventional programming language. A similar distinction between a requirement and a design specification is made in PA^{nn}dA-S, the language of the PROSPECTRA project [138], based on the notion of visible and private interface in Ada. Ada bodies are generated semi-automatically by transformation.

In the approach of [655] a component consists of four parts at four different levels of abstraction: a requirement specification, a design specification, a source program and object code. Components are written in a so-called *wide spectrum language* which provides facilities for system design and software development at many levels of abstraction. Matsumoto uses Ada, which has as a consequence that the requirement specification is merely an informal description. CIP-L [54] and COLD [341] are two languages which include algebraic specifications as well as predicative and imperative language styles in one coherent framework and therefore are better suited for such an approach.

In the object-oriented approach of [675] a reusable component is represented by a graph of object-oriented programs, each node of which stands for one level of abstraction in the description of the software. In the approach of the ESPRIT-project DRAGON [945], a reusable component consists of a tree of algebraic specifications, where similarly to [675], each node of the tree is an instance of a certain level of an abstraction of the program with the root being the most "abstract" formal specification and the leaves representing the "concrete" implementations. A methodology for reusing components is described in [947].

A reusability relation has recently been formally defined in [371]. Here, a distinction between *efficient* reusability and *general* reusability was introduced. Several results on reusability and hierarchy of specifications were stated.

A numerical library, *Sophus* (named after the Norwegian Mathematician Sophus Lie), for solving Partial Differential Equations was being developed in [467, 705, 704]. The development technique basically follows that of [842]. An early, simplified implementation of Sophus is described in [638]. A parallel version was also provided, based on the principles for SPMD programming developed in [913]. The third generation of the Sophus library has an expected release date early 1996 and is based on more categorically oriented reasoning [704]. This version is to be used by the Norwegian oil industry for seismic modelling of the North Sea.

5.4.1 Design of Modular Software Systems with Reuse

In [713] a formal algebraic framework for the GenVoca domain modeling approach of Batory and O'Malley [51] is suggested. GenVoca offers a domain-independent interconnection mechanism which allows one to compose the components of a domain-library to yield complex systems. [713] proposes to enhance library-components by algebraic specifications of their interfaces and uses a combination of the specification language SPECTRUM with the notion of algebraic module specification from [329] to give a formal definition of components and their composition.

5.4.2 Support Systems for the Construction and Use of Software Libraries

Current support systems for the construction and use of software-libraries are evaluated in [960, 959]. Based on the results of these evaluations and based on existing concepts for the representation and retrieval of reusable software documents, requirements concerning the development of a reuse manager are formulated. Three kinds of systems have been analyzed and compared: prototypic systems that have been developed at research institutes, in-house systems that have been developed by industrial companies in order to support their own reuse projects and commercially available tools.

Chapter 6

Tools

6.1 Analysis

Tool support for building, maintaining, editing formal specifications relies on requirement analysis, lexical and syntactic analysis, static semantic analysis, structural dependency analysis, but also on textual and graphical visualization. In various specification and development environments, such analysis and vizualization tools are available.

6.1.1 Requirement analysis and tracing

The development of software systems involves a multitude of tasks, deliverables, personnel, and interests contributing each part for the achievement of the whole. The process of developing software systems includes the organization, integration, and management of all these constituent parts. Requirements are the outcome of the initial stage of this process. They state what characteristics and functionalities a system should have, what constraints it should obey, the context where the system is going to be used, etc. Unfortunately, requirements cannot be confined to the initial phase of the development of software systems nor they can be viewed as static expressions. An examination of the situated nature of requirements is given in [396, 400].

In a real project requirements go on throughout the project and are subject to change. Keeping track of requirements evolution, of how they affect the other phases of the system development, and of how they are considered and incorporated in the design, specification, and production of the system is a crucial task. Requirements traceability helps to assure that the system is on the right path. It helps to manage the changes that occur in the system development. It increases the maintenability of the final product and helps the management of the process itself.

TOOR is a tool designed for tracing requirements in system development [781]; its design embodies the idea that the links between requirements, and between requirements and other objects, are a result of the process of system development itself. TOOR considers all artifacts used during the development of a software system, to be objects which can be traced. These objects might include, for example, an interview transcript, a video tape, a design chart, a program specification text, a system manual, etc. TOOR also considers the possible relations between any two objects to be itself an object. TOOR builds and maintains a database of these objects as they are created in the system's lifecycle, and uses this database to support requirements tracing. Features include regular expression search across object and relation identifiers and class identifiers, and hypermedia support for browsing and manipulating the different kinds of objects that may be entered in its database.

6.1.2 Correct Software Development

CSDM (Correct Software Development Munich) [365] is a system supporting the development of correct software through incorporation of formal methods and software reuse. It is designed as a meta CASE-tool, i.e. it is independent of specific software development methods or specific languages. The main concepts of CSDM are hyper-graphs, components, and palette. Hyper-graphs give a uniform formalism for the representation of the software development process as well as the software system under development. The nodes are labeled either with software documents, such as design models, code, or executables, or with graphs, such as the module structure of a software system. The edges relate either software documents or graphs. Components are used to structure the hyper-graphs. They allow for abstraction, encapsulation and modularisation. Components are stored and can be retrieved from the CSDM repository. Every palette gives an instantiation of the CSDM system to a specific appearance and behavior and allows one therefore to instantiate CSDM for a specific software development method. A prototypic instantiation of CSDM has been implemented for the wide spectrum specification language Spectrum and the functional programming language Standard ML, both described in Section 6.4.

6.1.3 Action Semantic Description

The ASD tools include facilities for parsing, syntax-directed (and textual) editing, checking, and interpretation of action semantic descriptions. Such facilities significantly enhance accuracy and productivity when writing large specifications, and are also particularly useful for students learning about the framework. The notation supported by the ASD tools is a direct ASCII representation of the standard notation used for action semantic descriptions in the literature, as defined in [694], which is specified using unified algebras (see Section 2.4 of Chapter 2.1). The ASD Tools are implemented using the ASF+SDF system [559].

6.1.4 Graphical description language

Existing specification systems like the ACT environment [222] usually provide some way to execute algebraic specifications. In this way a form of early prototyping is supported. Term rewriting and narrowing techniques are suitable as the operational processing concept, but they are not suited for recognition by the user of the prototyping tool. Therefore the GVT-System is developed which supports prototyping of algebraic specifications by providing suitable concepts to visualize term structures. These concepts are the foundations for the graphical description language GVT. According to an algebraic specification a graphic specification using the language GVT must be prepared, where for each constructor or sort symbol of the algebraic specification a graphical assignment must be given. During the execution (prototyping) each evaluated term will be converted into its graphical representation corresponding to the given assignments. The most common output format for the graphical respresentation of a term is postscript, which is visualized using the postscript interpreter 'ghostview'. [49] includes the concepts of the language GVT as well as the language description and how to use the system. All of this is illustrated by several examples.

6.2 Evaluation, Prototyping

Specification environments based on algebraic specifications usually provide some way to execute them. Support for prototyping is useful in validating design decisions in early stages. Evaluation by rewriting (using a matching substitution) or narrowing (with a unification substitution) provides the basis for prototyping tools for algebraic specifications. Rewriting techniques can be applied to a wide variety of concepts ranging from terms, graphs, λ-expressions to regular languages and logical formulas.

6.2.1 Rewriting in order-sorted algebras

Following previous work on order-sorted algebra [416, 409] and OBJ3 [551, 377], the problem of computing in order-sorted algebras is addressed in [485] with a new approach putting very few restrictions on the allowed presentations.

Equational, membership and existence formulas can be expressed in the algebraic framework proposed by [660]. A complete deduction calculus is provided to incorporate the interaction between all these formulas. In [485], the notion of decorated terms is proposed to memorize local sort information, dynamically changed by a rewriting process using so-called decoration rules. A completion procedure for equational presentations with ordered sorts computes a set of rewrite rules which allows proving not only equational theorems of the form $(t = t')$, but also membership formulas of the form "the term t is of sort A". This completion procedure, called decorated completion in the following, relies on the hypothesis of sort-inheritance that must be satisfied by the ordered set of sorts. Sort-inheritance can be interpreted as an extension of syntactic regularity, which means that each valid term has a unique least sort [416]. In fact, it expresses semantic regularity [937] in a sort lattice completed by new sorts corresponding to sort intersections. Sort-inheritance is in general undecidable, but a completion process with an additional test is proposed in [486] to check it. As a consequence of working with semantical sorts, the retracts defined in [398, 519] in order to handle syntactically ill-formed terms are superfluous: with decorated terms, the needed sort appears eventually in the decoration of the syntactically ill-formed term during its evaluation, if and only if it is semantically well-formed. The decorated completion is a conservative extension of the procedure in [377]: there, syntactic regularity is assumed and coincides with semantic regularity, thanks to the hypothesis of sort-decreasingness of rewrite rules, which means that for every instance of a rule, the least sort of the right-hand side is smaller than the least sort of the left-hand side.

In [202], an order-sorted rewriting approach is presented that allows for ill-sorted terms obtained from well-sorted ones by application of rewrite rules using syntactically well-sorted substitutions. Together with a condition called sort-convergence, a critical pair lemma can be obtained. The completion procedure in [202] constructs a sort-convergent specification via sort enrichment, i.e. adding new sorts and function symbols when needed. However, the approach may be problematic in case of an infinite number of function symbols added during completion.

The completion of rewrite systems with membership constraints in [228] is also motivated by the failure of the critical pair lemma in the order-sorted term algebra. This approach provides new deduction rules to compute critical pairs in a fragment of second-order logic.

Another approach for the case of flat linear function declarations is described in [937]. Using the concept of semantic sorts, a critical pair lemma is proved, that allows for syntactically ill-sorted terms. This leads to a decision procedure for order-sorted equalities over extended terms (i.e. not necessarily well-formed terms) if the rewriting system is weakly sort-decreasing, which is the extension of sort-decreasingness to multiple rewrite steps. The completion procedure given in [937], transforms weakly sort-decreasing rewrite systems into sort-decreasing ones. Before starting the completion procedure, any specification has to be transformed into a range unique one, where any function symbol can only be declared to belong to exactly one range sort, but may have several different domains. Similarly to [202], the rewriting relation is also defined for terms derived from syntactically typable terms under the rewrite relation using so-called T-substitutions, which test only the coarity of the top symbols of images, instead of the classical order-sorted substitutions of [377, 856, 884].

In [950], a lemma of decidability of order-sorted unification with term declarations (i.e. membership formulas) is given, based on the assumption that all critical overlaps between term declarations are solved, i.e. covered by already existing term declarations. But the decidability is obtained only for regular signatures, using the unification algorithm of [856].

In [935], in order to approximate semantic sorts during completion, instances of equalities generated by the completion procedure are used to propagate sorts. The data structures for the realization of approximated least semantical sorts, the unification algorithm and the corresponding undecidability problems are left open but an implementation is running in the system MERILL [656].

Apart from the transformation of order-sorted equational theories into conditional unsorted theories as in [729], the completion of order-sorted equational specifications by transformation into many-sorted conditional specifications is proposed in [363]. This is a serious alternative for coping with sort-decreasingness during completion. However, the transformed specifications are harder to read. Conversely, in [484], a transformation from ground-confluent and weakly decreasing conditional term rewriting systems (without extra variables) into order-sorted ground-confluent and terminating unconditional term rewriting systems with the same computational power is described.

6.2.2 Term Graph Rewriting

Functional expressions and logical formulas can be represented by graphs and hypergraphs, making it possible to share common subexpressions or subformulas. The evaluation of expressions and formulas can then be realized by graph and hypergraph rewriting. This observation has led to one of the best developed applications of graph transformation techniques to the area of specification and programming.

In [785], rewriting of term graphs—hypergraphs representing functional expressions such that common subexpressions can be shared—is based on the jungle evaluation approach which was developed in [492, 493, 449]. Rewrite steps with term graphs include applications of term rewrite rules as well as "folding steps" which identify common subexpressions. Different aspects of this model of computation are considered: (1) It is shown that term graph rewriting is complete with respect to equational validity in the same sense as term rewriting is. (2) Term graph rewriting and term rewriting are compared with respect to confluence. (3) Termination and convergence of term graph rewriting turn out to be modular properties, in contrast to the situation for term rewriting. The modularity of termination was first shown in [784].

The main results of [786] concern the following topics: soundness and completeness of term graph rewriting with respect to equational validity; the relationship to term rewriting w.r.t. termination, confluence and related properties; sufficient conditions for confluence based on critical pairs; criteria for the preservation of termination and convergence under combinations of systems.

It is well-known that the equational theory of an equational specification is decidable if the specification can be presented as a confluent and terminating term rewriting system. The corresponding fact holds for term graph rewriting with folding (or "collapsing"), and here the decision procedure applies to more systems than in the case of term rewriting. In order to provide a test for confluence, [787] introduces critical pairs in term graph rewriting and establishes a Critical Pair Lemma as an analogue to the well-known result in term rewriting. This leads to a decision procedure for confluence in the presence of termination. As a by-product, the procedure can be used as a confluence test for term rewriting and as such it extends the classical test of Knuth and Bendix because it applies to all terminating and to certain non-terminating term rewriting systems.

In [450, 451], the concept of graph substitution is applied to term graphs. As the main contribution of these papers, term graph narrowing is introduced as an efficient equation solving mechanism on term graphs. A completeness result is established which corresponds to Hullot's classical result for term narrowing [511, 512]: term graph narrowing is complete for all term rewriting systems over which term graph rewriting is convergent. This includes, in particular, all convergent term rewriting systems. Completeness means that if an equation is represented by a term graph, then for every solution of this equation, term graph narrowing can find a more general solution. The general motivation for using term graphs instead of terms is to improve efficiency: sharing common subterms saves space and avoids the repetition of computations.

6.2.3 Evaluation of Distributed Specifications

In [571], a tentative framework to study the evaluation of distributed specifications is introduced. The basic idea is to combine the ordinary evaluation of executable objects at the nodes of a network with sending executable objects to neighbour nodes for further evaluation. The very first results concern completeness and correctness of the distributed evaluation in the case that the underlying network is the import structure of a structured equational specification and the evaluation is done by term rewriting.

6.2.4 Concurrent rewriting

Concurrent rewriting consists in applying concurrently several rewrite rules on a same term. It provides a computation model with a large expressive power and modelizes systems with non-deterministic and irreversible behaviours [554, 918]. The system RECO [5, 6] implements a mathematical semantics based on rewriting logic, using graph rewriting. This implementation needs no explicit synchronisation and offers a high level of abstraction of the target machine. Another interesting feature of the application is its independence from the number of processors. A new parallel garbage collector on graphs has been developed to improve efficiency of the system [6]. The implementation runs on a workstation cluster under PVM (Parallel Virtual machine) and is portable on MIMD machines (IBM SP/1, KSR, ...). Experiments show a good speed-up for the computations of normal forms with respect to a sequential implementation. They also emphasize the importance of inter-processors communications.

6.2.5 Rewriting with nondeterministic operations

The logic for nondeterministic operations suffers from the inherent lack of unrestricted substitutivity. This problem has been solved with the natural deduction systems (cf. Section 2.7 of Chapter 2.1). An alternative is to develop a resolution based logic, and to design associated term rewriting systems. In [580] a ground complete system for rewriting with nondeterministic terms has been introduced. The non-ground case has been presented to the 95 COMPASS/ADT workshop, and is a complete system designed (without using lifting) from the ground case. There the unification procedure is tailored to handle restricted substitutivity. This work is moving gradually toward relational specifications with the syntax and reasoning systems developed on the top of the two earlier works on rewriting.

6.2.6 Rewriting Extended Regular Expressions

In [11], an extended algebra of regular events (languages) is considered, with intersection besides the usual operations. This algebra has the structure of a distributive lattice with monotonic operations; the latter property is crucial for some applications, for instance in the algebraic specification of abstract data types in the framework of unified algebras (see Section 2.4 of Chapter 2.1), where sorts of values are themselves treated algebraically as values. Such specifications are used in action semantic descriptions of programming languages [694]; this extended algebra of regular events has been found to be particularly appropriate in connection with the description of various operations on semantic entities, as well as with that of abstract syntax [698]. A new complete Horn equational axiomatization of the algebra is given, and some term-rewriting techniques for constructing logical inferences of valid equations are developed.

In [10], a notion of partial derivative of a regular expression is introduced and applied to finite automaton constructions. The notion is a generalization of the known notion of word derivative due to Brzozowski: partial derivatives are related to non-deterministic finite automata (NFA's) in the same natural way as derivatives are related to deterministic ones (DFA's). A constructive definition of partial derivatives is given and several facts are proved, in particular: 1) any derivative of a regular expression r can be represented by a finite set of partial derivatives of r; 2) the set of all partial derivatives of r is finite and its cardinality is less than or equal to one plus the number of occurrences of letters from the alphabet appearing in r; 3) any partial derivative of r is either a regular unit, or a subterm of r, or a concatenation of several such subterms. These theoretical results lead to a new algorithm for turning regular expressions into relatively small NFA's and provide certain improvements to Brzozowski's algorithm for constructing DFA's. A prototype implementation of the NFA construction is reported and several examples are presented.[1]

6.2.7 Higher-order rewriting and λ-calculi

Higher-order rewriting is rewriting based on higher-order variables and substitution. Context rewriting [524] shows higher-order rewrite systems in a categorical setting. For combinatory reduction systems (which have second-order variables), the paper [525] gives an implementation, i.e. a compilation scheme together with a correctness proof. A new proof method to show the termination of higher-order rewrite systems, with special emphasis on enriched λ-calculi is designed in [529]. The method extends the semantical method of Zantema [958] in a canonical way to the higher-order case and seems to be more powerful and easier to apply than the method of van de Pol [908]. In [531], the problem is reduced to the question of whether a finite, generalised higher-order rewrite system generates free variables to an open problem in the theory of PCF (Programming Language for Computable Functions, introduced by Scott in 1969), i.e. whether the fully abstract model of sequential PCF over the unit type is recursive. There is some ongoing work that indicates that this problem is decidable.

Presenting first-order terms in curried form is usually regarded as a trivial change of notation. This is less obvious in presence of a powerful type discipline and higher-order terms. In [528] it is proved that currying preserves confluence of arbitrary term rewriting systems. This is based on an earlier proof of the same result in [523], but a more thorough proof was devised to counter [548] where the opposite result is claimed and the construction of a counter-example is given. It has now been established that the original result in [523] was right and the counter-example was wrong.

The λ-calculus provides a convenient framework in which one can formally represent and reason about functional programs. In [373], some work has been done on the use of expansion/reduction systems reasoning about λ-terms, in particular using categorical techniques to formally derive new and innovative rewrite relations for a variety of type constructors. In [515] the basic mathematical properties of such rewrite relations are studied and in [374] the difficult task of extending these to coproducts is achieved. More powerful and expressive type systems containing features such as polymorphism and impredicativity are studied in [375].

6.2.8 Mixed evaluation

Since rewriting is used both for prototyping and theorem proving for algebraic specifications, its efficiency was considered in various respects. The algorithmic complexity of rewriting expressed in terms of the number of rewriting steps to the normal form is studied in [213]; in this work, the class of "s-regular"

[1] Our colleague and friend Valentin Antimirov died on May 31, 1995 [595].

rewriting systems is defined for which the average cost is expressed in terms of simple characteristics of the rewriting system. Efficiency of rewriting may be improved through compiling the rules into code (e.g [538]), or through the use of mixed evaluation combining rewriting for some specification modules and code evaluation for other modules [539, 207, 212].

In the ASSPEGIQUE environment, further described in Section 6.4, the symbolic evaluation tools take conditional rewrite rules into account. There are three different possibilities for evaluation : (i) "standard" rewriting, with textual and graphic trace options, and choice of the rewriting strategy, (ii) evaluation with compiled rules [538], (iii) and mixed evaluation (rewriting + user code) [539, 207, 212]. The mixed evaluation mode allows to perform computations where rewriting interacts with the execution of code. This code should include an implementation for constructor inverses and may be written by the user; it may be an implementation for some modules of the specification, or even for parts of modules of the specification. This facility is implemented in ASSPEGIQUE$^+$ in a very flexible way: the implemented modules may be any module in the specification (not necessarily the "bottom" ones), and for each module for which an implementation is provided, the user may choose to use either rewriting in the specification version or execution of the available code. In [819] conditions that allow the use of mixed rewriting without need for constructor inverses are established, and an evaluation strategy based on the hierarchy of modules is presented.

6.3 Verification

Verification tools include automatic theorem provers, interactive verification systems and program testing. Two important advances have been made in automated theorem proving: the first one is the use of constraints and the second one is the automation of proofs by induction. On the other hand, progress has been made at the conceptual level for reasoning with partial functions, errors and non-determinism, for proving termination, and for structuring proofs in theorem provers. Several experiments have been performed through case studies or specific applications with interactive verification systems.

6.3.1 Constraints in Automatic Theorem Provers

Constraints have been introduced in automated deduction since about 1990, although one could find similar ideas in theory resolution proposed by M. Stickel and in higher-order resolution as done by G. Huet. The idea is to distinguish two levels of deduction and to use adequate and efficient tools at each level. This idea has been widely studied in the context of programming languages (the class of CLP languages) and knowledge representation.

There are several advantages in using constraints in automated deduction [555]. They provide a way to make explicit every symbolic computation step, especially unification, orientation and typing problems to check while applying an inference rule. Moreover they keep track of choices made along a derivation and this is a way to avoid a new choice inconsistent with previous ones. They help to modularize deduction and in particular to design better controls by delaying complex problem solving and pruning some parts of the search tree. Constraints also schematize (infinitely) many objects, especially substitutions or ground instances of formulas, and thus give theorem provers more expressive power.

Continuing previous work by [552], completeness results have been developed for different strategies in automated theorem proving with ordering and equality constrained clauses. By means of equality constrained clauses, the completeness of basic superposition is proved. This is a restricted form of superposition in which only the subterms not created in previous inferences are superposed upon [719]. By adding ordering constraints, the search space is further reduced by inheriting the ordering restrictions of the inference rules [720, 722].

When building in some particularly frequent properties of some function symbols, again constraints help in reducing the search space. For example, in the case of associative-commutative (AC) function symbols, instead of AC-unifying the terms and generating a double exponential number of new clauses, the equality can be kept in an equality constraint (and then interpreted in an AC-theory). Completeness of the corresponding strategies was shown concurrently in [721] and in [916, 917]. In [822] a complete analysis of these constrained deduction methods is presented, also covering the case modulo A [826], and analysing other methods for deduction modulo equational theories E when E-unification is infinitary of even undecidable.

In [718], it is shown that these ideas turn out to be applicable as well to answer computation (the computation of values for existentially quantified variables, as done in logic programming, automated theorem proving/discovering or deductive data bases). By narrowing modulo equational theories like AC, compact representations of solutions are expressed by means of AC-equality constraints. Computing AC-unifiers is only needed at the end if one wants to "uncompress" such a constraint into its (doubly exponentially many) concrete substitutions.

A significant amount of work has been done on symbolic constraint solving, an essential ingredient for deduction with clauses with constraints. The problem of deciding satisfiability of ordering constraints over the Lexicographic Path Ordering (LPO) is solved in [227]; this is extended to the Recursive Path Ordering with Status (RPOS) in [521]. The practical feasibility of these algorithms, and some simpler subcases of the problems are addressed in [721], and in [717] NP-completeness of several cases is shown.

For ordering constraint solving in the case where axioms for associativity and commutativity (AC) are built in into the logic, it is needed to define a new AC-compatible term ordering, due to the complexity of the only previously existing such ordering (based on polynomial interpretations of terms) of [707]. This AC-RPO ordering (called like that because it is a RPO-like precedence based ordering), is defined in [824]. A version of AC-RPO which is uniformly defined for ground terms and for terms with variables is given in [825]. Finally, a satisfiability decision procedure for AC-RPO ordering constraints is proposed in [230].

6.3.2 Proofs by induction

Proofs by induction are important in many computer science and artificial intelligence applications, in particular, in program verification and specification of systems. Two main approaches have been developed for inductive reasoning. The first approach applies explicit induction arguments on the structure of terms [45, 179, 133]. In this framework, the system NQTHM was developed, which was for many years the only significant automated theorem proving system for induction. However, NQTHM often requires interaction with the user. The second approach involves a proof by consistency: this is the inductionless induction method [706, 509, 550, 540, 520, 355, 598, 46, 427, 802].

However, both methods have many limitations on the theorems that can be proved and on the underlying theory. Indeed, guiding a proof by explicit induction requires some skill in finding the right axioms or hypotheses to apply. On the other hand, the proof by consistency technique does not require guidance from the user since the generation of lemmas is performed automatically through the completion procedure. However, the completion often misses good lemmas and fails where explicit induction succeeds.

Test-set induction

Test-set induction has been proposed as an alternative proof system for automating inductive reasoning in theories defined by conditional axioms [130, 125, 129, 131]. Test-set induction combines advantages of explicit induction and inductive completion. For instance, it shares with inductive completion systems the nice property of refutational completeness in the following sense: any conjecture which is not valid in the initial model is guaranteed to be disproved. This detection of false conjectures reduces wasted time and effort in trying to prove them. Test-set induction relies on two ingredients: a notion of test-set, which can be viewed as a finite description of the initial model, and a new simplification strategy. The key idea of the simplification strategy is to use axioms, previously proved conjectures, and instances of the conjecture itself as soon as they are smaller than the currently examined proposition with respect to a well-founded relation. This last point captures the notion of induction hypothesis in the explicit induction paradigm. The refutational aspect of the procedure requires the ground confluence property of the axiomatization and, also, suitable test-sets. The test-set is used to synthesize an induction rule which is well-adapted to the conjecture and to the axioms. A test-set can be viewed as a partition of the set of ground normal forms. Therefore test-sets allow a natural case analysis when proving an inductive property. The SPIKE system [126, 125, 132] was initially developed for performing test-set induction. The principal functionalities of SPIKE are proof by induction and help in the construction of correct specifications. In contrast to the majority of current proof systems that construct their proofs step by step and require frequent user interaction, not to say a great expertise on the part of the user, SPIKE is meant to reduce the number of interactions due to the automation of numerous routine tasks. For instance, it has proved the Gilbreath Card Trick using two user lemmas [131] while provers like

COQ [284, 508], Nqthm, and RRL using classical format for induction required approximately fifteen to twenty lemmas. SPIKE also provides facilities for checking the consistency of function definitions, and detecting and correcting incomplete function definitions. Both tasks involve the theorem-proving engine in a non-trivial way.

A new procedure for proof by induction in parameterized conditional specifications in proposed in [126, 127, 128]. Unlike a previous procedure [131], this new one, when limited to non-parameterized conditional specifications, can refute general clauses; refutational completeness is also preserved for boolean ground convergent rewrite systems even if the functions are not sufficiently complete and the constructors are not free. The property of sufficient completeness is very important in specification systems but is in general undecidable. A procedure for testing this property for parameterized conditional specifications has been proposed and is implemented in SPIKE. Experiments illustrate the improvements in length and structure of proofs, due to parameterization. SPIKE has been extended by a powerful divergence critic [932] which proved to be very successful at identifying divergence and proposing appropriate lemmas and generalisations for a large number of theorems; it enables the system to prove many theorems from the definitions alone where other systems like Nqthm or Clam fail [932].

Proofs by consistency

Final algebra semantics for constructive equational systems provide a class of models with a final algebra. A method for proof by consistency with respect to the final model is described in [632], that contains the proof methods of Musser, Goguen, and Huet and Hullot as special cases. In [634], the proof method for the initial algebra due to Bachmair is extended to the observational congruence induced by a branch of final algebra semantics, and the refutational completeness results for initial algebra apply to the final case under a mild restriction. The notions of complete sets of positions and initial relatedness allow restricting the generation of critical pairs as much in the final algebra as in the initial case.

A firm connection between the processes of narrowing and proof by consistency is established in [635, 636]. Narrowing and proof by consistency may be used to simulate each other, which allows for the migration of results between the two process descriptions. Decidability results for validity of equations in the initial algebra are obtained from existing results on narrowing. Furthermore several results on completeness of position selection strategies for narrowing appear to be special cases of a generalization of a result on covering sets presented by Bachmair. For instance, only basic superpositions need to be considered for proof by consistency, and thereby sufficient criteria for decidability of validity of equations in the initial algebra are obtained by lifting a termination result for basic narrowing. The generic nature of this proof is used to show that terminating and complete narrowing strategies may only exist for theories where validity of equations in the initial algebra is decidable, and correspondingly that terminating complete narrowing strategies gives a decidability result for the initial algebra.

The first-order method for proof by consistency is lifted to higher-order rewrite systems, more precisely for a rewriting relation due to Nipkow, which is based on systems with patterns as left-hand sides of the rewrite rules [615]. This approach suggests further corresponding liftings for narrowing and equational disunification.

Ground reducibility

Proofs by consistency and sufficient completeness are closely related to the ground reducibility problem: a term is ground reducible if all its ground instances are reducible. This problem has been proved decidable for ordinary term rewriting systems [782, 542]. There are two main approaches for testing ground reducibility: the first one constructs a finite automaton such that the term is ground reducible if and only if the language accepted by the automaton is empty [121, 229]. The second one is based on test sets: the term is ground reducible if and only if all its instances by elements of the test set are reducible [565, 566, 543, 490, 855]. However, in presence of associative functions the problem is more complex. It was shown undecidable for term rewriting systems over a signature containing among other functions an associative function [541]. This result cannot be generalized in a direct way to word rewriting system with variables, i.e. rewriting systems over a signature consisting of one binary associative function (concatenation) and finitely many constant symbols(letters), as the auxiliary non-associative functions are essential in the proof. In [597], undecidability of ground reducibility for word rewriting systems with variables is proved. If the rewriting system is restricted to be left-linear, the ground reducibility problem is decidable for arbitrary (non-linear) word w. In [596] the complexity of the problem is addressed: it is co-NP-complete when w is restricted to be linear too. This case can be viewed as the inclusion problem for regular languages represented by regular expressions of a particular class. This problem is known to be $PSPACE$-complete for general regular expressions.

Inductive theorem prover

InterACT (Interactive Theorem and Completeness Prover for ACT ONE Specifications) described in [558], is a proof support tool for algebraic specifications of abstract data types described in the language ACT ONE. It is possible to prove universally and existencially quantified conditional equations of a specification's theory and to prove the completeness of a specification relative to a set of constructors. The main focus is to use InterACT for educational purposes, e.g. in courses about algebraic specifications, term rewriting, etc.

Terminating generator induction

An applicative language level called TGI, which stands for *terminating generator induction*, has been developed for the specification language Abel. TGI specifications give rise to convergent rewrite rules, which enable efficient manipulation of formulas and other expressions for purposes of simplification and proof [631]. At the TGI level, all proof obligations in connection with the composition of modules and module simulation are quantifier-free formulas in constructively defined functions, provided that user specified axioms are quantifier-free.

In this context, definedness and strictness in generator inductive definitions is studied in [631, 745]. A generalized form of inductive definitions as first suggested by Guttag is considered. Partial functions may be defined by introducing error values in the right-hand sides. For such definitions the intuitive semantics is not consistent with that of unrestricted rewriting. In general, the addition of rewrite rules capturing the intuitively motivated error propagation would destroy confluence of the rewrite system. However it is possible to mechanically modify the rewrite system in order to obtain a confluent system consistent with the intuitive semantics.

In proofs by structural induction, equational subproofs emerge and play an important role in the proof process, but existing theorem provers tend to use ad hoc methods rather than existing semi-decision methods for such equational proofs. These ad-hoc methods are formalized in [633] and illustrated in fully automatized proofs by structural induction.

6.3.3 Behavioural Proofs

Behavioural proof techniques are needed for proving the correctness of abstract implementations in an observational framework. The ISAR system [53] provides an environment for correctness proofs of algebraic implementations. The underlying correctness notion is based on observational semantics and the proof procedure of the system is based on the principle of context induction (which is a particular instance of structural induction). The input of the ISAR system is an implementation step consisting of an abstract specification to be implemented, a concrete specification used as a basis for the implementation and an implementation construction. If all steps of the (interactive) proof procedure are successfully performed, the system has proved the correctness of the implementation step. The underlying algorithm of the ISAR system developed in [473] uses the context induction principle.

6.3.4 Termination

Proving termination of rewrite systems is an important issue. There are essentially two types of well-founded reduction orderings on terms, surveyed in [274]. The first one, called syntactical orderings, provides the ordering via an analysis of the term structure and are based on a concept of precedence which is a partial order on the signature. Another family consists of semantical orderings, that interpret the terms in another structure where a well-founded ordering is known. Both methods can be combined and a general path ordering dealing with semantical as well as syntactical aspects has more recently been proposed in [275].

The interpretation method generalizes the idea that for proving that a process terminates, one can associate a number with each state and prove that each step decreases this number. When working with open terms, i.e. terms with variables, one can associate with each term a function. Usually people use polynomial. A new method for proving termination based on elementary interpretations, i.e. function built with polynomials and exponentials. This method has been implemented in CAML and is available in ORME [608, 610].

In [823], the framework of extension orderings has been developed as a technique for combining termination orderings for term rewrite systems (TRSs), where the combined (or extended) ordering preserves properties like totality on ground terms, or the ones of being a reduction ordering, or a simplification ordering. This also leads to simple and constructive proofs for modularity results of the termination property for rewrite systems. Work has importantly progressed on termination aspects related to data type systems, a framework whose basic idea is to incorporate algebraic calculi into a pure type system. A first result for this purpose is the definition of RPO-like reduction orderings as a standard tool for proving termination of typed λ-calculus in combination with first-order (and, in a second stage, higher-order) rewrite rules. A main tool used for the first-order case are the aforementioned extension orderings.

Termination of order-sorted rewriting is studied in [731]. The dominating method for proving termination of order-sorted systems has been to simply ignore sort information, and use the methods developed for unsorted rewriting. The problem with this approach is that many order-sorted rewrite systems terminate because of the structure of the set of sorts. In these cases the corresponding unsorted system would not terminate. The problem of order-sorted termination is approached by mapping the order-sorted rewrite system into an unsorted one such that termination of the latter implies termination of the former. By encoding sort information into the unsorted mapping, general purpose termination orderings may be used to prove termination of order-sorted rewrite systems whose termination depend on the sort hierarchy. A sequence of gradually stronger methods is presented and compared with other published methods.

The termination ordering for higher-order rewrite systems presented in [614] is an extension of the recursive path ordering. Keeping close to the general path ordering of Dershowitz and Hoot, sufficient properties of the termination functions are proved for this method to apply. Thereby a class of different orderings is described.

In [593], termination of curryfied term rewriting systems where functional values are introduced by "partial application", is studied. The limitations of syntactic simplification orderings for such systems are discussed. A proof method is proposed, based on three techniques: 1) Requirements on stability and monotonicity are relaxed. 2) Variables and inner function symbols of (potentially) functional types are allowed to contribute in the underlying precedence. 3) A standard polymorphic type system is refined so as to express non-functional polymorphism. Founded on this method, the curryfied path ordering with status (CPOS) is introduced. CPOS coincides with the recursive path ordering with status on first-order terms, and is comparable in strength to other approaches for higher-order rewrite systems, but also allows for polymorphism. Under mild restrictions on the underlying precedence, the so-called curry rules are oriented correctly under CPOS.

6.3.5 Reasoning with partial functions, errors and non-determinism

Partial functions play an important role in computer science. In order to reason about partial functions, one may extend classical logic to a logic supporting partial functions, a so-called partial logic [741]. Usually such an extension necessitates side-conditions on classical proof rules in order to ensure consistency, and introduces non-classical proof rules in order to maintain completeness. These complications depend on the choice of consequence relation and non-monotonic operators. In computer science applications such complications are undesirable, because they affect (semi-) mechanical reasoning methods, and make manual reasoning difficult for computer scientists who are not logicians. By carefully choosing the consequence relation and non-monotonic operators, a simple calculus for partial functions arises. The resulting logic is "healthy" in the sense that "meaningless" formulas (such as $x/0 > 0$) or $top(emptystack) > 1$) cannot be concluded, except from contradictory or false assumptions, and a meaningless assumption provides no information. This requires all axioms to be healthy; and as a consequence the "excluded middle" $(a \vee \not a)$ must be weakened (to meaningful a's). All the well-known classical rules preserve healthiness and are therefore sound in the logic, provided substitutions are restricted to meaningful terms. This means that program reasoning methods based on classical logic usually can be adapted to the presented partial logic.

Classical logic is based on two truth values. However, in computer science applications, it is often desirable, or even necessary, to consider a third truth value, representing non-terminating computations, or computational errors. It is therefore of interest to investigate whether reasoning methods for standard propositional logic can be generalized to the case of three truth values ("three-valued logic"). In [744] it is shown how rewrite systems for standard propositional logic can be adapted to the case of three-valued logic, considering various extensions of the standard logical connectives. It is argued that (inductive) completeness is more essential for practical application in the case of three truth values than in the standard case. For three-valued propositional logic with strict operators, we present a convergent and (inductively) complete system, modulo associativity and commutativity axioms, with a normal form similar to that of Hsiang's system. Unfortunately, convergent and complete rewriting in three-valued logic with Kleene operators requires more complex normal forms. By adding an operator we give a convergent and (inductively) complete system using Hsiang's normal form of subterms.

In [743], a logic for reasoning about requirement specification of processes with (internal and external) non-determinism is developed. Conceptual simplicity is achieved by avoiding the use of infinite sequences, and by formulating a proof system within first-order logic, with a non-standard interpretation. In particular, non-deterministic processes are specified by means of a simple event relation, called the ready relation, whereas the underlying semantics is a set of models, each with a set of such relations. By weakening the usual requirement of (sequence prefix) monotonicity, an expressive specification language is proposed, enabling us to relate incompatible executions in the same specification, without the risk of making meaningless or inconsistent specifications. It is then possible to specify internal non-determinism without underspecification. A sound and (relative) complete basic proof system is presented. The classical rules and axioms of first-order logic with equality are sound. In addition, some rules and axioms are needed for the ready relation and other process operators considered. Refinement can be done in two ways: by enriching a specification over a set of processes, one may refine several processes (reducing the set of possible models). By the explicit refinement operator, one may refine a single process (reducing the number of executions in each model).

6.3.6 Structuration and validation

For proving confluence or termination of rewrite systems especially when there are many rewrite rules, modularity results are useful in order to state that a rewrite system is confluent or terminating if it can be partitioned into smaller systems which have the same property. The research on modularity for disjoint rewrite systems originated with Toyama who proved that the disjoint union of two confluent rewrite systems is confluent [902] and that the disjoint union of two terminating rewrite systems is in general not terminating [901]. Sufficient syntactic restrictions on the rules, in terms of collapsing and duplicating rules, can be required to keep the property of termination [827]. These first results have been extended by Middeldorp [676], who also considered the case of conditional rewrite systems. The disjointness assumption was relaxed in the case of constructor systems that are allowed to share constructors, while preserving the confluence and termination properties [678]. A survey of properties of rewrite systems preserved under (disjoint) unions can be found in [677].

In [629], term rewriting in structured algebraic specifications is addressed, in particular the question whether properties such as confluence and termination are preserved by structuring operations. By generalizing the categorical treatment of universal algebra, a compositional semantics for term rewriting systems is given centering on the concept of a monad. This allows to describe structuring operations in terms of colimits of monads. In this framework, the question of whether such properties as mentioned above are preserved by arbitrary structuring operations can be broken down into the question of whether they are preserved by coproducts and coequalizers of monads respectively, since all other colimits can be built from these. The work has concentrated on the preservation of confluence, because it is more amenable to a categorical treatment than termination. The preservation of confluence under the coproduct of two monads has been proven (this is Toyama's theorem in a categorical guise); the preservation of confluence under coequalizers is currently under investigation.

In [461], a rudimentary language of structured theory presentations is presented, together with an exploration of the use of this structure in proof search in an arbitrary object logic. The behaviour of structured theory presentations under representation in a logical framework such as LF [459] is studied, focussing on the problem of "lifting" presentations from the object logic to the metalogic of the framework. The topic of imposing structure on logic presentations, so that logical systems may themselves be defined in a modular fashion, is also briefly considered.

6.3.7 Program testing

The growing interest in the use of formal methods has raised several works on testing based on formal specifications. It turns out that formal specifications are quite fundamental to a rigorous and systematic approach of program testing. A testing theory from formal specifications has been proposed in Orsay [73, 81, 369]. It as been applied to algebraic specifications [81], VDM specifications [280], exception-specifications [360] and LUSTRE specifications/programs [507]. Its application to an enrichment of algebraic specifications with data type boundaries descriptions is currently investigated. Moreover, this testing theory has numerous similarities with the works that have been done in the area of protocol testing from formal models such as LOTOS specifications, finite transitions systems [142, 625, 779] or CSP descriptions [768].

This testing theory is based on the notion of observable exhaustive test data set, which is defined from the semantics of a formal specification. The success of such a test data set ensures the coverage of each required property through program executions. A finite subset of this test data set is selected by assuming hypotheses (selection hypotheses) on the program under test. Such a test data selection can be assisted by tools, as it as been demonstrated in the case of positive conditional algebraic specifications by the LOFT tool [650, 651]. LOFT (LOgic for Functions and Testing) makes it possible to implement selection strategies corresponding to general selection hypotheses. It is based on a conditional equational resolution procedure (conditional narrowing) enriched with some control mechanisms similar to those proposed by logic programming languages with coroutining (Prolog II, Mu-Prolog, ECLIPSE ...).

The testing method as well as the LOFT tool have been experimented on several real sized case studies. Test data selection experiments have been done on two modules from the software of the Lyon unmanned subway (in collaboration with the French certification agency of railway systems, INRETS). The results of these experiments [264, 263] are encouraging in two aspects: first, they have shown that general selection hypotheses and their corresponding strategies, which have been elaborated on academic examples (sorting problems, binary search trees ...), can be used as such on industrial examples; second, the sub-domains covered by these selection strategies naturally focus on critical cases and conjunctions of such cases. This case study is reported in [652].

Other experiments of test data selection have been done in collaboration with the LAAS laboratory (Toulouse) within the PDCS basic research action (Predictable and Dependable Computing Systems). The objective was to compare the method of the LRI and the statistical testing methods (functional or structural) proposed by the LAAS, with other classical methods. This comparison uses mutation analysis (simple fault injections) on software modules of a nuclear plant control system. The experimental results [653] clearly showed the superiority of the LRI and LAAS methods w.r.t. to other classical methods. Furthermore, these results pointed out the complementarity between these two advanced testing methods.

6.3.8 Validation case studies

Several experiments on validation have been performed, either concerning applications like hardware verification, or using various interactive proof systems such as COQ or LP.

Hardware and protocol validation

An important application of formal specification is proof and design of correct hardware. A main approach is based on Mealy finite state machines. In this framework, both the specification and the implementation of circuits are described by finite state machines or automata and the goal is to prove that the two machines are behaviorally equivalent. This problem is exponential, thus it is hopeless to build an algorithm that would work in any case. So people build algorithms that work on large ranges of examples different from one algorithm to the other. Several new algorithms for proving the equivalence of two abstract machines have been designed and implemented in FANCY, a software that includes also other algorithms in order to compare their efficiency [592].

In [204], the prover LP is used to prove the correctness of a non restoring division circuit. LP has also been experimented for proving parallel programs written in UNITY [201]. This requires coding the theory of UNITY programs into that of LP. This approach was applied to prove correctness (safety and liveness) of a communication protocol on a faulty channel [205].

Interactive Verification with COQ

COQ [284] is a proof assistant for the calculus of construction. The main idea of COQ is to use a constructive logic based on a polymorphic typed λ-calculus and type dependencies to built algorithms. One of the characteristics of this approach is that one proves the correctness of an algorithm while describing its implementation in this strongly coercive approach. In other words, only correct algorithm can be described this way. A unification algorithm has been developed in [821], as well as algorithms dealing with graphs (minimal spanning trees [820]) or terms (lexicographical path ordering [603]).

Data base management

In [97], the design of a data base for an algebraic specification environment, such as ASSPEGIQUE, and an algebraic specification for this data base management are provided. One of the issues in this database design is to distinguish between "static" aspects, i.e. the attribute updating that is necessary to preserve the database consistency when a single action takes place, and the "dynamic" aspects, that take into account the management of concurrent accesses to the database.

Transit-node case study

The VTT research action (Verification of data Types and Temporal properties) was an opportunity to study the complementary aspects of proving and testing. The aim of this action was to study the complementarities of different formalisms and their tools such as temporal logics, finite labeled transition systems, LOTOS specifications, and algebraic specifications in the Verification process (does the specifications fulfill their requirements?). The tools used in this experiment were MEC, XESAR, LP and LOFT. A representative common case study was chosen, the Transit Node, and specified in various formalisms (e.g. [103]).

The use of these tools during the validation step has pointed out some ambiguities and errors in the original version of the specification [12].

In [210] and [101], the specification of this case study was studied regarding its adequacy to the informal specification; expected behaviours and properties are expressed in terms of the formal specification. The validation process first uses prototyping to "test" the specification, and then theorem proving. Issues regarding the checking of such specification properties as consistency, sufficient completeness are presented in [208].

Verifying a Distributed List System

In [594], a caching protocol for shared memory parallel computers is defined and used to illustrate the application of rewriting techniques in program verification. The paper concentrates on a linked list system, which is a central aspect of the caching protocol. First an informal proof of this system, including a rather large invariant, is given. Thereafter it is shown how the list system and the invariant can both be described in the formalism of rewriting logic, and this is used to carry a significant part of the verification through mechanically, using the OBJ3 interpreter.

Verification of a package router

In [256], a package router is considered. Two specifications are given: a discrete event simulation model in traditional style useful for studying the dynamic behaviour of the system, and an applicative, more abstract specification useful for proving properties of the system. It is shown how the two specifications can be proved equivalent in a certain sense, using techniques of program verification, more precisely a suitably extended Hoare logic. Additional proof rules are introduced to cater for special scheduling statements and the concept of system time. An important part of the verification is to identify a correct and sufficiently strong system invariant.

6.4 Specification and Development Environments

6.4.1 Asspegique

Asspegique$^+$ is an integrated environment for the development of large algebraic specifications and the management of a specification data base. The Asspegique$^+$ specification environment supports the specification langage Pluss [104, 93, 368]. The previous version of this environment, Asspegique [100, 98] supported only the enrichment primitive, while Asspegique$^+$ [209, 819] also supports the renaming, parameterization and instanciation primitives. The tools available in the Asspegique$^+$ specification environment include browsing tools, a specification editing tool, a specification modules integrator, symbolic evaluation tools and interfaces to other tools or environments. Most of these tools use Cigale [921], a system for incremental grammar construction and parsing, which has been especially designed in order to cope with a flexible, user-oriented way of defining operators, with coercion and overloading of operators.

While the aim of the Asspegique$^+$ environment is to provide the user with a wide range of tools, this is achieved both through specific Asspegique$^+$ tools and through interfaces to tools built by other colleagues. Interfaces between Asspegique and Reve [607] and Slog [354] were first built, then a generic way to provide inter-operability between tools was designed [102, 211, 832]. The principles developed for designing a generic interfacing design between algebraic specification tools or environments comprise both issues concerning a common interchange format, and issues regarding interaction with the user, that is how to get and display information using the specification language known to the user. These ideas were used for developing interfaces with the proof checker LP [364], and, within the Salsa project [832], interfaces with environments for the algebraic specification languages Lpg [85] and Glider.

6.4.2 Extended ML

The long-term goal of work on Extended ML is to provide a practical framework for formal development of Standard ML programs together with an integrated suite of computer-based specification and development support tools and complete mathematical foundations to substantiate claims of correctness. The complete formal definition of the Extended ML language [533] constitutes an important milestone in this programme, necessary to provide a basis for further research on foundations and tools. One of the key issues that had to be resolved was the design of a logic suitable for describing properties of Standard ML programs, taking into account features like recursive type and function definitions, exceptions, polymorphism, and non-termination. A number of errors in the Standard ML semantics [679] have been discovered in the course of this work [526]. Work has begun on rules for proving theorems about EML specifications, to be used in verifying correctness conditions that arise in the course of formal development. Research has so far concentrated on equality reasoning [530] and quantifier rules.

The length and requisite formality of the definition of Extended ML renders it rather difficult to penetrate. Accordingly, [532, 534] provides an informal overview of the definition, explaining most of the main issues involved and justifying some of the choices taken.

6.4.3 OBJ

Order-sorted term rewriting provides the operational semantics for the language OBJ [422] (cf. Section 4.2.2 of Chapter 4), so OBJ's functional sublanguage is intended to provide support for prototyping. Its sophisticated module language also provides some support for prototyping at the level of (horizontal) system structure. It has been successfully used for research and teaching [402] in software design and specification, rapid prototyping, theorem proving, and hardware verification.

6.4.4 ABEL

Abel [259, 260, 257] is a language together with a formal logic for use in program development. The overall goal has been to support specification and program development through semi-mechanical aids for reasoning and verification. Major concerns for the language and the associated reasoning formalism are that consistency requirements and other proof obligations are as simple and manageable as possible. In particular it was sought to offer language constructs well suited for mechanical aids to reasoning, with some emphasis on mechanisms for constructive specification; to encourage modularization and abstract specification of interfaces; to facilitate reusability of modules through the use of parameters; to offer powerful ways of putting modules together, including inheritance, restriction, inclusion, extension, as well as assumption specification and checking; and to enable simple and manageable proof obligations, including those related to the composition and (internal) consistency of modules, in the form of first order formulas.

A major idea behind ABEL has been to allow the same language cover all stages of program development from abstract requirement specifications to efficient low level programming. Thus ABEL includes facilities for non-constructive requirements specification, constructive specification, applicative programming, and object oriented, imperative programming. (The term "specification" covers all of these.) The language is built around a few concepts which may be applied at all levels. Different stages may be related to each other through a concept of module simulation. If a low level module is proved to simulate an abstract one, then the latter may be used as an abstract specification of the low level module.

The ABEL language has been developed in close interaction with a regular student course on program specification and verification. The most important sources of ideas have been SIMULA 67 [258], LARCH [445, 444], and OBJ [359].

6.4.5 Elan, a logical framework based on rewriting logic

Rewriting logic formalized in [669] appears both as a semantic framework for specification of systems and languages, well-suited to concurrency, parallel programming, object-oriented specification, and as a logical framework for representing logics. In this direction, logic programming in a broad sense, theorem proving and constraint solving can be described in a uniform way by giving the syntax of formulas, a set of axioms, a set of deduction rules, a proof calculus making precise how proofs are built, altogether with one or a class of models. These three computational processes can be formulated within rewriting logic that provides the uniform framework to reason about combination and integration of these three paradigms. ELAN is a logical framework that allows the formalisation of various logics and constraint solvers. The rewriting logic [669] is the formal basis of the evaluation mechanism in ELAN. Rules may be conditional and are enriched by local variables. A notion of strategy is added and a computational system is a rewrite theory enriched by a set of strategies controlling application of rules and restricting the search space of computation. A strategy language is provided with concatenation, iteration and choice operators [553]. The system is used to prototype, experiment and study the combination of different computational systems that provide basis for constraint solving, theorem proving and logic programming paradigms. Different deduction processes using constraints have been designed in ELAN, such as a narrowing process with constraints to solve queries in theories defined by rewrite rules, the kernel of a constraint logic programming language using for instance a constraint solver for systems of equations in commutative theories. In [556], a completion process using a basic superposition strategy expressed with constraints, is described in ELAN. The strategy language is used for experimenting various simlification strategies in this context. Another significant example developed in ELAN is a higher-order unification algorithm based on λ-calculus with explicit substitutions [285, 123]. A compiler of ELAN in C considerably improves the efficiency of the interpreted system.

6.4.6 The Opal Compilation System

OPAL is an algebraic high-level language described in [343, 282]. An important aspect of the work performed on the development of the algebraic programming language OPAL is to provide tools for development of adequate software in an algebraic framework. Therefore the OPAL COMPILATION SYSTEM $\mathcal{OCS}$ has been developed which supports the development of modular software systems. The kernel of $\mathcal{OCS}$ is an optimizing compiler that generates efficient code for OPAL-programs [463]. The current version of the OPAL COMPILATION SYSTEM is $\mathcal{OCS}$ 2.1e which has been released in July 1995.

Apart from the kernel, the OPAL compilation system contains a restructured and augmented standard library, a revised scheme for incorporating C code, and a introductory tutorial [338]. A documentation system for OPAL was developed, which supports a literate programming style which can also deal with code which is not or only partially documented.

6.4.7 Spectrum

The language SPECTRUM whose concepts are described in [155, 156] is based on axiomatic specification techniques and is oriented towards functional programs. SPECTRUM includes partial functions, definedness logic and fixed point theory, higher-order elements and typed λ-abstraction, non-strict functions and infinite objects, full first-order predicate logic with induction principles, predicative polymorphism with type classes, parameterization and modularization. SPECTRUM is based on the concept of loose semantics. The specification language SPECTRUM is supported by a parser, abstractly described in [434]. An extension of the semantics of SPECTRUM wich supports higher order non-continous functions was encoded in the higher-order logic theory HOL of the interactive theorem prover Isabelle [803]. An implementation of this extended semantics in Isabelle is given in [803].

6.4.8 Focus

FOCUS is a framework for the systematic formal specification and development of distributed interactive systems and their components. An informal introduction to FOCUS can be found in [152], case-studies in [151, 157], and a formal language definition is given in [853]. An implementation of FOCUS in the interactive theorem prover Isabelle is given in [853].

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] J. Abrial. *B-Tool Reference Manual*. Edinburgh Portable Compiler, 1991.

[3] M. Aiguier and G. Bernot. Algebraic Semantics of Object Type Specifications. In R. Wieringa and R. Feenstra, editors, *Selected Papers from ISCORE '94 (International Workshop on Information Systems - Correctness and Reusability)*, Singapore, 1994. World Scientific. ⊕.

[4] W. Aitken, R. Constable, and J. Underwood. Metalogical Frameworks II: Using Reflected Decision Procedures. *Journal of Automated Reasoning*, 1996. to appear, ⊕.

[5] I. Alouini. Réécriture concurrente: Etude et implantation. Rapport de DEA, Université Henri Poincaré – Nancy 1, Sept. 1993. ⊕.

[6] I. Alouini. Concurrent Garbage Collection for Concurrent Rewriting. In J. Hsiang, editor, *Rewriting Techniques and Applications. 6th International Conference, RTA-95, Kaiserslautern (Germany)*, pages 132–146. Springer LNCS 914, 1995. ⊕.

[7] D. Ancona, E. Astesiano, and E. Zucca. Towards a classification of inheritance relations. In *Proc. ISCORE '93 (International Workshop on Information Systems - Correctness and Reusability)*, number 01 in Informatik-Berichte, Hannover, 1993. Institut für Informatik. ⊕.

[8] D. Ancona and E. Zucca. A Formal Framework for Modules with State. In *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology*. Springer LNCS, 1996. To appear, ⊕.

[9] T. Andersson. Safir, dataavhengighetsanalyse og programmering for parallelle arkitekturer. Master's thesis, Universitetet i Bergen, 1994. ⊕.

[10] V. M. Antimirov. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theoretical Computer Science*, 155:291–319, 1996. ⊕.

[11] V. M. Antimirov and P. D. Mosses. Rewriting Extended Regular Expressions. *Theoretical Computer Science*, 143:51–72, 1995. ⊕.

[12] A. Arnold, M.-C. Gaudel, and B. Marre. An Experiment on the Validation of a Specification by Heterogeneous Formal Means: the Transit Node. In *Fifth IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, 1995. ⊕.

[13] D. Aspinall. Subtyping with Singleton Types. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic. 8th Workshop, CSL'94. Kazimierz, Poland, September 1994*, pages 1–15. Springer LNCS 933, 1995. ⊕.

[14] D. Aspinall. Types, Subtypes, and ASL+. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 116–131. Springer LNCS 906, 1995. ⊕.

[15] D. Aspinall and A. Compagnoni. Subtyping Dependent Types. In *Proc. 11th IEEE Symp. on Logic in Computer Science*, 1996. To appear, ⊕.

[16] D. R. Aspinall. *Algebraic Specification in a Type-Theoretic Setting.* PhD thesis, University of Edinburgh, 1996. Forthcoming, ⊕.

[17] E. Astesiano and M. Cerioli. On the existence of initial models for partial (higher-order) conditional specifications. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 74–88. Springer LNCS 351, 1989.

[18] E. Astesiano and M. Cerioli. Non–strict don't care Algebras and Specifications. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91. Proceedings of the International Joint Conference on Theory and Practice of Software Development. Volume 1: Colloquium on Trees in Algebra and Programming*, pages 121–142. Springer LNCS 493, 1991. ⊕.

[19] E. Astesiano and M. Cerioli. Partial Higher–Order Specification. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, pages 74–83. Springer LNCS 520, 1991. ⊕.

[20] E. Astesiano and M. Cerioli. Partial Higher–Order Specifications. *Fundamenta Informaticae*, 16:101–126, 1992.

[21] E. Astesiano and M. Cerioli. Relationships between logical frameworks. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 126–143. Springer LNCS 655, 1993. ⊕.

[22] E. Astesiano and M. Cerioli. Multiparadigm Specification Languages: a First Attempt at Foundations. In D. Andrews, J. Groote, and C. Middelburg, editors, *Semantics of Specification Languages (SoSL'93)*, pages 168–185. Springer-Verlag Workshops in Computing, 1994. ⊕.

[23] E. Astesiano and M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoretical Computer Science*, 152(1):91–138, Dec. 1995. ⊕.

[24] E. Astesiano and M. Cerioli. Non-strict Don't Care Algebras and Specifications. *Mathematical Structures in Computer Science*, 1996. To appear. ⊕.

[25] E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, and E. Zucca. The Ada Challenge for new formal semantic techniques. In *Proceedings of the Ada-Europe 1986 Internationall Conference on Ada: Managing the Transition*, pages 239–248. Cambridge University Press, 1986.

[26] E. Astesiano, A. Giovini, and G. Reggio. Observational structures and their logics. *Theoretical Computer Science*, 96(1), 1992. ⊕.

[27] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specifications. Surveys and state of the art reports.* Chapman and Hall, 1996. to appear.

[28] E. Astesiano, F. Morando, and G. Reggio. The SMoLCS Toolset. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 810–801. Springer LNCS 915, 1995. ⊕.

[29] E. Astesiano and G. Reggio. A syntax-directed approach to the semantics of concurrent languages. In *Proc. IFIP Congress 1986*, pages 571–576. North–Holland, 1986.

[30] E. Astesiano and G. Reggio. Direct semantics for concurrent languages in the SMoLCS approach. *IBM Journal of Research and Development*, 31(5), 1987.

[31] E. Astesiano and G. Reggio. A Metalanguage for the Formal Requirement Specification of Reactive Systems. In J. C. P. Woodcock and P. Larsen, editors, *FME'93: Industrial-Strength Formal Methods. First International Symposium of Formal Methods Europe*, pages 110–128. Springer LNCS 670, 1993. ⊕.

[32] E. Astesiano and G. Reggio. Algebraic specification of concurrency. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991.*, pages 1–39. Springer LNCS 655, 1993. ⊕.

[33] E. Astesiano and G. Reggio. Specifying Reactive Systems by Abstract Events. In *Proc. of Seventh International Workshop on Software Specification and Design (IWSSD-7)*, Los Alamitos, CA, 1993. IEEE Computer Society. ⊕.

[34] E. Astesiano and G. Reggio. A Case Study in Friendly Specifications of Concurrent Systems (Lamport & Broy's Specification Problem Presented at the Dagstuhl Seminar "Specification and Refinement of Reactive Systems – A Case Study"). Technical Report DISI–TR–94–21, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994. ⊕.

[35] E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995, ⊕.

[36] E. Astesiano and G. Reggio. Algebraic Dynamic Specifications: An Outline. Technical Report DISI–TR–95–08, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, 1995. ⊕.

[37] E. Astesiano, G. Reggio, and A. Tarlecki, editors. *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types.* Springer LNCS 906, 1995. ⊕.

[38] E. Astesiano, G. Reggio, and E. Zucca. Stores as Homomorphisms and their Transformations. In A. M. Bozyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993. 18th International Symposium, MFCS '93, Gdaánsk, Poland, August/September 1993*, pages 242–251. Springer LNCS 711, 1993. ⊕.

[39] E. Astesiano, G. Reggio, and E. Zucca. Stores as Homomorphisms and their Transformations. Technical Report DISI-TR-94-14, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy, 1994. Submitted for publication, ⊕.

[40] E. Astesiano and M. Wirsing. Bisimulation in algebraic specifications. In *Proc. Colloq. on Resolution of Equations in Algebraic Structures*. Academic Press, 1989. ⊕.

[41] E. Astesiano and E. Zucca. A Semantic Model for Dynamic Systems. In U. Lipeck and B. Thalheim, editors, *Modelling Database Dynamics, Volkse 1992*, pages 63–83. Springer-Verlag, Workshops in Computing, 1993. ⊕.

[42] E. Astesiano and E. Zucca. D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science*, 5:257–282, 1995. ⊕.

[43] E. Astesiano and E. Zucca. A Free Construction of Dynamic Terms. *Journal of Computer and System Sciences*, 52(1):143–156, 1996. ⊕.

[44] P. R. J. Asveld and J. V. Tucker. Complexity theory and the operational structure of algebraic programming systems. *Acta Informatica*, 17:451–76, 1982.

[45] R. Aubin. Mechanizing Structural Induction. *Theoretical Computer Science*, 9:329–362, 1979.

[46] L. Bachmair. Proof by consistency in equational theories. In *3rd Annual Symposium on Logic in Computer Science, Edinburgh, Scotland, July 5–8, 1988*, pages 228–233. IEEE, Computer Society Press, 1988.

[47] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the Association for Computing Machinery*, 21(8):613–641, 1978.

[48] J. Baeten and W. Weijland. *Process Algebra.* Cambridge University Press, Cambridge, 1990.

[49] R. Bardohl. $\mathrm{G_VT}$ - user manual. Technical report, TU Berlin, FB 13, Sept. 1995. ⊕.

[50] T. Bastiansen. Parametric Subtypes in ABEL. Research Report 207, Dept. of Informatics, Univ. of Oslo, 1995. available by http://www.ifi.uio.no/∼ftp/publications/research-reports/Bastiansen-1.ps. ⊕.

[51] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, Oct. 1992.

[52] E. Battiston, F. D. Cindio, and G. Mauri. OBJSA nets: a class of high-level nets having abstract data types as domains. In *Advances in Petri Nets '88*. Springer LNCS 340, 1988.

[53] B. Bauer and R. Hennicker. Proving the correctness of algebraic implementations by the ISAR system. In A. Miola, editor, *Proc. DISCO '93, International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 2–16. Springer LNCS 722, 1993. ⊕.

[54] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L.* Springer LNCS 183, 1985.

[55] F. L. Bauer, M. Broy, W. Dosch, T. Gnatz, B. Krieg-Brückner, A. Laut, M. Luckmann, T. A. Matzner, B. Möller, H. Partsch, P. Pepper, K. Samelson, R. Steinbrüggen, M. Wirsing, and H. Wössner. Programming in a wide spectrum language: a collection of examples. *Science of Computer Programming*, 1:73–114, 1981.

[56] F. L. Bauer, H. Partsch, P. Pepper, and H. Wössner. Techniques for program development. Software Engineering Techniques. Infotech State of the Art Report 34, Infotech International, 1977.

[57] F. L. Bauer, H. Partsch, P. Pepper, and H. Wössner. A transformational approach to programming. In *Proc. 3rd Symp. on Programming*, pages 248–262. Dunod, 1978.

[58] F. L. Bauer and M. Wirsing. Crypt-equivalent algebraic specifications. *Acta Informatica*, 25(2):111–153, 1988.

[59] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer, 1982.

[60] H. Baumeister. Relations as Abstract Datatypes: An Institution to Specify Relations between Algebras. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 756–771. Springer LNCS 915, 1995. ⊕.

[61] H. Baumeister. Using Algebraic Specification Languages for Model-Oriented Specifications. Technical Report MPI-I-96-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, Feb. 1996. ⊕.

[62] C. Beierle and A. Voß. Viewing implementations as an institution. In *Proc. Summer Conf. on Category Theory and Computer Science*, pages 196–218. Springer LNCS 283, 1987.

[63] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 1996. To appear. ⊕.

[64] J. Bergstra, J. Heering, and P. Klint. ASF - an Algebraic Specification Formalism. Technical Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[65] J. Bergstra and J. Klop. Process Algebra for Synchronous Communication. *Information & Control*, 60(1/3):109–137, 1984.

[66] J. A. Bergstra, M. Broy, J. V. Tucker, and M. Wirsing. On the power of algebraic specifications. In *Proc. 1981 Symp. on Mathematical Foundations of Computer Science*, pages 193–204. Springer LNCS 118, 1981.

[67] J. A. Bergstra, J. Heering, and P. Klint. Module Algebra. *Journal of the Association for Computing Machinery*, 37:335–372, 1990.

[68] J. A. Bergstra, J. Heering, and R. Klint, editors. *Algebraic Specification*. Addison-Wesley, 1989.

[69] J. A. Bergstra and J. W. Klop. Initial algebra specifications for parameterized data types. Technical Report IW 186, Mathematical Centrum, Dept. of Computer Science, Amsterdam, 1981.

[70] J. A. Bergstra and J. V. Tucker. Initial and final algebra semantics: two characterization theorems. *SIAM Journal of Computation*, 12(2):366–387, 1983.

[71] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.

[72] G. Bernot. Good functors ... are those preserving philosophy. In *Proc. Summer Conf. on Category Theory and Computer Science*, pages 182–195. Springer LNCS 283, 1987.

[73] G. Bernot. Testing Against Formal Specification: a Theoretical View. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91. Proccedings of the International Joint Conference on Theory and Practice of Software Development. Volume 1: Colloquium on Trees in Algebra and Programming*, pages 103–120. Springer LNCS 493, 1991. ⊕.

[74] G. Bernot and M. Bidoit. Proving the Correctness of Algebraically Specified Software: Modularity and Observability Issues. Rapport de Recherche du LIENS LIENS–91–8, Ecole Normale Supérieure, 1991. ⊕.

[75] G. Bernot, M. Bidoit, and C. Choppy. Abstract data types with exception handling: an initial approach based on a distinction between exceptions and errors. *Theoretical Computer Science*, 46(1):13–46, 1986. ⊕.

[76] G. Bernot, M. Bidoit, and C. Choppy. Abstract implementations and correctness proofs. In *Proc. 3rd Symp. on Theoretical Aspects of Computer Science*, pages 236–251. Springer LNCS 210, 1986.

[77] G. Bernot, M. Bidoit, and T. Knapik. Behavioural approaches to algebraic specifications. a comparative study. *Acta Informatica*, 31:651–671, 1994. ⊕.

[78] G. Bernot and P. L. Gall. Exception Handling and Term Labelling. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development. 4th International Joint Conference CAAP/FASE*, pages 421–436. Springer LNCS 668, 1993. ⊕.

[79] G. Bernot and P. L. Gall. Label algebras: a systematic use of terms. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 144–163. Springer LNCS 655, 1993. ⊕.

[80] G. Bernot, P. L. Gall, and M. Aiguier. Label algebras and exception handling. *Science of Computer Programming*, 23(2/3):227–286, 1994. ⊕.

[81] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991. ⊕.

[82] D. Bert, P. Drabik, and R. Echahed. LPG: A generic, logic and functional programming language. In *Proc. of the 4th Symposium on Theoretical Aspects of Computing Science (STACS 87), LNCS 247*. Springer-Verlag, 1987.

[83] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. of ESOP-86, LNCS 213*, pages 119–132. Springer-Verlag, 1986.

[84] D. Bert and R. Echahed. On the operational semantics of the algebraic and logic language LPG. In *Recent Trends in Data Types Specification, LNCS 906, 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, selected papers*, pages 132–152. Springer-Verlag, 1995.

[85] D. Bert and R. Echahed. On the Operational Semantics of the Algebraic and Logic Programming Language LPG. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 132–152. Springer LNCS 906, 1995.

[86] D. Bert and C. Oriat. A Model Inference System for Generic Specification with Application to Code Sharing. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 741–755. Springer LNCS 915, 1995. ⊕.

[87] D. Bert and C. Oriat. A model inference system for generic specification with application to code sharing. In *Proc. of TAPSOFT-95, Col. on Formal Approaches in Software Engineering, LNCS 915*, pages 741–755. Springer-Verlag, 1995.

[88] R. Betschko, S. Dick, K. Didrich, and W. Grieskamp. Formal Development of an Efficient Implementation of a Lexical Scanner within the KorSo Methodology Framework. Technical Report 93–30, TU Berlin, Oct. 1993. ⊕.

[89] M. Bettaz. An association of algebraic term nets and abstract data types. In *Recent Trends in Data Type Specification. 7th Workshop on Specification of Abstract Data Types*, pages 11–30. Springer LNCS 534, 1991.

[90] M. Bettaz and M. Maouche. How to specify non determinism and true concurrency with algebraic term nets. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 164–180. Springer LNCS 655, 1993.

[91] M. Bettaz and G. Reggio. A SMoLCS Based Kit for Defining the Semantics of High-Level Algebraic Petri Nets. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, 1992*, pages 98–112. Springer LNCS 785, 1994. ⊕.

[92] W. Bibel and K. M. Hoernig. LOPS — a system based on a strategical approach to program synthesis. In A. W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 69–91. Macmillan, 1984.

[93] M. Bidoit. PLUSS, un langage pour le développement de spécifications algébriques modulaires. Thèse d'Etat, Université Paris-Sud, Orsay, 1989. ⊕.

[94] M. Bidoit. Some key issues in designing an algebraic specification language. In *Proceedings of the International Congress on New Technologies for Software and Supercomputer Developments (ICNTSSD'89), Caracas, Venezuela*, Nov. 1989. ⊕.

[95] M. Bidoit. The stratified loose semantics: An attempt to provide an adequate algebraic model of modularity. In *Proceedings of the 1st International Conference on Algebraic Methodology and Software Technology. Iowa City, U.S.A.*, pages 77–83, 1989. ⊕.

[96] M. Bidoit, B. Biebow, M.-C. Gaudel, C. Gresse, and G. Guiho. Exception handling: formal specification and systematic program construction. *IEEE Transactions on Software Engineering*, SE-11(3):242–252, 1985.

[97] M. Bidoit, F. Capy, and C. Choppy. The design and specification of the ASSPEGIQUE database. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems. International Symposium DISCO 1990*, pages 205–214. Springer LNCS 429, 1990. ⊕.

[98] M. Bidoit and C. Choppy. ASSPEGIQUE: an integrated environment for algebraic specifications. In *Proceedings of the Joint Conference on Theory and Practice of Software Development. TAPSOFT '85*, pages 246–260. Springer LNCS 186, 1985.

[99] M. Bidoit and C. Choppy, editors. *Recent Trends in Data Type Specification. Dourdan France, August 1991.* Springer LNCS 655, 1993. ⊕.

[100] M. Bidoit, C. Choppy, and F. Voisin. The ASSPEGIQUE specification environment: motivations and design. In *Recent Trends in Data Type Specification, Proc. 3rd Workshop on Theory and Applications of Abstract Data Types*, pages 54–72. Springer Informatik Berichte 116, 1985.

[101] M. Bidoit, C. Choppy, and F. Voisin. Validation d'une spécification algébrique du "Transit-node" par prototypage et démonstration. Chapitre du Rapport final de l'Opération VTT, Validation et vérification de propriétés Temporelles et de Types de données (commune aux PRC PAOIA et C3), LaBRI, Bordeaux, 1994. ⊕.

[102] M. Bidoit, C. Choppy, and F. Voisin. Interchange Format for Inter-Operability of Tools and Translation. The SALSA and ASSPEGIQUE$^+$/LP Experience. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 102–124. Springer–Verlag, 1996. To appear. ⊕.

[103] M. Bidoit, M.-C. Gaudel, J. Hagelstein, A. Mauboussin, and H. Perdrix. From an ERAE Requirements Specification to a PLUSS Algebraic Specification: A Case Study. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, pages 395–397. Springer LNCS 490, 1991. ⊕.

[104] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? an experiment with the PLUSS specification language. *Science of Computer Programming*, 12:1–38, 1989. ⊕.

[105] M. Bidoit and R. Hennicker. How to prove observational theorems with LP. In U. Martin and J. Wing, editors, *Proc. First International Workshop on Larch, Boston, 1992*. Springer Verlag Workshop in Computing Series, 1992. ⊕.

[106] M. Bidoit and R. Hennicker. A General Framework for Modular Implementations of Modular System Specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development. 4th International Joint Conference CAAP/FASE*, pages 199–214. Springer LNCS 668, 1993. ⊕.

[107] M. Bidoit and R. Hennicker. Characterizing behavioural semantics and abstractor semantics. In D. Sannella, editor, *Proc. ESOP '94, 5th European Symposium on Programming*, pages 105–119. Springer LNCS 788, 1994. ⊕.

[108] M. Bidoit and R. Hennicker. Proving behavioural theorems with standard first-order logic. In M. Rodriguez-Artalejo, editor, *Proc. Algebraic and Logic Programming, 4th International Conference, ALP '94, Algebraic and Logic Programming*, pages 41–58. Springer LNCS 850, 1994. ⊕.

[109] M. Bidoit and R. Hennicker. Behavioural Theories. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 153–169. Springer LNCS 906, 1995. ⊕.

[110] M. Bidoit and R. Hennicker. Behavioural Theories and the Proof of Behavioural Properties. Rapport de Recherche LIENS-95-5, LIENS, Mar. 1995. ⊕.

[111] M. Bidoit and R. Hennicker. Proving the Correctness of Behavioural Implementations. In *Proc. AMAST '95, 4th International Conference on Algebraic Methodology and Software Technology*, pages 152–168. Springer LNCS 936, 1995. ⊕.

[112] M. Bidoit and R. Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 1996. To appear. ⊕.

[113] M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25(2–3):149–186, 1995. ⊕.

[114] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development*. Springer LNCS 501, 1991. ⊕.

[115] M. Bidoit and A. Tarlecki. Behavioural satisfaction and equivalence in concrete model categories. In *Proc. CAAP '96*, 1996. To appear. ⊕.

[116] B. Biebow and J. Hagelstein. Algebraic specification of synchronisation and errors: a telephonic example. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 294–308. Springer LNCS 186, 1985.

[117] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[118] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–218. Springer, 1989.

[119] S. Biundo. A synthesis system mechanizing proofs by induction. In *Proc. 1986 European Conf. on Artificial Intelligence*, pages 69–78, 1986.

[120] E. K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnections. *Journal of Computer and System Sciences*, 34:293–339, 1987.

[121] B. Bogaert and S. Tison. Equality and Disequality Constraints on direct subterms in Tree Automata. In A. Finkel and M. Jantzen, editors, *STACS 92. 9th Annual Symposium on Theoretical Aspects of Computer Science*, pages 161–172. Springer LNCS 577, 1992.

[122] K. Böhm and A. Sernadas. A Logic to Specify Real-time Object Behaviour. In A. Verbraeck, H. Sol, and P. Bots, editors, *Proceedings of the Fourth International Working Conference on Dynamic Modelling and Information Systems*, pages 53–69. Delft University Press, 1994. ⊕.

[123] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In J. Staudek, editor, *Proceedings of SOFSEM'95*, pages 363–368. Springer LNCS 1012, 1995. ⊕.

[124] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.

[125] A. Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles.* PhD thesis, Université Henri Poincaré – Nancy 1, Mar. 1994. ⊕.

[126] A. Bouhoula. SPIKE: a system for sufficient completeness and parameterized inductive proof. In A. Bundy, editor, *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, pages 836–840. Springer LNAI 814, June 1994. ⊕.

[127] A. Bouhoula. Sufficient Completeness and Parameterized Proofs by Induction. In M. Rodriguez-Artalejo, editor, *Proc. Algebraic and Logic Programming, 4th International Conference, ALP '94, Algebraic and Logic Programming*, pages 23–40. Springer LNCS 850, Sept. 1994. ⊕.

[128] A. Bouhoula. Using Induction and Rewriting to Verify and Complete Parameterized Specifications. *Theoretical Computer Science*, 170, Dec. 1996. To appear. ⊕.

[129] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated Mathematical Induction. *Journal of Logic and Computation*, 5(5):631–668, 1995. ⊕.

[130] A. Bouhoula and M. Rusinowitch. Automatic Case Analysis in Proof by Induction. In R. Bajcsy, editor, *Proceedings 13th International Joint Conference on Artificial Intelligence, Chambéry (France)*, volume 1, pages 88–94. Morgan Kaufmann, Aug. 1993. ⊕.

[131] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995. ⊕.

[132] A. Bouhoula and M. Rusinowitch. SPIKE: a system for automatic inductive proofs. In V. Alagar and M. Nivat, editors, *Proccedings of the fourth International Conference on Algebraic Methodology and Software Technology Montreal (Canada).* Springer LNCS 936, July 1995. ⊕.

[133] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.

[134] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[135] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. 3rd IEEE Symp. on Logic in Computer Science*, 1988.

[136] V. Breazu-Tannen and T. Coquand. Extensional models of polymorphism. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 291–307. Springer LNCS 250, 1987.

[137] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *Proc. 16th Intl. Colloq. on Automata, Languages and Programming*, pages 137–150. Springer LNCS 372, 1989.

[138] M. Breu, M. Broy, T. Grünler, and F. Nickl. Semantics of PA^{nn}dA-S. PROSPECTRA Report M.1.1.S3-R-56.2, Universität Bremen, 1990. ⊕.

[139] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming.* Springer LNCS 562, 1991. ⊕.

[140] R. Breu and E. Zucca. An algebraic compositional semantics of an object-oriented notation with concurrency. In *Proc. 9th Conf. on Foundations of Software Technology and Theoretical Computer Science.* Springer LNCS 405, 1989. ⊕.

[141] D. Briaud. An explicit *Eta* rewrite rule. In M. Dezani, editor, *International Conference on Typed Lambda Calculus and Applications*, pages 94–108. Springer LNCS 902, 1995. ⊕.

[142] E. Brinksma. A theory for the derivation of tests. In *8th International Conference on Protocol Specification, Testing and Verification.* North-Holland, 1988.

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

[144] M. Broy. Partial interpretations of higher order algebraic types. In *Lecture Notes of the International Summer School on Logic of Programming and Calculi of Discrete Design, Marktoberdorf*, 1986.

[145] M. Broy. Compositional Refinement of Interactive Systems. Technical Report SRC 89, DIGITAL Systems Research Center, 1992. ⊕.

[146] M. Broy. Functional Specification of Time Sensitive Communicating Systems. REX Workshop. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 153–179. Springer LNCS 430, 1993. Extended version: ACM Transactions on Software Engineering and Methodology 2:1, January 1993, 1–46. ⊕.

[147] M. Broy. (Inter-) Action Refinement: The Easy Way. In M. Broy, editor, *Proc. Program Design Calculi, Summerschool, Marktoberdorf*, pages 121–158. Springer, 1993. ⊕.

[148] M. Broy. A Functional Rephrasing of the Assumption/Commitment Specification Style. TUM-I 9417, Technische Universität München, Institut für Informatik, June 1994. ⊕.

[149] M. Broy. Advanced Component Interface Specification. In T. Ito and A. Yonezawa, editors, *Proc. TPPP'94*, pages 369–392. Springer LNCS 907, 1995. ⊕.

[150] M. Broy. Equations for Describing Dynamic Nets of Communicating Systems. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 170–187. Springer LNCS 906, 1995. ⊕.

[151] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. Summary of Case Studies in Focus - a Design Method for Distributed Systems. Technical Report SFB 342/3/92 A, Technische Universität München, 1992. ⊕.

[152] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems — An Introduction to Focus (Revised Version). Technical Report SFB 342/2/92 A, Technische Universität München, 1993. ⊕.

[153] M. Broy, W. Dosch, H. Partsch, P. Pepper, and M. Wirsing. Existential quantifiers in abstract data types. In *Proc. 6th Intl. Colloq. on Automata, Languages and Programming*, pages 73–87. Springer LNCS 71, 1979.

[154] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. (Version 1.0). Part I, II. Technical Report I9311,I9312, Technische Universität München, May 1993. ⊕.

[155] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. (Version 1.0). Part I. TUM-I 9311, Technische Universität München, May 1993. ⊕.

[156] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. (Version 1.0). Part II. TUM-I 9312, Technische Universität München, May 1993. ⊕.

[157] M. Broy, M. Fuchs, T. Gritzner, B. Schätz, K. Spies, and K. Stølen. Summary of Case Studies in Focus - a Design Method for Distributed Systems. Technical Report SFB 342/13/94 A, Technische Universität München, 1994. ⊕.

[158] M. Broy, B. Möller, P. Pepper, and M. Wirsing. Algebraic implementations preserve program correctness. *Science of Computer Programming*, 7:35–53, 1986. ⊕.

[159] M. Broy, C. Pair, and M. Wirsing. A systematic study of models of abstract data type. *Theoretical Computer Science*, 33:139–174, 1984.

[160] M. Broy, H. Partsch, P. Pepper, and M. Wirsing. Semantic relations in programming languages. In *Proc. IFIP Congress 1980*, pages 101–106. North-Holland, 1980.

[161] M. Broy and P. Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, SE-7(1):14–22, 1981.

[162] M. Broy and P. Pepper. Combining algebraic and algorithmic reasoning: an approach to the Schorr-Waite algorithm. *ACM Transactions on Programming Languages and Systems*, 4(3):362–381, 1982.

[163] M. Broy and P. Pepper. On the coherence of programming language and programming methodology. In *Proc. IFIP TC2 Working Conf. on Programming Languages and System Design*, pages 41–53. North-Holland, 1983.

[164] M. Broy, P. Pepper, and M. Wirsing. On relations between programs. In *Proc. 4th Symp. on Programming*, pages 59–78. Springer LNCS 83, 1980.

[165] M. Broy, P. Pepper, and M. Wirsing. Programming in a wide-spectrum language. In J. W. de Bakker and J. C. von Vliet, editors, *Algorithmic languages*, pages 73–114. North-Holland, 1981.

[166] M. Broy, P. Pepper, and M. Wirsing. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9:54–99, 1987.

[167] M. Broy and K. Stølen. Specification and Refinement of Finite Dataflow Networks — a Relational Approach. Technical Report SFB 342/7/94 A, Technische Universität München, 1994. ⊕.

[168] M. Broy and K. Stølen. Specification and Refinement of Finite Dataflow Networks — a Relational Approach. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. FTRTFT'94*, pages 247–267. Springer LNCS 863, 1994. ⊕.

[169] M. Broy and M. Wirsing. Programming languages as abstract data types. In *Proc. 1980 Colloq. on Trees in Algebra and Programming*, pages 160–177, 1980.

[170] M. Broy and M. Wirsing. On the algebraic specification of nondeterministic programming languages. In *Proc. 1981 Colloq. on Trees in Algebra and Programming*. Springer LNCS 112, 1981.

[171] M. Broy and M. Wirsing. Partial abstract data types. *Acta Informatica*, 18(1):47–64, 1982.

[172] M. Broy and M. Wirsing. Algebraic definition of a functional programming language and its semantic models. *RAIRO Informatique Théorique*, 17:137–161, 1983.

[173] M. Broy and M. Wirsing. Generalized heterogeneous algebras and partial interpretations. In *Proc. 1983 Colloq. on Trees in Algebra and Programming*, pages 1–34. Springer LNCS 159, 1983.

[174] M. Broy and M. Wirsing. Ultra–loose algebraic specification. *Bulletin of the EATCS*, 35:117–128, 1988.

[175] M. Broy and M. Wirsing. Correct Software: From Experiments to Applications. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, pages 1–24. Springer LNCS 1009, 1995. ⊕.

[176] Ø. Brusevold. Programmering i Safir. Master's thesis, Universitetet i Bergen, 1993. ⊕.

[177] G. Bueltzingsloewen. Translating and optimizing SQL queries having aggregates. In *Proc. 13th Conf. on Very Large Data Bases*, 1987.

[178] P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*. Akademie-Verlag, 1986.

[179] R. Burstall. Proving Properties of Programs by Structural Induction. *Computer Journal*, 12:41–48, 1969.

[180] R. Burstall and R. Diaconescu. Hiding and behaviour: an institutional approach. In A. W. Roscoe, editor, *A Classical Mind: Essays Dedicated to C.A.R. Hoare*. Prentice-Hall, 1994. ⊕.

[181] R. Burstall and J. Goguen. An informal introduction to specifications using Clear. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.

[182] R. Burstall and J. Goguen. Algebras, theories and freeness: An introduction for computer scientists. In M. Wirsing and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 329–350. Reidel, 1982. Proceedings, 1981 Marktoberdorf NATO Summer School, NATO Advanced Study Institute Series, Volume C91.

[183] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, pages 1045–1058, 1977.

[184] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Proc. Advanced Course on Abstract Software Specifications*, pages 292–332. Springer LNCS 86, 1980.

[185] R. M. Burstall and B. Lampson. A kernel language for abstract data types and modules. *Information and Computation*, 76:278–346, 1988.

[186] C. Caleiro. On the relationship between operational and denotational semantics of temporal logic specification of object behaviour. In R. Wieringa and R. Feenstra, editors, *IS-CORE'94 - Selected papers*, pages 69–83. World Scientific Publishers, 1995. ⊕.

[187] V. Carchiolo, A. Faro, F. Minassale, and G. Scollo. Some Topics in the Design of the Specification language LOTOS. In M. Paul and B. Robinet, editors, *Proc. 4th Int. Symp. on Programming*, Berlin, 1984. Springer LNCS 167.

[188] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[189] M. V. Cengarle. *Formal Specification with Higher–Order Parameterization*. PhD thesis, Ludwig–Maximilians–Universität München, 1994. ⊕.

[190] M. V. Cengarle. Semantic Typing for Parametric Algebraic Specifications. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '95, Proceedings)*, pages 261–276. Springer LNCS 936, July 1995. ⊕.

[191] M. V. Cengarle and L. Mandel. Finite Domains in the Constrained Lambda Calculus. In *Proceedings of the Post-Conference Workshop on Constraints, Databases and Logic Programming. International Logic Programming Symposium (ILPS)y*, pages 75–89, 1995. ⊕.

[192] M. V. Cengarle and M. Wirsing. A Calculus of Higher-Order Parameterization for Algebraic Specifications. *Bulletin of the Interest Group in Pure and Applied Logics (IGPL)*, 3(4):615–641, July 1995. Special Issue 'Workshop on Logic, Language, Information and Computation 1994'. ⊕.

[193] S. Ceri and G. Gottlob. Translating SQL into relational algebra: optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, SE-11, 1985.

[194] M. Cerioli. *Relationships between Logical Formalisms.* PhD thesis, Universities of Genova, Pisa and Udine, 1992. Available as internal report of Pisa University, TD-4/93 or by anonymous ftp at ftp.disi.unige.it in /pub/cerioli/thesis92.ps.z. ⊕.

[195] M. Cerioli. A Lazy Approach to Partial Algebras. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 188–202. Springer LNCS 906, 1995. ⊕.

[196] M. Cerioli and J. Meseguer. May I Borrow Your Logic? In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993; 18th International Symposium, MFCS'93 Gdansk, Poland, August 30 - September 3.* Springer LNCS 711, 1993. ⊕.

[197] M. Cerioli and J. Meseguer. May I Borrow Your Logic? (Transporting Logical Structures along Maps). *Theoretical Computer Science*, 1996. To appear, ⊕.

[198] M. Cerioli, T. Mossakowski, and H. Reichel. From total equational to partial first order. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications.* Chapman and Hall, 1996. to appear⊕.

[199] M. Cerioli and G. Reggio. Algebraic-Oriented Institutions. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, pages 103–210. Springer Workshops in Computing, 1994. ⊕.

[200] M. Cerioli and G. Reggio. Institutions for Very Abstract Specifications. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, 1992*, pages 113–127. Springer LNCS 785, 1994. ⊕.

[201] K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[202] H. Chen and J. Hsiang. Order-Sorted Equational Specification and Completion. Technical report, State University of New York at Stony Brook, Nov. 1991.

[203] P. Chen, R. Hennicker, and M. Jarke. On the retrieval of reusable software components. In W. F. R. Prieto-Diaz, editor, *Advances in Software Reuse. Selected Papers from the Second International Workshop on Software Reusability*, pages 99–108. IEEE Computer Society Press, Order Number 3130, 1993. ⊕.

[204] B. Chetali and P. Lescanne. An exercise in LP, the proof of a non restoring division circuit. In U. Martin and J. Wing, editors, *Proc. of First International Workshop on Larch.* Springer Verlag, 1992. ⊕.

[205] B. Chetali and P. Lescanne. Formal Verification of a Protocol for Communications over faulty channel. In *Proc. of 8th Int. Conf. on Formal Descriptions Techniques for Distributed Systems and Communications Protocols, FORTE'95*, 1995. ⊕.

[206] C. Choppy. Formal specification, prototyping and integration tests. In *Proc. European Software Engineering Conference*, pages 185–192, 1987.

[207] C. Choppy. Prototyping and formal specification. In C. Rattray and R. Clark, editors, *The unified computation laboratory*, pages 141–170. Oxford University Press, 1992. ⊕.

[208] C. Choppy. About the "correctness" and "adequacy" of PLUSS specifications. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, Spain, 1992, Selected Papers*, pages 128–143. Springer LNCS 785, 1994. ⊕.

[209] C. Choppy. Spécifications algébriques: Prototypage et validation. Habilitation à diriger des recherches, Université Paris-Sud, Orsay, 1994. ⊕.

[210] C. Choppy, D. Bert, M. Bidoit, R. Echahed, C. Roques, and F. Voisin. Rapid prototyping with algebraic specifications: A case study. Research Report 844, L.R.I., 1993. ⊕.

[211] C. Choppy and M. Bidoit. Integrating ASSPEGIQUE and LP. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch*, pages 69–85. Springer-Verlag Workshops in Computing, 1993. ⊕.

[212] C. Choppy and S. Kaplan. Mixing abstract and concrete modules : specification, development and prototyping. In *Proc. of the 12th International Conference on Software Engineering*, pages 173–184, 1990. ⊕.

[213] C. Choppy, S. Kaplan, and M. Soria. Complexity analysis of term–rewriting systems. *Theoretical Computer Science*, 67(2+3):261–282, 1989. ⊕.

[214] M.-A. Choquer. Specification of the evaluation tool: leftmost-outermost reduction strategy in the equational case. METEOR report, LRI, Université de Paris-Sud, Orsay, 1986.

[215] S. Cicerone and F. Parisi-Presicce. Strategies in modular system design by interface rewriting. In D. Sannella, editor, *Proc. ESOP '94, 5th European Symposium on Programming*, pages 165–179. Springer LNCS 788, 1994. ⊕.

[216] S. Cicerone and F. Parisi-Presicce. On the Complexity of Specification Morphisms. *Theoretical Computer Science*, 1996. Accepted for publication. ⊕.

[217] C. Cîrstea. A Distributed Semantics for FOOPS. Technical Report PRG-TR-20-95, Programming Research Group, University of Oxford, 1995. ⊕.

[218] I. Claßen. Revised ACT ONE: categorical constructions for an algebraic specification language. In *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology*, pages 124–141. Springer LNCS 393, 1989.

[219] I. Claßen. *Compositionality of Application Oriented Structuring Mechanisms for Algebraic Specification Languages with Initial Semantics.* PhD thesis, TU Berlin, FB Informatik, June 1993. ⊕.

[220] I. Claßen, H. Ehrig, B. Mahr, and F. Orejas. Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development. *British Computer Journal*, 35(5):460–467, 1992. ⊕.

[221] I. Claßen, H. Ehrig, B. Mahr, and F. Orejas. Introduction to Algebraic Specification. Part 2: From Classical View to Foundations of System Specification. *British Computer Journal*, 35(5):468–477, 1992. ⊕.

[222] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development: The ACT Approach*, volume 1 of *AMAST Series in Computing*. World Scientific Publishing Co. Pte. Ltd., 1993. ⊕.

[223] I. Claßen and M. Gogolla. Towards a Conceptual Model for the Environment of the Algebraic Specification Language ACT ONE. Forschungsberichte des Fachbereichs Informatik 91/11, Technische Universität Berlin, 1991. ⊕.

[224] I. Claßen, M. Große-Rhode, and U. Wolter. Categorical Concepts for Parameterized Partial Specifications. *Mathematical Structures in Computer Science*, 5:153–188, 1995. A preliminary version appeared as Tech. Report 92-42, Technische Universität Berlin, FB Informatik, 1992, ⊕.

[225] I. Claßen, M. Löwe, S. Waßerroth, and J. Wortmann. Static and Dynamic Semantics of Entity-Relationship Models Based on Algebraic Methods. In B. Wolfinger, editor, *Innovationen bei Rechen-und Kommunikationssystemen*, pages 2–10, 1994. ⊕.

[226] S. Clérici and F. Orejas. GSBL: an algebraic specification language based on inheritance. In *Proc. 1988 European Conf. on Object Oriented Programming*, pages 78–92. Springer LNCS 322, 1988.

[227] H. Comon. Solving Symbolic Ordering Constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[228] H. Comon. Completion of rewrite systems with membership constraints. In W. Kuich, editor, *Proceedings of ICALP 92*. Springer LNCS 623, 1992. ⊕.

[229] H. Comon and F. Jacquemard. Ground Reducibility and Automata with Disequality Constraints. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *STACS94. 11th Annual Symposium on Theoretical Aspects of Computer Science*, pages 151–162. Springer LNCS 775, Feb. 1994. ⊕.

[230] H. Comon, R. Nieuwenhuis, and A. Rubio. Orderings, AC-Theories and Symbolic Constraint Solving. In D. Kozen, editor, *Tenth Annual IEEE Symposium on Logic in Computer Science*, San Diego, June 1995. IEEE Computer Society Press. ⊕.

[231] S. Conrad. A Basic Calculus for Verifying Properties of Synchronously Interacting Objects. Informatik-Bericht 94–05, Technische Universität Braunschweig, 1994. ⊕.

[232] S. Conrad. On Certification of Specifications for TROLL *light* Objects. In H. Ehrig and F. Orejas, editors, *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (ADT'92)*, pages 158–172. Springer, LNCS 785, 1994.

[233] S. Conrad. Temporal Logic Specification of Objects: An Approach to Compositionality and Reusability Allowing `next` operators. In R. Wieringa and R. Feenstra, editors, *Working papers of the International Workshop on Information Systems - Correctness and Reusability*, pages 228–241. Vrije Universiteit Amsterdam, RapportNr. IR-357, 1994. ⊕.

[234] S. Conrad and H.-D. Ehrich. An Elementary Logic for Object Specification and Verification. In U. Lippek and G. Vossen, editors, *Workshop Formale Grundlagen für den Entwurf von Informationssystemen, Tutzing*, pages 197–206. Technical Report Univ. Hannover, No. 03/94, 1994. ⊕.

[235] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik–Berichte 92–02, Technische Universität Braunschweig, 1992. ⊕.

[236] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleavelend, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[237] P. D. M. (Coordinator). CoFI catalogue of existing specification languages. Preliminary version. WWW page with URL: `http://www.brics.dk/Projects/CoFi/Catalogue`, 1996. ⊕.

[238] P. D. M. (Coordinator). Initiative for a *Common Framework for Algebraic Specification*. WWW page with URL: `http://www.brics.dk/Projects/CoFi`, 1996. ⊕.

[239] T. Coquand and G. Huet. Constructions: a higher-order proof system for mechanizing mathematics. In *Proc. 1985 European Conf. on Computer Algebra*, pages 151–184. Springer LNCS 204, 1985.

[240] F. Cornelius, H. Ehrig, F. Orejas, and M. Baldamus. Abstract and behaviour module specifications. *Mathematical Structures in Computer Science*, 1995. Submitted for publication. ⊕.

[241] G. Costa and G. Reggio. Abstract Dynamic Data Types: a Temporal Logic Approach. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, pages 103–112. Springer LNCS 520, 1991. ⊕.

[242] G. Costa and G. Reggio. Specification of Abstract Dynamic DataTypes: A Temporal Logic Approach. *Theoretical Computer Science*, 1996. To appear. ⊕.

[243] J. Costa, A. Sernadas, and C. Sernadas. Data Encapsulation and Modularity: Three Views of Inheritance. In A. M. Bozyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993. 18th International Symposium, MFCS '93, Gdaánsk, Poland, August/September 1993*, pages 382–391. Springer LNCS 711, 1993. ⊕.

[244] J. Costa, A. Sernadas, and C. Sernadas. Object Inheritance Beyond Subtyping. *Acta Informatica*, 31:5–26, 1995. ⊕.

[245] J. F. Costa and A. Sernadas. Progress Assumption in Concurrent Systems. *Formal Aspects of Computing*, 7(1):18–36, 1995. Available as Research Report since 1993, $\oplus$.

[246] J.-F. Costa, A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object Interaction. In I. Havel and V. Koubek, editors, *Mathematical Foundations of Computer Science. 17th International Symposium*, pages 200–208. Springer LNCS 629, 1992. $\oplus$.

[247] J. N. Crossley, L. Mandel, and M. Wirsing. Untyped Constrained Lambda Calculus. Technical Report Number 9318, Ludwig-Maximilians-Universität München, Oct. 1993. $\oplus$.

[248] J. N. Crossley, L. Mandel, and M. Wirsing. First Order Constrained Lambda Calculus. In *Proceedings of the Conference Frontiers on Combining Systems (FroCOs)*, 1996. $\oplus$.

[249] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. RR 1617, INRIA, Rocquencourt, Feb. 1992. To appear in J. of the ACM.

[250] R. Curien, Z. Qian, and H. Shi. Efficient second-order matching. In *Proc. 7th Int. Conf. on Rewriting Techniques and Application*. Springer LNCS, 1996. To appear. $\oplus$.

[251] V. Čyras and M. Haveraaen. Modular programming of recurrencies: a comparison of two approaches. *Informatica*, 6(4):397–444, 1995.

[252] V. Čyras and M. Haveraaen. The Structural Blanks approach to solving Generalized Recurrences. Technical report 100, Department of Informatics, University of Bergen, N-5020 BERGEN, Norway, 1996.

[253] O.-J. Dahl. Object Orientation and Formal Techniques. In *Formal Methods in Software Development (VDM'90)*, pages 1–11. Springer LNCS 428, 1990. $\oplus$.

[254] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992. Revised version 1993. $\oplus$.

[255] O.-J. Dahl. Value types and object classes. In *Proceedings of the 4. Nordic Workshop on Program Correctness, Bergen, Nov. 1992*, pages 111–122. Dept. of Informatics, Univ. of Bergen, Technical Report 78, 1993. $\oplus$.

[256] O.-J. Dahl. Relating a Simulation Model to an Applicative Specification. In M. Snorek, M. Sujansky, and A. Verbraeck, editors, *Modelling and Simulation 1995 (ESM 95, Prague)*, pages 633–637. Society for Computer Simulation International, 1995. $\oplus$.

[257] O.-J. Dahl and B. Kristoffersen. On Introducing Higher Order Functions in ABEL. Research Report 210, Dept. of Informatics, Univ. of Oslo, 1995. Presented at the 10th COMPASS WADT '95, Lisboa. Submitted to Nordic Journal of Computation. Available by http://www.ifi.uio.no/~ftp/publications/research-reports/Dahl-1.ps. $\oplus$.

[258] O.-J. Dahl, B. Myrhaug, and K. Nygaard. Simula 67 common base language. Report S-22, Norwegian Computing Center, Oslo, 1970. revised edition 1984.

[259] O.-J. Dahl and O. Owe. Formal Development with ABEL. In *Formal Software Development Methods (VDM'91)*, pages 320–362. Springer LNCS 552, 1991. $\oplus$.

[260] O.-J. Dahl and O. Owe. On the use of subtypes in ABEL. Research report no 206, Dept. of Informatics, Univ. of Oslo, 1995. Available by http://www.ifi.uio.no/~ftp/publications/research-reports/Owe-1.ps $\oplus$.

[261] P. Dauchy and M. Gaudel. Implicit State in Algebraic Specifications. In *Proc. ISCORE '93 (International Workshop on Information Systems - Correctness and Reusability)*, number 01 in Informatik–Berichte. Institut für Informatik, Hannover, 1993. $\oplus$.

[262] P. Dauchy and M.-C. Gaudel. Algebraic specifications with implicit state. Technical report, Univ. Paris-Sud, 1994. $\oplus$.

[263] P. Dauchy, M.-C. Gaudel, and B. Marre. Using Algebraic Specifications in Software Testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229–244, 1993. $\oplus$.

[264] P. Dauchy and B. Marre. Test data selection from algebraic specifications: application to an automatic subway module. In *3rd European Software Engineering Conference (ESEC'91)*, pages 80–100. Springer LNCS 550, 1991. ⊕.

[265] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[266] O. Declerfayt, B. Demeuse, F. Wautier, P.-Y. Schobbens, and E. Milgrom. Precise standards through formal specifications: a case study: the UNIX file system. In *Proceedings of the Autumn'88 EUUG Conference, Cascais, Portugal*, pages 115–130, 1988.

[267] G. Denker. Towards Semantic Foundation of Basic TROLL *light* Constructs. In U. Lipeck and R. Manthey, editors, *4. GI Workshop Grundlagen von Datenbanken*, pages 22–26. ECRC, München, Technical Report ECRC-92-13, 1992. ⊕.

[268] G. Denker. Transactions in Object–Oriented Specifications. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 203–218. Springer LNCS 906, 1995. ⊕.

[269] G. Denker. Reification – Changing Viewpoint but Preserving Truth. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 182–199. Springer–Verlag, 1996. To appear. ⊕.

[270] G. Denker and H.-D. Ehrich. Action Reification In Object Oriented Specification. In R. Wieringa and R. Feenstra, editors, *Selected Papers from ISCORE '94 (International Workshop on Information Systems - Correctness and Reusability)*, pages 103–118, Singapore, 1994. World Scientific. ⊕.

[271] G. Denker and H.-D. Ehrich. An Event-Based Semantics for Transactions. In G. Bernot and M. Aiguier, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability (IS-CORE'95), Technical Report, Evry, Sept. 1995*, pages 57–72. Universite d'Evry Val d'Essonne, Laboratoire de Mathematiques et d'Informatique, 1995. ⊕.

[272] G. Denker and M. Gogolla. Translating TROLL *light* Concepts to Maude. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, 1992*, pages 173–187. Springer LNCS 785, 1994. ⊕.

[273] N. Dershowitz. Synthesis by completion. In *Proc. 9th Intl. Joint Conf. on Artificial Intelligence*, pages 208–214, 1985.

[274] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, pages 69–116, 1987.

[275] N. Dershowitz and C. Hoot. Natural Termination. *Theoretical Computer Science*, 142(2):179–207, May 1995.

[276] J. Despeyroux-Savonitto. An algebraic specification of a Pascal compiler. *ACM SIGPLAN Notices*, 18(12):34–48, 1983.

[277] R. Diaconescu. *Category-Based Operational Semantics of Equational Logic Programming*. PhD thesis, Programming Research Group, Oxford University, 1994. ⊕.

[278] R. Diaconescu. Completeness of Category-based Equational Deduction. *Mathematical Structures in Computer Science*, 5(1):9–40, 1994. ⊕.

[279] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical Support for Modularisation. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993. ⊕.

[280] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods. First International Symposium of Formal Methods Europe*, pages 268–284. Springer LNCS 670, 1993. ⊕.

[281] K. Didrich, J. Exner, C. Gerke, W. Grieskamp, C. Maeder, P. Pepper, and S. Mario. The OPAL 2 Language — Alpha Version. Technical Report 96-3, TU Berlin, 1996. to appear,⊕.

[282] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In J. Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, pages 228–244. Springer LNCS 782, 1994. ⊕.

[283] Z. Diskin. Algebraizing Institutions: Incorporating Algebraic Logic Methodology into the Institution Framework for Building Specifications. Technical report, Laboratory for Database Design, Riga, Latvija, 1994.

[284] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner. The COQ Proof Assistant User's Guide. Version 5.6. Rapports Techniques 134, INRIA, 1991.

[285] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions, Extended abstract. In D. Kozen, editor, *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 366–374, San Diego, June 1995. IEEE Computer Society Press.

[286] H.-D. Ehirch. Fundamentals of Object-Oriented Information Systems Specification and Design: The OBLOG/TROLL Approach. *Nuclear Instruments and Methods in Physics Research A*, pages 375–278, 1994. ⊕.

[287] H. Ehler. Making formal specifications readable. Report TUM-I8527, Technische Universität München, 1985.

[288] H.-D. Ehrich. On the theory of specification, implementation and parameterization of abstract data types. *Journal of the Association for Computing Machinery*, 29:206–227, 1982.

[289] H.-D. Ehrich. Object Specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *IFIP 14.3 Volume on Foundations of System Specification, Chapter 11*. Springer LNCS, 1996. To appear. ⊕.

[290] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development. 4th International Joint Conference CAAP/FASE*, pages 453–467. Springer LNCS 668, 1993. ⊕.

[291] H.-D. Ehrich, M. Gogolla, and U. W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen*. Teubner-Verlag, Stuttgart, 1989.

[292] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorical Theory of Objects as Observed Processes. In J. de Bakker, W. deRoever, and G. Rozenberg, editors, *Proceedings REX/FOOL, Noordwijterhout*. Springer LNCS 489, 1990. ⊕.

[293] H.-D. Ehrich and P. Hartel. Temporal Specification of Information Systems. In H. Lin, editor, *Proc. Workshop honoring the 70th birthday of Prof. C. S. Tang, Beijing, August 1995*, 1996. To appear. ⊕.

[294] H.-D. Ehrich, R. Jungclaus, and G. Denker. Object Roles and Phases. In U. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, pages 114–121, 1993. ⊕.

[295] H.-D. Ehrich, R. Jungclaus, G. Denker, and A. Sernadas. Object-Oriented Design of Information Systems: Theoretical Foundations. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, pages 201–218. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1993. ⊕.

[296] H.-D. Ehrich, G. Saake, and A. Sernadas. Concepts of Object-Orientation. In *Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany)*, pages 1–19. Springer IFB 303, 1992. ⊕.

[297] H.-D. Ehrich and A. Sernadas. Algebraic Implementation of Objects over Objects. In J. de Bakker, W. deRoever, and G. Rozenberg, editors, *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness"*. Springer LNCS 430, 1990. ⊕.

[298] H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. In G. Saake and A. Sernadas, editors, *Information Systems – Correctness and Reusability*, pages 1–24. TU Braunschweig, Informatik Bericht 91-03, 1991. ⊕.

[299] H.-D. Ehrich and A. Sernadas. Local Specification of Distributed Families of Sequential Objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Selected papers*, pages 219–235. Springer LNCS 906, 1995. ⊕.

[300] H.-D. Ehrich, A. Sernadas, G. Saake, and C. Sernadas. Distributed Temporal Logic for Concurrent Object Families. In *Working papers of the International Workshop on Information Systems - Correctness and Reusability*, pages 22–30. Vrije Universiteit Amsterdam, Rapport Nr. IR-357, 1994. ⊕.

[301] H.-D. Ehrich, A. Sernadas, and C. Sernadas. Objects, Object Types, and Object Identification. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology*. Springer LNCS 393, 1989. ⊕.

[302] H.-D. Ehrich, A. Sernadas, and C. Sernadas. From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, 26(1-2):33–48, 1990. ⊕.

[303] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proceedings of the International Workshop on Graph Grammars and their Application to Computer Science and Biology*, pages 1–69. Springer LNCS 73, 1979.

[304] H. Ehrig. Algebraic theory of parameterized specifications with requirements. In *Proc. 1981 Colloq. on Trees in Algebra and Programming*, pages 1–24. Springer LNCS 112, 1981.

[305] H. Ehrig. Combining initial and loose algebraic specification methods including compositionality and modules. In *Proc. Workshop on Formal Software Development: Combining Specification Methods*, 1984.

[306] H. Ehrig, M. Baldamur, and F. Orejas. New concepts of amalgamation and extension for a general theory of specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 199–221. Springer LNCS 655, 1993. ⊕.

[307] H. Ehrig, M. Baldamus, and F. Cornelius. Theory of Algebraic Module Specification including Behavioural Semantics, Constraints and Aspects of Generalized Morphisms. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of 2nd International Conference on Algebraic Methodology and Software Technology*, pages 101–125, Iowa City, Iowa, USA, 1991. Springer. ⊕.

[308] H. Ehrig, M. Baldamus, and F. Orejas. New Concepts of Amalgamation and Extension for a General Theory of Specifications. In *Recent Trends in Data Type Specification, Proceedings ADT–COMPASS–Workshop Durdan'91*, pages 199–221. Springer LNCS 655, 1992. ⊕.

[309] H. Ehrig, M. Baldamus, and F. Orejas. New Concepts for Amalgamation and Extension in the Framework of Specification Logics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science. Essays and Tutorials*, pages 161–182. World Scientific Series in Computer Science — Vol. 40, 1993. ⊕.

[310] H. Ehrig and W. Fey. Methodology for the specification of software system: from formal requirements to algebraic design specifications. In *GI — 11. Jahrestagung*, pages 255–269. Springer IFB 50, 1981.

[311] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: an algebraic specification language with two levels of semantics. Technical Report 83-03, Technische Universität Berlin, 1983.

[312] H. Ehrig, W. Fey, H. Hansen, M. Löwe, and F. Parisi-Presicce. Algebraic theory of modular specification development. Technical Report 87-06, Technische Universität Berlin, 1987.

[313] H. Ehrig, W. Fey, and K. P. Hasler. Algebraische Spezifikationen: Konzepte und Sprachen für die Software-Entwicklung. In *Tagungsband "10 Jahre Informatik an der Universität Dortmund"*, 1983.

[314] H. Ehrig, W. Fey, and F. Parisi-Presicce. Distributive laws for composition and union of module specification for software systems. In *Proc. IFIP TC2 Working Conference on Program Specification and Transformation*, pages 293–312. North-Holland, 1986.

[315] H. Ehrig, W. Fey, F. Parisi-Presicce, and E. K. Blum. Algebraic theory of module specifications with constraints. In *Proc. 1986 Symp. on Mathematical Foundations of Computer Science*, pages 59–77. Springer LNCS 233, 1986.

[316] H. Ehrig and M. Große-Rhode. Functorial theory of parameterized specifications in a general specification framework. *Theoretical Computer Science*, 135(2):221–266, 1994. ⊕.

[317] H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors. *Recent Trends in Data Type Specification. 7th Workshop on Specification of Abstract Data Types.* Springer LNCS 534, 1991.

[318] H. Ehrig, A. Jimenez, and F. Orejas. Compositionality Results for Different Types of Parameterization and Parameter Passing in Specification Languages. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development. 4th International Joint Conference CAAP/FASE*, pages 31–45. Springer LNCS 668, 1993. ⊕.

[319] H. Ehrig and H.-J. Kreowski. Compatibility of parameter passing and implementation of parameterized data types. *Theoretical Computer Science*, 27(3):255–286, 1983.

[320] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209–263, 1982.

[321] H. Ehrig, H.-J. Kreowski, and P. Padawitz. Stepwise specification and implementation of abstract data types. In *Proc. 5th Intl. Colloq. on Automata, Languages and Programming*, pages 205–226. Springer LNCS 62, 1978.

[322] H. Ehrig, H.-J. Kreowski, and P. Padawitz. Algebraic implementation of abstract data types: an announcement. *SIGACT News*, 11(2):25–29, 1979.

[323] H. Ehrig, H.-J. Kreowski, J. Thatcher, E. Wagner, and J. Wright. Parameter passing in algebraic specification languages. *Theoretical Computer Science*, 28:45–81, 1984.

[324] H. Ehrig, H.-J. Kreowski, and H. Weber. Algebraic specification schemes for data base systems. In *Proc. 4th Conf. on Very Large Data Bases*, 1978.

[325] H. Ehrig, H.-J. Kreowski, and H. Weber. Neue Aspekte algebraischer Spezifikationsschemata für Datenbanksysteme. In *Proc. Workshop Formale Modelle für Informationssysteme*, pages 181–198. Springer IFB 21, 1979.

[326] H. Ehrig, M. Löwe, and F. Orejas. Dynamic Abstract Data Types Based on Algebraic Graph Transformations. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 236–254. Springer LNCS 906, 1995. ⊕.

[327] H. Ehrig and B. Mahr. Complexity of algebraic implementations for abstract data types. *Journal of Computer and System Sciences*, 23(2):223–253, 1981.

[328] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer, 1985.

[329] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints.* EATCS Monographs on Theoretical Computer Science, Vol. 21. Springer, 1990.

[330] H. Ehrig and F. Orejas. Dynamic Abstract Data Types: An Informal Proposal. *Bulletin of the EATCS*, 53:162–169, 1994. ⊕.

[331] H. Ehrig and F. Orejas, editors. *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, Spain, 1992, Selected Papers*. Springer LNCS 785, 1994. ⊕.

[332] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 2:217–256, 1995. ⊕.

[333] H. Ehrig and F. Parisi-Presicce. Interaction between Algebraic Specification Grammars and Modular System Design. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, 1993*, Workshops in Computing, pages 219–226. Springer-Verlag, London, 1993. ⊕.

[334] H. Ehrig, J. W. Thatcher, P. Lucas, and S. N. Zilles. Denotational and initial algebra semantics of the algebraic specification language LOOK. Technical Report 84-22, Technische Universität Berlin, 1984.

[335] H. Ehrig and H. Weber. Algebraic specification of modules. In *Proc. IFIP Working Conf. on Formal Models in Programming*, pages 231–258. North-Holland, 1985.

[336] H. Ehrig and H. Weber. Programming in the large with algebraic module specifications. In *Proc. IFIP Congress 1986*, pages 675–684. North-Holland, 1986.

[337] M. Elvang-Gøransson and O. Owe. A simple sequent calculus for partial functions. *Theoretical Computer Science*, 114(2):317–330, 1993. ⊕.

[338] J. Exner. The OPAL Tutorial. Technical Report 94-9, TU Berlin, May 1994. ⊕.

[339] L. M. Feijs and H. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge tracts in theoretical computer science*. Cambridge Univ. Press, 1992.

[340] L. M. G. Feijs. The calculus $\lambda\pi$. In M. Wirsing and J. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 307–328. Springer LNCS 394, 1989.

[341] L. M. G. Feijs, H. B. M. Jonkers, C. P. J. Koymans, and G. R. R. de Lavalette. Formal definition of the design language COLD-K. METEOR Report t7/PRLE/7, Philips Research Lab., Eindhoven, 1987.

[342] L. M. G. Feijs, H. B. M. Jonkers, J. H. Obbink, C. P. J. Koymans, G. R. R. de Lavalette, and P. M. Rodenburg. A survey of the design language COLD. In *Proc. ESPRIT Conf. '86: Results and Achievements*, pages 631–644. North-Holland, 1987.

[343] A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. Algebraic Programming in OPAL. *Bulletin of the EATCS*, pages 171–181, June 1993. ⊕.

[344] W. Fey. Syntax, Semantik und Korrektheit eines algebraischen Spezifikationsschemas für ein Stücklisten-Datenbanksystem. Technical Report 80-12, Technische Universität Berlin, 1980.

[345] W. Fey. Introduction to algebraic specification in ACT TWO. Technical Report 86-13, Technische Universität Berlin, 1986.

[346] W. Fey. *Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language*. PhD thesis, Technische Universität Berlin, 1988. Also: Technical Report 88-26.

[347] W. Fey. ACT TWO: An Algebraic Module Specification and Interconnection Language. In *Proc. 2nd Int. Conf. on Algebraic Methodology and Software Technology (AMAST)*, pages 131–134, 1991. ⊕.

[348] J. Fiadeiro and A. Sernadas. Structuring theories on consequence. In *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*. Springer LNCS 332, 1988.

[349] J. Fiadeiro, C. Sernadas, T. Maibaum, and A. Sernadas. Describing and Structuring Objects for Conceptual Schema Development. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development*, pages 117–138. John Wiley, 1992. ⊕.

[350] J. F. Fiadeiro, J. L. Costa, A. Sernadas, and T. S. E. Maibaum. Process semantics of temporal logic specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 236–253. Springer LNCS 655, 1993. ⊕.

[351] M. Franova. Fundamentals for a new methodology for inductive theorem proving: CM-construction of atomic formulae. In *Proc. 8th European Conf. on Artificial Intelligence*, pages 137–141. Pitman, 1988.

[352] M. Franova. *Fundamentals of a New Methodology for Program Synthesis from Formal Specifications: CM-construction of Atomic Formulae.* PhD thesis, Université de Paris-Sud, Orsay, 1988.

[353] M. Franova. An implementation of program synthesis from formal specifications. In *Proc. 8th European Conf. on Artificial Intelligence*, pages 559–564. Pitman, 1988.

[354] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. 1985 IEEE Symp. on Logic Programming*, pages 172–184, 1985.

[355] L. Fribourg. A strong restriction of the inductive completion procedure. *Journal of Symbolic Computation*, 8(3):253–276, Sept. 1989.

[356] M. Fuchs. *Technologieabhängigkeit von Spezifikationen digitaler Hardware.* PhD thesis, Technische Universität München, 1994. Technical Report TUM-I9431. ⊕.

[357] M. Fuchs and K. Stølen. Development of a Distributed Min/Max Component. In W. Grass and M. Mutz, editors, *Proc. 3. GI/ITG Workshop, Anwendung formaler Methoden beim Entwurf von Hardwaresystemen*, pages 126–135. Verlag Shaker, 1995. ⊕.

[358] K. Futatsugi, J. Goguen, J. Meseguer, and K. Okada. Parameterized programming in OBJ2. In *Proc. 9th IEEE Intl. Conf. on Software Engineering*, pages 50–61, 1987.

[359] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 52–66, 1985.

[360] P. L. Gall. *Les algèbres étiquetés : une sémantique pour les spécifications algébriques fondée sur une utilisation systématique des termes. Application au test de logiciel avec traitement d'exceptions.* Nouvelle thèse, Université de Paris Sud, Orsay, 1993.

[361] H. Ganzinger. Programs as Transformations of Algebraic Theories (Extended Abstract). *Informatik Fachberichte*, 50:22–41, 1981.

[362] H. Ganzinger. Parameterized specifications: parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems*, 5(3):318–354, 1983.

[363] H. Ganzinger. Order–sorted completion the many–sorted way. *Theoretical Computer Science*, 89:3–32, 1991. ⊕.

[364] S. J. Garland and J. V. Guttag. An overview of LP, the Larch Prover. In *Proc. 3rd Conf. on Rewriting Techniques and Applications*, pages 137–151. Springer LNCS 355, 1989.

[365] S. Gastinger, R. Hennicker, and R. Stabl. Design of Modular Software Systems with Reuse. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, pages 1–24. Springer LNCS 1009, 1995. ⊕.

[366] M.-C. Gaudel. A first introduction to PLUSS. Technical Report, LRI, Université de Paris-Sud, Orsay, 1984.

[367] M.-C. Gaudel. Towards structured algebraic specifications. In *Esprit '85 Status Report*, pages 493–510. North-Holland, 1985.

[368] M.-C. Gaudel. Structuring and Modularizing Algebraic Specifications: the PLUSS specification language, evolutions and perspectives. In A. Finkel and M. Jantzen, editors, *STACS 92. 9th Annual Symposium on Theoretical Aspects of Computer Science*, pages 13–18. Springer LNCS 577, 1992. $\oplus$.

[369] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 82–96. Springer LNCS 915, 1995. $\oplus$.

[370] M.-C. Gaudel and B. Marre. Algebraic specifications and software testing: theory and application. Report 407 LRI, Université de Paris-Sud, Orsay, 1988.

[371] M.-C. Gaudel and T. Moineau. A theory of software reusability. In *Proc. 1988 European Symp. on Programming*, pages 115–130. Springer LNCS 300, 1988.

[372] A. Geser and H. Hußmann. Rapid prototyping for algebraic specifications — examples for the use of the RAP system. Report MIP-8517, Universität Passau, 1985.

[373] N. Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, 1995. $\oplus$.

[374] N. Ghani. $\beta\eta$-Equality for Coproducts. In M. Dezani and G. Plotkin, editors, *Proceedings of TLCA'95*. Springer LNCS 902, 1995. $\oplus$.

[375] N. Ghani. Eta-Expansions in $F\omega$. Draft report, Ecole Normale Superieure. $\oplus$, 1996.

[376] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proc. 1976 Symp. on Mathematical Foundations of Computer Science*, pages 567–578. Springer LNCS 45, 1976.

[377] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational Completion in Order–sorted Algebras. *Theoretical Computer Science*, 72:169–202, 1990. $\oplus$.

[378] M. Gogolla. Partially ordered sorts in algebraic specifications. In *Proc. 1984 Colloq. on Trees in Algebra and Programming*, pages 139–153. Cambridge Univ. Press, 1984.

[379] M. Gogolla. A final algebra semantics for errors and exceptions. In *Recent Trends in Data Type Specification, Proc. 3rd Workshop on Theory and Applications of Abstract Data Types*, pages 89–103. Springer IFB 116, 1985.

[380] M. Gogolla. *Über Partiell Geordnete Sortenmengen und deren Anwendung zur Fehlerbehandlung in Abstrakten Datentypen*. PhD thesis, Technische Universität Braunschweig, 1986.

[381] M. Gogolla. On parametric algebraic specifications with clean error handling. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 81–95. Springer LNCS 249, 1987.

[382] M. Gogolla and I. Claßen. An Object-Oriented Design for the ACT ONE Environment. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, 1993*, Workshops in Computing, pages 219–226. Springer-Verlag, London, 1993. $\oplus$.

[383] M. Gogolla, S. Conrad, G. Denker, R. Herzig, and N. Vlachantonis. A Development Environment for an Object Specification Language. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):505–508, June 1995. $\oplus$.

[384] M. Gogolla, S. Conrad, G. Denker, R. Herzig, N. Vlachantonis, and H.-D. Ehrich. TROLL *light* — The Language and its Development Environment. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, pages 205–220. Springer LNCS 1009, 1995. $\oplus$.

[385] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. DISCO '93, International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 2–16. Springer LNCS 722, 1993. $\oplus$.

[386] M. Gogolla, K. Drosten, U. W. Lipeck, and H.-D. Ehrich. Algebraic and operational semantics of specifications allowing exceptions and errors. *Theoretical Computer Science*, 34:289–313, 1984.

[387] M. Gogolla and H.-D. Ehrich. Algebraic specifications with subsorts using declarations. *Bulletin of the EATCS*, 21:31–38, 1983.

[388] M. Gogolla and R. Herzig. An Algebraic Development Technique for Information Systems. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '95, Proceedings)*, pages 446–460. Springer LNCS 936, 1995. ⊕.

[389] M. Gogolla and R. Herzig. An Algebraic Semantics for the Object Specification Language TROLL *light*. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 290–306. Springer LNCS 906, 1995. ⊕.

[390] M. Gogolla, R. Herzig, S. Conrad, G. Denker, and N. Vlachantonis. Integrating the ER Approach in an OO Environment. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Proc. 12th Int. Conf. on the Entity-Relationship Approach (ER'93)*, pages 376–289. Springer LNCS 823, 1994. ⊕.

[391] M. Gogolla and U. Hohenstein. Towards a semantic view of an extended entity-relationship model. *ACM Transactions on Database Systems*, 1(1):9–36, 1991. ⊕.

[392] J. Goguen. Memories of ADJ. *Bulletin of the European Association for Theoretical Computer Science*, 36:96–102, October 1989. Guest column in the 'Algebraic Specification Column.' Also in *Current Trends in Theoretical Computer Science: Essays and Tutorials*, World Scientific, 1993, pages 76–81.

[393] J. Goguen. Principles of Parameterized Programming. In T. Biggerstaff and A. Perlis, editors, *Software Reusability, Volume I: Concepts and Models*. Addison Wesley, 1989. 159–222.

[394] J. Goguen. Types as Theories. In G. M. Reed, A. W. Roscoe, and R. F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[395] J. Goguen. Sheaf Semantics for Concurrent Interacting Objects. *Mathematical Structures in Computer Science*, 11:159–191, 1992.

[396] J. Goguen. Requirements Engineering as the reconciliation of social and technical issues. In J. Goguen and M. Jirotka, editors, *Requirements Engineering: Social and Tecnical Issues*, pages 165–199. Academic Press, 1994. ⊕.

[397] J. Goguen and R. Burstall. A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charter and Parchments. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydehard, editors, *Proceedings of Summer Workshop on Category Theory and Computer Programming*, pages 313–333. Springer LNCS 240, 1986.

[398] J. Goguen and R. Diaconescu. A Short Oxford Survey of Order Sorted Algebra. *Bulletin of the EATCS*, 48:121–133, 1992.

[399] J. Goguen and R. Diaconescu. An Introduction to Category-based Equational Logic. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '95, Proceedings)*, pages 91–126. Springer LNCS 936, 1995. ⊕.

[400] J. Goguen and Luqi. Formal Methods and Social Context in Software Development. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 62–81. Springer LNCS 915, 1995. ⊕.

[401] J. Goguen and G. Malcolm. Proof of Correctness of Object Representations. In A. W. Roscoe, editor, *A Classical Mind: Essays Dedicated to C.A.R. Hoare*, pages 119–142. Prentice-Hall, 1994. ⊕.

[402] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996. ⊕.

[403] J. A. Goguen. Abstract errors for abstract data types. In *Proc. IFIP Working Conference on the Formal Description of Programming Concepts.* North-Holland, 1978.

[404] J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, 1984.

[405] J. A. Goguen and R. M. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, Computer Science Laboratory, SRI International, 1980.

[406] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop*, pages 221–256. Springer LNCS 164, 1984.

[407] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[408] J. A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, Spain, 1992, Selected Papers*, pages 1–29. Springer LNCS 785, 1994. ⊕.

[409] J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebra. In *Proc. 12th Intl. Colloq. on Automata, Languages and Programming*, pages 221–231. Springer LNCS 194, 1985.

[410] J. A. Goguen and G. Malcolm. More Higher Order OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. Cambridge University Press, 1996. To appear. ⊕.

[411] J. A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *ACM SIGPLAN Notices*, 17(1):9–17, 1982.

[412] J. A. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, pages 265–281. Springer LNCS 140, 1982.

[413] J. A. Goguen and J. Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problem. In *Proc. 2nd IEEE Symp. on Logic in Computer Science*, 1987.

[414] J. A. Goguen and J. Meseguer. Unifying functional, object oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[415] J. A. Goguen and J. Meseguer. Order-sorted algebra I: partial and overloaded operations, errors and inheritance. Technical report, SRI International, Computer Science Lab, 1990. ⊕.

[416] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[417] J. A. Goguen and A. Socorro. Module Composition and System Design for the Object Paradigm. *Journal of Object Oriented Programming*, 7(14), 1995. ⊕.

[418] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Report RC-6487, IBM T. J. Watson Research Center, Yorktown Heights, 1976.

[419] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, pages 80–149. Prentice-Hall, 1978.

[420] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Abstract data types as initial algebras and the correctness of data representation. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Representations*, 1975.

[421] J. A. Goguen and T. Winkler. Introducing OBJ3. Research report SRI-CSL-88-9, SRI International, 1988.

[422] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. Cambridge University Press, 1996. To appear. ⊕.

[423] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[424] F. Golshani, T. Maibaum, and M. Sadler. A modal systems of algebras for database specification and query/update language support. In *Proc. 9th Conf. on Very Large Data Bases*, 1983.

[425] M. J. C. Gordon, A. J. R. G. Milner, and C. P. Wadsworth. *Edinburgh LCF*. Springer LNCS 78, 1979.

[426] S. Goto. Program synthesis from natural deduction proofs. In *Proc. 6th Intl. Joint Conf. on Artificial Intelligence*, pages 339–341, 1979.

[427] B. Gramlich. UNICOM: a refined completion based inductive theoremprover. In M. Stickel, editor, *10th International Conference on Automatic Deduction, Kaiserslautern (Germany)*, pages 655–656. Springer LNAI 449, 1990.

[428] M. Große-Rhode. Towards object–oriented algebraic specifications. In *Recent Trends in Data Type Specification. 7th Workshop on Specification of Abstract Data Types*, pages 98–116. Springer LNCS 534, 1991.

[429] M. Große-Rhode. *Specification of Transition Categories — An Approach to Dynamic Abstract Data Types —*. PhD thesis, TU Berlin, 1995. ⊕.

[430] M. Große-Rhode. Transition Specifications for Dynamic Abstract Data Types. *Bulletin of the EATCS*, 55:121–127, 1995. ⊕.

[431] M. Große-Rhode. Transition Specifications for Dynamic Abstract Data Types. accepted for publication in Applied Categorical Structures, ⊕, 1996.

[432] M. Große-Rhode and U. Wolter. 2-Categorical Specification of Partial Algebras. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, Spain, 1992, Selected Papers*, pages 207–219. Springer LNCS 785, 1994. ⊕.

[433] R. Grosu. *A Formal Foundation for Concurrent Object Oriented Programming*. PhD thesis, Technische Universität München, 1994. Technical Report TUM-I9444. ⊕.

[434] R. Grosu and D. Nazareth. The Specification Language SPECTRUM - Core Language Report V1.0. TUM-I 9429, Technische Universität München, 1994. ⊕.

[435] R. Grosu and D. Nazareth. Towards a New Way of Parameterization. In *Proceedings of the Third Maghrebian Conference on Software Engineering and Artificial Intelligence*, pages 383–392, 1994. ⊕.

[436] R. Grosu and F. Regensburger. The Logical Framework of SPECTRUM. TUM-I 9402, Institut für Informatik, Technische-Universität München, 1994.

[437] R. Grosu and F. Regensburger. The Semantics of SPECTRUM. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting. First International Workshop, HOA'93. Amsterdam*, pages 124–145. Springer LNCS 816, 1994. ⊕.

[438] R. Grosu and B. Rumpe. Concurrent Timed Port Automata. TUM-I 9533, Technische Universität München, 1995. ⊕.

[439] R. Grosu and K. Stølen. A Denotational Model for Mobile Point-to-Point Dataflow Networks. Technical Report SFB 342/14/95 A, Technische Universität München, 1996. An extended abstract also occurs in Proc. AMAST'96. ⊕.

[440] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[441] Y. Gurevich. Evolving algebras, a tutorial introduction. *Bulletin of the EATCS*, 43:264–284, 1991.

[442] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1993. ⊕.

[443] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[444] J. V. Guttag and J. J. Horning. A Larch Shared Language Handbook. *Science of Computer Programming*, 6(2):135–158, 1986.

[445] J. V. Guttag and J. J. Horning. Report on the Larch shared language. *Science of Computer Programming*, 6(2):103–134, 1986.

[446] J. V. Guttag and J. J. Horning. *Larch: Languages and tools for formal specification*. Texts and monographs in computer science. Springer, 1993.

[447] J. V. Guttag, J. J. Horning, and J. M. Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming*, 2:53–68, 1982.

[448] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. Research Report ISI/RR-76-48, University of Southern California, 1976.

[449] A. Habel, H.-J. Kreowski, and D. Plump. Jungle Evaluation. *Fundamenta Informaticae*, 15(1):37–60, 1991. ⊕.

[450] A. Habel and D. Plump. Unification, Rewriting, and Narrowing on Term Graphs. In *Proc. Joint* COMPUGRAPH/SEMAGRAPH *Workshop on Graph Rewriting and Computation (SEGRAGRA '95)*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. ⊕.

[451] A. Habel and D. Plump. Term graph narrowing. *Mathematical Structures in Computer Science*, 1996. To appear. ⊕.

[452] T. Hagino. A Categorical Programming Language. Technical Report ECS-LFCS-87-38, University of Edinburgh, Department od Computer Science, Sept. 1987.

[453] L. Hamel. *Behavioural Verification and Implementation of an Optimizing Compiler for OBJ3*. PhD thesis, Oxford University Computing Laboratory, 1996. ⊕.

[454] L. H. Hamel and J. Goguen. Towards a Provably Correct Compiler for OBJ3. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, pages 132–146. Springer LNCS 844, 1994. ⊕.

[455] B. S. Hansen and J. U. Toft. The Formal Specification of ANDF, an Application of Action Semantics. In P. D. Mosses, editor, *Proc. 1st Intl. Workshop on Action Semantics, Edinburgh, 1994*, number NS-94-1 in Notes Series, pages 34–42. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994. ⊕.

[456] H. Hansen. *Von Algebraischen Spezifikationen zu Algebraischen Programmen*. PhD thesis, Technische Universität Berlin, 1983.

[457] T. Hardin. Eta-conversion for the languages of explicit substitutions. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*. Springer LNCS 632, 1992.

[458] D. Hare. Dynamic Logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic: Vol. 2: Extensions of Classical Logic*, pages 497–604. Kluwer, Dordrecht, 1984.

[459] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[460] R. Harper and J. Mitchell. On the Type Structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[461] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994. ⊕.

[462] R. W. Harper, D. B. MacQueen, and R. G. Milner. Standard ML. Report ECS-LFCS-86-2, University of Edinburgh, 1986.

[463] P. H. Hartel et al. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 1996. to appear.

[464] M. Haveraaen. Distributing Programs on Different Parallel Architectures. In D. A. Padua, editor, *The 1990 International Conference on Parallel Processing, ICPP'90. Chicago, USA, August 1990*, volume II – Software, pages 288–289. Penn State Press, 1990.

[465] M. Haveraaen. Comparing Some Approaches to Programming Distributed Memory Machines. In Q. Stout and M. Wolfe, editors, *The 6th Distributed Memory Computing Conference, DMCC6'91. Portland, Oregon, USA, April 1991*, pages 222–227. IEEE Computer Society Press, 1991.

[466] M. Haveraaen. How to Create Parallel Programs without Knowing it. In S. Meldal and M. Haveraaen, editors, *The 4th Nordic Workshop on Program Correctness, NWPC4. Bergen, Norway, November 1992*, number 78 in Reports in Informatics, pages 165–176, N-5020 BERGEN, Norway, April 1993. ⊕.

[467] M. Haveraaen, V. Madsen, and H. Munthe-Kaas. Algebraic Programming Technology for Partial Differential Equations. In A. Maus et al., editors, *Norsk Informatikk Konferanse, NIK'92. Tromsø, Norway, November 1992*, pages 55–68. Tapir, 1992.

[468] M. Haveraaen and N. A. Magnussen. Why Parallel Programming is Easy, and CSP Programming is Hard. In B. Kirkerud et al., editors, *Norsk Informatikk Konferanse, NIK'90. Bergen, Norway, November 1990*, pages 183–192. Tapir, 1990.

[469] M. Haveraaen, O. Owe, and O.-J. Dahl, editors. *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types, WADT11. Oslo Norway, September 1995.* Springer LNCS, 1996. In preparation. ⊕.

[470] M. Haveraaen and E. G. Wagner. Guarded algebras and data type specification. Technical report 108, Department of Informatics, University of Bergen, N-5020 BERGEN, Norway, October 1995. ⊕.

[471] A. E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST'96, Proceedings*, 1996. ⊕.

[472] R. Hennicker. Context Induction: a Proof Principle for Behavioural Abstractions and Algebraic Implementations. *Formal Aspects of Computing*, 3(3), 1991. ⊕.

[473] R. Hennicker. A semi-algorithm for algebraic implementation proofs. *Theoretical Computer Science*, 104(1):53–88, 1992. ⊕.

[474] R. Hennicker and F. Nickl. A behavioural algebraic framework for modular system design with reuse. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, 1992.* Springer LNCS 785, 1994. ⊕.

[475] R. Hennicker and M. Wirsing. Behavioural Specifications. In H. Schwichtenberg, editor, *Proof and Computation*, NATO ASI Series F: Computer and Systems Sciences, pages 193–230. Springer Verlag, 1995. ⊕.

[476] R. Hennicker, M. Wirsing, and M. Bidoit. Proof systems for structured specifications with observability operators. *Theoretical Computer Science*, 1996. To appear. $\oplus$.

[477] U. Hensel and H. Reichel. Defining Equations in Terminal Coalgebras. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 307–318. Springer LNCS 906, 1995. $\oplus$.

[478] R. Herzig, S. Conrad, and M. Gogolla. Compositional Description of Object Communities with TROLL *light*. In C. Chrisment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94)*, pages 183–194. Cépaduès-Éditions, Toulouse, 1994. $\oplus$.

[479] R. Herzig and M. Gogolla. A SQL-like Query Calculus for Object-Oriented Database Systems. In E. Bertino and S. Urban, editors, *Proc. Int. Symp. on Object-Oriented Methodologies and Systems (ISOOMS'94)*, pages 20–39. Springer LNCS 858, 1994. $\oplus$.

[480] R. Herzig and M. Gogolla. An Animator for the Object Specification Language TROLL *light*. In V. Alagar and R. Missaoui, editors, *Proc. Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, pages 4–17. Université du Quebéc à Montréal, 1994. $\oplus$.

[481] W. H. Hesselink. A mathematical approach to nondeterminism in data types. *ACM Transactions on Programming Languages and Systems*, 10(1):87–117, 1988.

[482] R. Hettler. *Entity/Relationship-Datenmodellierung in axiomatischen Spezifikationssprachen*. PhD thesis, Technische Universität München, 1995.

[483] H. Hilberdink. The End of Syntax. The End of Algebra. Manuscripts, St. Peter's College, Oxford, 1995.

[484] C. Hintermeier. How to Transform Canonical Decreasing HCTRs into Equivalent Canonical TRSs. In N. Dershowitz and N. Lindenstrauss, editors, *Conditional and Typed Rewriting Systems. 4th International Workshop, CTRS–94. Jerusalem, Israel, July 1994*, pages 186–205. Springer LNCS 968, 1995.

[485] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-Typed Computations for Order-Sorted Equational Presentations –Extended Abstract–. In S. Abiteboul and E. Shamir, editors, *Proc. 21st International Colloquium on Automata, Languages, and Programming*, pages 450–461. Springer LNCS 820, 1994. $\oplus$.

[486] C. Hintermeier, C. Kirchner, and H. Kirchner. Sort Inheritance for Order–Sorted Equational Presentations. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 319–335. Springer LNCS 906, 1995. $\oplus$.

[487] C. Hintermeier, H. Kirchner, and P. D. Mosses. Combining Algebraic and Set-Theoretic Specification. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 255–273. Springer–Verlag, 1996. To appear. $\oplus$.

[488] C. Hintermeier, H. Kirchner, and P. D. Mosses. R^n- and G^n-logics. In G. Dowek, J. Heering, K. Meinke, and B. Möller, editors, *Higher-Order Algebra, Logic and Term Rewriting. Second International Workshop, HOA '95. Paderborn, Germany, September 1995*, pages 90–108. Springer LNCS 1074, 1996. $\oplus$.

[489] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[490] D. Hofbauer and M. Huber. Linearizing Term Rewriting Systems Using Test Set. *Journal of Symbolic Computation*, 17(1):91–129, 1994. $\oplus$.

[491] B. Hoffmann and B. Krieg-Brückner, editors. *Program Development by Specification and Transformation. The PROSPECTRA Methodology, Language Family, and System*. Springer LNCS 680, 1993. $\oplus$.

[492] B. Hoffmann and D. Plump. Jungle Evaluation for Efficient Term Rewriting. In *Proc. Intl. Workshop on Algebraic and Logic Programming* Mathematical Research 49, pages 191–203, Berlin, 1988. Akademie–Verlag. Also in Springer Lecture Notes in Computer Science 343, 191–203, 1989.

[493] B. Hoffmann and D. Plump. Implementing term rewriting by jungle evaluation. *RAIRO Theoretical Informatics and Applications*, 25(5):445–472, 1991. ⊕.

[494] M. Hofmann. Sound and complete axiomatisations of call by value control operators. In *Proc. 5th Conf. on Category Theory and Computer Science*. Springer LNCS, 1993. ⊕.

[495] M. Hofmann. Elimination of extensionality for Martin-Löf type theory. In *Types for Proofs and Programs*. Springer LNCS 806, 1994. ⊕.

[496] M. Hofmann. Conservativity of equality reflection over intensional type theory. In S. Berardi and M. Coppo, editors, *Proceedings of BRA Types workshop*, 1995. to appear, ⊕.

[497] M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995. ⊕.

[498] M. Hofmann. On the Interpretation of Type Theory in Locally Cartesian Closed Categories. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic. 8th Workshop, CSL'94. Kazimierz, Poland, September 1994*, pages 427–441. Springer LNCS 933, 1995. ⊕.

[499] M. Hofmann. A simple model for quotient types. In M. Dezani and G. Plotkin", editors, *Proceedings of TLCA'95.* "Springer LNCS 902, 1995. ⊕.

[500] M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science*, 1996. To appear, ⊕.

[501] M. Hofmann and D. T. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 247–261. Springer LNCS 915, 1995. ⊕.

[502] M. Hofmann and D. T. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 247–261. Springer LNCS 915, 1995. ⊕.

[503] M. Hofmann and T. Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proc. 9th IEEE Symp. on Logic in Computer Science*, 1994. ⊕.

[504] E. Holz and K. Stølen. An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus. In D. Hogrefe and S. Leue, editors, *Proc. Forte'94*, pages 324–339, 1994.

[505] G. Hommel and D. Kroenig, editors. *Requirements Engineering*. Springer IFB 74, 1983.

[506] I. V. Horebeek and J. Lewi. *Algebraic Specifications in Software Engineering. An Introduction*. Springer, 1989.

[507] N. Hsiao. *Sélection de tests de propriétés de sûreté à partir d'une modélisation algébrique de programmes LUSTRE*. Nouvelle thèse, Université de Paris Sud, Orsay, 1994.

[508] G. Huet. The Gilbreath trick: A case study in axiomatisation and proof development in the COQ proof assistant. Rapports de Recherche 1511, INRIA, 1991.

[509] G. Huet and J. M. Hullot. Proofs by Induction in Equational Theories with Constructors. *Journal of Computer and System Sciences*, 25(2), 1982.

[510] M. Huhn, H. Wehrheim, and G. Denker. Action Refinement - An Application of Process Theory on Object-Oriented Specification. Technischer Report 40/95, Universität Hildesheim, Institut für Informatik, Postfach 101363, D-31113 Hildesheim, Nov. 1995. ⊕.

[511] J. M. Hullot. Canonical forms and unification. In *Proc. 5th Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[512] J. M. Hullot. *Compilation de formes canoniques dans de théories équationelles.* PhD thesis, University de Paris–Sud, 1980.

[513] H. Hußmann. Formal foundations for SSADM. Habilitation Thesis, Technische Universität München, 1994. ⊕.

[514] S. Jähnichen, F. A. Hussain, and M. Weber. Program development using a design calculus. In *Proc. ESPRIT Conf. '86: Results and Achievements*, pages 645–658. North-Holland, 1987.

[515] C. Jay and N. Ghani. The Virtues of η-Expansion. *Journal of Functional Programming*, 5:135–154, 1995. ⊕.

[516] R. Jiménez, F. Orejas, and H. Ehrig. Compositionality and Compatibility of Parameterization and Parameter Passing in Specification Languages. *Mathematical Structures in Computer Science*, 5(2):283–313, 1995. ⊕.

[517] T. Johnsson. Lambda-lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Proceedings of 2nd Conference on Functional Programming Languages and Computer Architecture, Nancy(France)*, pages 190–203. Springer LNCS 201, 1985.

[518] H. B. M. Jonkers. An introduction to COLD-K. In *Algebraic Methods: Theory, Tools and Applications*, pages 139–206. Springer LNCS 394, 1989.

[519] J.-P. Jouannaud, C. Kichner, H. Kirchner, and A. Megrelis. Programming with Equalities, Subsorts, Overloading and Parameterization in OBJ. *Journal of Logic Programming*, 12:257–279, 1992.

[520] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989. ⊕.

[521] J.-P. Jouannaud and M. Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *Automata, Languages and Programming, 18th International Colloquium*, Madrid, Spain, July 1991. Springer LNCS 510. ⊕.

[522] R. Junclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 1995. to appear.

[523] S. Kahrs. *λ-rewriting*. PhD thesis, Universität Bremen, 1991. ⊕.

[524] S. Kahrs. Context rewriting. In M. Rusinowitch and J.-L. Rémy, editors, *Conditional Term Rewriting Systems. CTRS-92 Pont-à-Mouson, France, July 1992*, pages 21–35. Springer LNCS 656, 1992. ⊕.

[525] S. Kahrs. Compilation of combinatory reduction systems. In *Proc. Conf. on Higher-Order Algebra, Logic and Term Rewriting*, pages 169–188. Springer LNCS 816, 1993. ⊕.

[526] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Internal Report ECS-LFCS-93-257, University of Edinburgh, 1993. An update listing further errors is available at `ftp://ftp.dcs.ed.ac.uk/pub/smk/SML/errors-new.ps.Z`, ⊕.

[527] S. Kahrs. First-class polymorphism for ml. In *Proc. 5th European Symp. on Programming*, pages 333–347. Springer LNCS 788, 1994. ⊕.

[528] S. Kahrs. Confluence of curried term-rewriting systems. *Journal of Symbolic Computation*, 19:601–623, 1995. ⊕.

[529] S. Kahrs. Towards a domain theory for termination proofs. In *Proc. Conf. on Rewriting Techniques and Applications*, pages 241–255. Springer LNCS, 1995. ⊕.

[530] S. Kahrs. Proof rules for equality and comparison in Extended ML. Draft report, University of Edinburgh, ⊕, 1996.

[531] S. Kahrs. The Variable Containment Problem. In G. Dowek, J. Heering, K. Meinke, and B. Möller, editors, *Higher-Order Algebra, Logic and Term Rewriting. Second International Workshop, HOA '95. Paderborn, Germany, September 1995*, pages 109–123. Springer LNCS 1074, 1996. ⊕.

[532] S. Kahrs, D. Sannella, and A. Tarlecki. The semantics of Extended ML: a gentle introduction. In *Proc. Intl. Workshop on Semantics of Specification Languages*, pages 186–215. Springer Workshops in Computing, 1994. ⊕.

[533] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Internal Report ECS-LFCS-94-300, University of Edinburgh, 1994. ⊕.

[534] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 1996. to appear, ⊕.

[535] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. In *PLILP'95*. Springer LNCS, 1995.

[536] H. Kaphengst. What is computable for abstract data types? In *Proc. 1981 Colloq. on Foundations of Computation Theory*, pages 173–181. Springer LNCS 117, 1981.

[537] S. Kaplan. Rewriting with a nondeterministic choice operator: from algebra to proofs. In *Proc. 1986 European Symp. on Programming*, pages 351–374. Springer LNCS 213, 1986.

[538] S. Kaplan. A compiler for conditional term rewriting systems. In *Proc. 2nd Conf. on Rewriting Techniques and Applications*, pages 25–41. Springer LNCS 256, 1987.

[539] S. Kaplan and C. Choppy. Abstract rewriting with concrete operators. In N. Dershowitz, editor, *Proc. of the Third International Conference on Rewriting Techniques and Applications*, pages 178–186. Springer-Verlag LNCS 355, 1989. ⊕.

[540] D. Kapur and D. R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, 1987.

[541] D. Kapur, P. Narendran, D. J. Rosenkrantz, and H. Zhang. Sufficient Completeness, Ground–Reducibility and their Complexity. *Acta Informatica*, 28:311–350, 1991.

[542] D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–415, 1987.

[543] D. Kapur, P. Narendran, and H. Zhang. Automating Inductionless Induction using Test Sets. *Journal of Symbolic Computation*, pages 83–111, 1991.

[544] D. Kapur and M. Srivas. Computability and implementability issues in abstract data types. *Science of Computer Programming*, 10:33–63, 1988.

[545] D. Katiyar, D. Luckham, N. Madhav, S. Meldal, J. Mitchell, and S. Sankar. Subtyping, Assignment and Cloning in a Concurrent Object-Oriented Language. In *The DARPA Software Technology Conference 1992*, pages 458–470. DARPA, Apr. 1992. ⊕.

[546] D. Katiyar, D. Luckham, S. Meldal, and J. Mitchell. Polymorphism and subtyping in interfaces. In J. M. Wing, editor, *The Workshop on Interface Definition Languages*. Carnegie-Mellon University, Aug. 1994. Also in Carnegie-Mellon University tech. report no. CMU-CS-94 and [547]. ⊕.

[547] D. Katiyar, D. Luckham, S. Meldal, and J. Mitchell. Polymorphism and subtyping in interfaces. *ACM SIGPLAN Notices*, 29(8):22–34, 1994. Also in Carnegie-Mellon University tech. report no. CMU-CS-94. ⊕.

[548] R. Kennaway, J. Klop, R. Sleep, and F. de Vries. Comparing curried and uncurried rewriting. In *Dirk van Dalen Festschrift*. Utrecht University, 1993.

[549] S. Khosla, T. Maibaum, and M. Sadler. Database specification. In *Proc. IFIP Working Conference on Database Semantics*. North-Holland, 1985.

[550] C. Kirchner. A New Equational Unification Method: A generalization of Martelli–Montanari's Algorithm. In R. E. Shostak, editor, *7th International Conference on Automated Deduction*. Springer LNCS 170, 1984.

[551] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ3. In *Proc. 15th Intl. Colloq. on Automata, Languages and Programming*, pages 287–301. Springer LNCS 317, 1988.

[552] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with Symbolic Constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990.

[553] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. V. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. MIT press, 1995. $\oplus$.

[554] C. Kirchner and P. Viry. Implementing Parallel Rewriting. In P. Deransart and J. Maluszynski, editors, *Proceedings of PLILP'90*, pages 1–15, Linköping (Sweden), Aug. 1990. Springer LNCS 456. $\oplus$.

[555] H. Kirchner. On the Use of Constraints in Automated Deduction. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, pages 128–146. Springer LNCS 910, 1995. $\oplus$.

[556] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Rewriting Techniques and Applications. 6th International Conference, RTA-95, Kaiserslautern (Germany)*, pages 438–443. Springer LNCS 914, 1995. $\oplus$.

[557] H. A. Klaeren. *Algebraische Spezifikation — eine Einführung*. Springer, 1983.

[558] M. Klar, R. Geisler, and F. Cornelius. *InterACT*: An Interactive Theorem and Completeness Prover for Algebraic Specifications with Conditional Equations. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 274–288. Springer–Verlag, 1996. To appear. $\oplus$.

[559] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[560] A. Knapp and L. Mandel. A Rewriting System with Explicit Substitutions for the First Order Constrained Lambda Calculus. Interne Reports der FG Programmiersysteme FR I-Passau-1994-094, FORWISS (Universität Passau), 1994.

[561] Y. Kodratoff and M. Picard. Complétion de systèmes de réécriture et synthèse de programmes à partir de leurs spécifications. *Bigre No. 35*, 1983.

[562] M. Korff. True concurrency semantics for single pushout graph transformations with applications to actor systems. In R. J. Wieringa and R. B. Feenstra, editors, *Working papers of the International Workshop on Information Systems - Correctness and Reusability*, pages 33–50. World Scientific, 1995.

[563] P. Kosiuczenco and M. Wirsing. Timed Rewriting Logic. In *Proceedings of the Third AMAST Workshop on Real-Time Systems, Salt Lake City, Utah, 1996*, 1996. $\oplus$.

[564] P. Kosiuczenco and M. Wirsing. Timed Rewriting Logic with an Application to Object-Based Specification. *Science of Computer Programming*, 1996. To appear. $\oplus$.

[565] E. Kounalis. Testing for inductive (co)-reducibility. In A. Arnold, editor, *Proceedings 15th CAAP, Copenhagen (Denmark)*, pages 221–238. Springer LNCS 431, May 1990.

[566] E. Kounalis. Testing for the ground (co-)reducibility property in term-rewriting systems. *Theoretical Computer Science*, 106:87–117, 1992.

[567] G. Kreisel. Some uses of proof theory for finding computer programs. In *Colloque International de Logique*. Clermont-Ferrand, 1975.

[568] G. Kreisel. Extraction of bounds: interpreting some tricks of the trade. In P. Suppes, editor, *University-Level Computer-Assisted Instruction at Stanford, 1968–1980*, pages 149–164. Stanford University, 1981.

[569] G. Kreisel. Neglected possibilities of processing assertions and proofs mechanically: choice of problems and data. In P. Suppes, editor, *University-Level Computer-Assisted Instruction at Stanford, 1968–1980*, pages 131–148. Stanford University, 1981.

[570] G. Kreisel. Proof theory and the synthesis of programs: potential and limitations. In *Proc. 1985 European Conf. on Computer Algebra*, pages 136–151. Springer LNCS 204, 1985.

[571] H.-J. Kreowski. Evaluation in a distributed specification. Talk at the 10th ADT workshop, June 1994, Santa Margherita Ligure, Italy. Unpublished note. German version published in [576]. ⊕.

[572] H.-J. Kreowski. Algebraische Spezifikation von Softwaresystemen. In *Proc. GACM-Konf. Software Engineering*, pages 46–74. Teubner-Verlag, 1981.

[573] H.-J. Kreowski, editor. *Recent Trends in Data Type Specification, Proc. 3rd Workshop on Theory and Applications of Abstract Data Types.* Springer IFB 116, 1985.

[574] H.-J. Kreowski. Partial algebras flow from algebraic specifications. In *Proc. 14th Intl. Colloq. on Automata, Languages and Programming*, pages 521–530. Springer LNCS 267, 1987.

[575] H.-J. Kreowski. Colimits as parameterized data types. In *Proceedings of the Workshop on Categorical Methods in Computer Science with Aspects from Topology*, pages 36–49. Springer LNCS 393, 1989.

[576] H.-J. Kreowski. Ein Vorschlag zum Testen strukturierter algebraischer Spezifikationen. In P. Liggesmeier, H. Sneed, and A. Spillner, editors, *Testen, Analysieren und Verifizieren von Software, Informatik aktuell*, pages 130–142. Springer-Verlag, 1992. ⊕.

[577] H.-J. Kreowski. Some Initial Sections of the Algebraic Specification Tale. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science. Essays and Tutorials*, pages 54–75. World Scientific Series in Computer Science — Vol. 40, 1993. ⊕.

[578] H.-J. Kreowski and T. Mossakowski. Equivalence and difference of institutions: Simulating horn clause logic with based algebras. *Mathematical Structures in Computer Science*, 5:189–215, 1995. ⊕.

[579] H.-J. Kreowski and Z. Qian. Relation-sorted specifications with built-in coercers: basic notions and results. In *Proc. 7th Symp. on Theoretical Aspects of Computer Science*, pages 165–175. Springer LNCS 415, 1990. ⊕.

[580] V. Kriaučiukas and M. Walicki. Reasoning and Rewriting with Set–Relations I: The Ground Case. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic. 8th Workshop, CSL'94. Kazimierz, Poland, September 1994*, pages 264–278. Springer LNCS 933, 1995. ⊕.

[581] B. Krieg-Brückner. Integration of program construction and verification: the PROSPECTRA methodology. In *Proc. CRAI Intl. Spring Conf. on Innovative Software Factories and Ada*, pages 173–194. Springer LNCS 275, 1987.

[582] B. Krieg-Brückner. The PROSPECTRA methodology of program development. In *Proc. IFIP/IFAC Working Conf. on Hardware and Software for Real Time Process Control*, 1988.

[583] B. Krieg-Brückner. Algebraic specification and functionals for transformational program and meta program development. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 36–59. Springer LNCS 352, 1989.

[584] B. Krieg-Brückner. COMPASS, a COMPrehensive Algebraic approach to System Specification and development, ESPRIT Basic Research Working Group 3264. *Bulletin of the EATCS*, 40:144–157, 1990. ⊕.

[585] B. Krieg-Brückner. PROgram development by SPECification and TRAnsformation. *Technique et Science Informatiques. Special Issue on Advanced Software Engineering in ESPRIT*, pages 134–149, 1990. Invited paper. ⊕.

[586] B. Krieg-Brückner. Seven years of COMPASS. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 1–13. Springer–Verlag, 1996. To appear. ⊕.

[587] B. Krieg-Brückner, B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möncke, B. Weisgerber, A. McGettrick, I. G. Campbell, and G. Winterstein. PROgram development by SPECification and TRAnsformation. In *Proc. ESPRIT Conf. '86: Results and Achievements*, pages 301–312. North-Holland, 1987.

[588] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. UniForM, Universal Formal Methods Workbench. In U. Grote, editor, *Statusseminar Softwaretechnologie. BMBF.* BMBF, 1996. To appear. Preliminary version obtainable through the WWW: `http://www.informatik.uni-bremen.de/~uniform`. *Extended version in German:* Informatik Bericht Nr. 8/95, Universität Bremen. ⊕.

[589] B. Krieg-Brückner and D. Plump (eds.). COMPASS, a COMPrehensive Algebraic approach to System Specification and development (ESPRIT Basic Research Working Group 3264). Informatik Bericht Nr. 7/91, Universität Bremen, 1991. ⊕.

[590] B. Krieg-Brückner and D. Sannella. Structuring Specifications in-the-large and in-the-small: Higher-Order Functions, Dependent Types and Inheritance in SPECTRAL. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91. Proccedings of the International Joint Conference on Theory and Practice of Software Development. Volume 1: Colloquium on Trees in Algebra and Programming*, pages 103–120. Springer LNCS 493, 1991. ⊕.

[591] B. Krieg-Brückner (ed.). COMPASS, a COMPrehensive Algebraic approach to System Specification and development (ESPRIT basic research working group 3264): Objectives, state of the art, references. Bericht Nr. 6/89, Universität Bremen, 1989.

[592] S. Krischer. The Backward Walk Approach in FSM verification. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and Applications (CHDL-93), Ottawa (Ont., Canada)*, volume A-32 of *IFIP transactions*, pages 155–162. Elsevier Science Publishers B.V. (North-Holland), 1993. ⊕.

[593] B. Kristoffersen. Termination of Curryfied Rewrite Systems. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 305–324. Springer–Verlag, 1996. To appear. ⊕.

[594] S. Krogdahl and O. Lysne. Verifying a Distributed List System: A Case History. Research Report 182, Dept. of Informatics, Univ. of Oslo, Dec. 1993. Submitted to Formal Aspects of Computing.

[595] G. Kucherov, P. Lescanne, and P. D. Mosses. Valentin Antimirov (1961–1995). *Theoretical Computer Science*, 155:289–290, 1996. Obituary. ⊕.

[596] G. Kucherov and M. Rusinowitch. The Complexity of Testing Ground Reducibility for Linear Word Rewriting Systems with Variables. In N. Dershowitz and N. Lindenstrauss, editors, *Conditional and Typed Rewriting Systems. 4th International Workshop, CTRS–94. Jerusalem, Israel, July 1994*, pages 262–275. Springer LNCS 968, 1995. ⊕.

[597] G. Kucherov and M. Rusinowitch. Undecidability of ground reducibility for word rewriting systems with variables. *Information Processing Letters*, 53(4):209–216, 1995. ⊕.

[598] W. Küchlin. Inductive Completion by Ground Proof Transformation. In H. Aït-Kaci and M. Nivat, editors, *Colloquium on the Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 211–244. Academic Press inc., 1989.

[599] B. Kutzler and F. Lichtenberger. *Bibliography on Abstract Data Types.* Springer IFB 68, 1983.

[600] A. Laut and H. Partsch. Tuning algebraic specifications by type merging. In *Proc. 5th Symp. on Programming*, pages 283–304. Springer LNCS 137, 1982.

[601] F. W. Lawvere. Functorial Semantics of Algebraic Theories. In *Proc. National Academy of Science, U.S.A., 50*, pages 869–872. Columbia University, 1963.

[602] U. Lechner, C. Lengauer, and M. Wirsing. An Object–Oriented Airport: Specification and Refinement in Maude. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 351–367. Springer LNCS 906, 1995. ⊕.

[603] F. Leclerc. Termination proof of term rewriting systems with the multiset path ordering. a complete development in the system COQ. In M. Dezani, editor, *International Conference on Typed Lambda Calculus and Applications*, pages 312–327. Springer LNCS 902, 1995. ⊕.

[604] T. Lehmann and J. Loeckx. The specification language of OBSCURE. In *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, pages 131–153. Springer LNCS 332, 1988.

[605] T. Lehmann and J. Loeckx. Obscure, A Specification Language for Abstract Data Types. *Acta Informatica*, 30(4):303–350, 1993.

[606] P. Lescanne. Modèles non déterministes de types abstraits. *RAIRO Informatique Théorique*, 16(3):225–244, 1982.

[607] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, 1983.

[608] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, pages 21–36. Springer LNCS 632, Sept. 1992. ⊕.

[609] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In H. Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994. ⊕.

[610] P. Lescanne. Termination of Rewrite Systems by Elementary Interpretations. *Formal Aspects of Computing*, 7(1):77–90, 1995. ⊕.

[611] P. Lescanne. The lambda calculus as an abstract data type. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 74–80. Springer–Verlag, 1996. To appear. ⊕.

[612] P. Lescanne and J. Rouyer-Degli. Explicit Substitutions with de Bruijn's Levels. In J. Hsiang, editor, *Rewriting Techniques and Applications. 6th International Conference, RTA-95, Kaiserslautern (Germany)*, pages 294–308. Springer LNCS 914, 1995. ⊕.

[613] H. Lieberman. Using protypal objects to implement shared behaviour in object oriented systems. In *Proc. OOPSLA '85*, 1985.

[614] H. Linnestad and O. Lysne. Consistency Proofs in Higher Order Rewrite Systems. In J. Hsiang, editor, *Rewriting Techniques and Applications. 6th International Conference, RTA-95, Kaiserslautern (Germany)*, pages 26–40. Springer LNCS 914, 1995. Also in Proceedings Journees du GDR Programmation, Lille (France), GDR Programmation du CNRS 1994. ⊕.

[615] H. Linnestad and O. Lysne. Consistency Proofs in Higher Order Rewrite Systems. Research report, Dept. of Informatics, Univ. of Oslo, 1996. Submitted to CADE'12 (part of FLOC'96). ⊕.

[616] J. Liu and Z. Qian. Modular higher-order normalizing narrowing. Technical report, Universität Bremen, 1994. ⊕.

[617] J. Liu and Z. Qian. Using first-order narrowing to solve goals of higher-order patterns. In M. Takeichi and T. Ida, editors, *Proc. Fuji Int. Workshop on Functional and Logic Programming*. World Scientific Publ., 1994. $\oplus$.

[618] H. Lock, A. Mück, and T. Streicher. A tiny constraint functional logic language and its continuation semantics. In D. Sannella, editor, *ESOP'94. LNCS 788.*, pages 439–454, 1994. $\oplus$.

[619] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996. to appear.

[620] LOTOS Group. Enhanced lotos documentation. `ftp://ftp.dit.upm.es/pub/lotos/elotos/`, 1996.

[621] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993. $\oplus$.

[622] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M. Sleep, M. Plasmeijer, and M. van Eekelen1, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993. $\oplus$.

[623] M. Löwe and U. Wolter. Parametric algebraic specifications with Gentzen formulas – from quasi-freeness to free functor semantics. *Mathematical Structures in Computer Science*, 5:69–111, 1995. $\oplus$.

[624] R. Lundevik. Oversettelse fra Standard ML til Safir. Master's thesis, Universitetet i Bergen, 1994. $\oplus$.

[625] G. Luo, G. Bochmann, and A. Petrenko. Test selection based on communicating finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.

[626] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[627] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3(3), 1993.

[628] Z. Luo. *Computation and Reasoning. A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science*. Oxford Science Publications, 1994.

[629] C. Lüth. *Compositional Term Rewriting in Structured Algebraic Specifications*. PhD thesis, University of Edinburgh, 1996. Forthcoming, $\oplus$.

[630] N. A. Lynch and E. K. Blum. Relative complexity of algebras. *Math. Syst. Theory*, 14:193–214, 1981.

[631] O. Lysne. *Term Rewriting Techniques for Systems based on Generator Induction*. PhD thesis, Dept. of Informatics, Univ. of Oslo, Norway, 1991. Research Report 163. $\oplus$.

[632] O. Lysne. Proof by Consistency in Constructive Systems with Final Algebra Semantics. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, pages 276–290. Springer LNCS 632, 1992.

[633] O. Lysne. The Equational Part of Proofs by Structural Induction. *BIT*, 33:596–618, 1993. $\oplus$.

[634] O. Lysne. Extending Bachmair's method for proof by consistency to the final algebra. *Information Processing Letters*, 51:303–310, 1994. $\oplus$.

[635] O. Lysne. *E*-Unification by Consistency. In *Proceedings 5th Nordic Workshop on Program Correctness*, volume 18 of *Reports on Computer Science & Mathematics*, pages 127–136. Åbo Akademi, 1994. $\oplus$.

[636] O. Lysne. On the Connection between Narrowing and Proof by Consistency. In A. Bundy, editor, *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, pages 133–147. Springer LNAI 814, June 1994. $\oplus$.

[637] N. Madhav and S. Melda. An object-oriented approach to formal specification of reactive system. In *The 11th International Conference on Distributed Computing System*, pages 134–14. IEEE Computer Society, 1991. ⊕.

[638] V. Madsen. Tensorfelt, fra spesifikasjon til implementasjon. Master's thesis, Universitetet i Bergen, 1993. ⊕.

[639] M. E. Majster. Data types, abstract data types and their specification problem. *Theoretical Computer Science*, pages 89–127, 1979.

[640] G. Malcolm. Behavioural Equivalence, Bisimulation and Minimal Realisation. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 342–361. Springer–Verlag, 1996. To appear. ⊕.

[641] G. Malcolm. Interconnection of Object Specifications. In S. Goldsack and S. Kent, editors, *Formal Methods in Object Technology*. Springer Workshops in Computing, 1996. ⊕.

[642] G. Malcolm and C. Cîrstea. Distributed Operational Sematics for the Object Paradigm. In preparation, Oxford University Computing Laboratory, 1996. ⊕, 1996.

[643] G. Malcolm and J. Goguen. Proving correctness of refinement and implementation. Technical Monograph PRG-114, Oxford University Computing Laboratory, 1984.

[644] G. Malcolm and J. Goguen. A hidden agenda. In preparation. ⊕, 1996.

[645] V. Manca, A. Salibra, and G. Scollo. On the nature of TELLUS: a Typed Equational Logic Look over Uniform Specification. In *Proc. 1989 Symp. on Mathematical Foundations of Computer Science*, pages 338–349. Springer LNCS 379, 1989.

[646] V. Manca, A. Salibra, and G. Scollo. Equational Type Logic. *Theoretical Computer Science*, 77:131–159, 1990.

[647] L. Mandel. The Semantics of the Untyped Constrained Lambda Calculus. Technical Report Number 9319, Ludwig-Maximilians-Universität München, Oct. 1993. ⊕.

[648] L. Mandel. *Constrained Lambda Calculus*. PhD thesis, Ludwig-Maximilians-Universität München, Leopoldstraße 11b, 80802 München, Germany, Dec. 1995. ⊕.

[649] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[650] B. Marre. Toward automatic test data set selection using Algebraic Specifications and Logic Programming. In *Eigth International Conference on Logic Programming, ICLP'91*, pages 202–219. MIT Press, 1991. ⊕.

[651] B. Marre. *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques.* Nouvelle thèse, Université de Paris Sud, Orsay, 1991.

[652] B. Marre. LOFT: a tool for assisting selection of test data sets from algebraic specifications. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 799–800. Springer LNCS 915, 1995. Tool demonstration in plenary session. ⊕.

[653] B. Marre, P. Thevenod-Fosse, H. Waeselynck, P. L. Gall, and Y. Crouset. An Experimental Evaluation of Formal Testing and Statistical Testing. In H. H. Frey, editor, *Safety of Computer control systems (SAFECOMP'92)*, pages 311–316. Pergamon Press, 1992.

[654] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. CSL Technical Report SRI-CSL-93-05, SRI International, 1993.

[655] Y. Matsumoto. Some experiences in promoting reusable software. *IEEE Transactions on Software Engineering*, SE-10(5):502–513, 1984.

[656] B. Matthews. MERILL: An Equational Reasoning System in Standard ML. In C. Kirchner, editor, *Rewriting Techniques and Applications. 5th International Conference, RTA-93*, pages 331–445. Springer LNCS 690, 1993.

[657] S. Mauw and G. J. Veltink. An introduction to PSF_d. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 272 – 285. Springer LNCS 351, 1989.

[658] B. Mayoh. Galleries and Institutions. Technical Report DAIMI PB - 191, Aarhus University, 1985.

[659] J. McKinna and R. Burstall. Deliverables: a categorical approach to program development in type theory. In A. M. Bozyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993. 18th International Symposium, MFCS '93, Gdaánsk, Poland, August/September 1993*, pages 32–67. Springer LNCS 711, 1993.

[660] A. Mégrelis. *Algèbre galactique - Un procédé de calcul formel.* Thèse de Doctorat d'Université, Université de Nancy 1, Sept. 1990. ⊕.

[661] K. Meinke. Universal Algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.

[662] S. Meldal. Supporting Architecture Mappings in Concurrent Systems Design. In *Australian Software Engineering Conference '90*. IREE Australia, May 1990. ⊕.

[663] S. Meldal, S. Sankar, and J. Vera. Exploiting locality in maintaining potential causality. In *The 10th ACM Symposium on the Principles of Distributed Computing*. ACM, ACM Press, Aug. 1991. Also published as Technical Report CSL-TR-91-466, Computer Systems Laboratory, Stanford University. ⊕.

[664] S. Meldal and M. Walicki. Towards an Equational Theory of Nondeterminism. In *Proceedings of NIK'91: Norwegian Conference on Informatics*. TAPIR, 1991. ⊕.

[665] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In M. Dezani, editor, *International Conference on Typed Lambda Calculus and Applications*. Springer LNCS 902, 1995.

[666] P. Menezes, J. Costa, and A. Sernadas. Refinement Mapping for (Discrete Event) System Theory. In *Computer Aided System Technology, EUROCAST'95*. Springer LNCS 1030, 1996. ⊕.

[667] J. Meseguer. General logics. In *Proc. Logic Colloquium '87*. North-Holland, 1989.

[668] J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *ECOOP-OOPSLA '90 Conference on Object-Oriented Programming*, pages 101–115, Ottawa (Canada), Oct. 1990. ACM.

[669] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–156, 1992.

[670] J. Meseguer. A Logical Theory of Concurrent Objects and Its Realization in the Maude Language. In G. Agha, P. Wegener, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[671] J. Meseguer and J. A. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–540. Cambridge Univ. Press, 1985.

[672] J. Meseguer and N. Martí-Oliet. From Abstract Data Types to Logical Frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 48–80. Springer LNCS 906, 1995.

[673] J. Meseguer and T. Winkler. Parallel Programming in Maude. CSL Technical Report SRI-CSL-91-08, SRI International, 1991.

[674] B. Meyer. Eiffel: programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–94, 1987.

[675] B. Meyer. Reusability: the case for object-oriented design. *IEEE Software*, 4(2):50–64, 1987.

[676] A. Middeldorp. A sufficient condition for the termination of the direct sum of term rewriting systems. In *Annual IEEE Symposium on Logic in Computer Science*, pages 396–401. IEEE Computer Society Press, 1989.

[677] A. Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[678] A. Middeldorp and Y. Toyama. Completeness of Combinations of Constructor Systems. In R. Book, editor, *Rewriting Techniques and Applications. 4th International Conference, RTA-91*, pages 188–199. Springer LNCS 488, 1991.

[679] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[680] J. C. Mitchell, S. Meldal, and N. Madhav. An Extension of Standard ML Modules with Subtyping and Inheritance. In *Proc. of the 18th ACM Symp. on Principles of Programming Languages*, pages 270–278. ACM, ACM Press, 1991. Also published as Technical Report CSL-TR-91-472, Computer Systems Laboratory, Stanford University. ⊕.

[681] B. Möller. On the algebraic specification of infinite objects: ordered and continuous algebraic types. *Acta Informatica*, 22:537–578, 1985.

[682] B. Möller. Algebraic specifications with higher-order operators. In *Proc. IFIP TC2 Working Conference on Program Specification and Transformation*, pages 367–392. North-Holland, 1986.

[683] B. Möller. Higher-Order Algebraic Specifications. Habilitationsschrift, Technische Universität München, 1987.

[684] B. Möller, A. Tarlecki, and M. Wirsing. Algebraic specifications of reachable higher-order algebras. In *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, pages 154–169. Springer LNCS 332, 1988.

[685] T. Mossakowski. Parameterized Recursion Theory – A Tool for the Systematic Classification of Specification Methods. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, 1993*, Workshops in Computing, pages 139–146. Springer-Verlag, London, 1993. ⊕.

[686] T. Mossakowski. A Hierarchy of Institutions Separated by Properties of Parameterized Abstract Data Types. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 389–405. Springer LNCS 906, 1995. ⊕.

[687] T. Mossakowski. Different Types of Arrow Between Logical Frameworks. In *23rd International Colloquium on Automata, Languages, and Programming.* Springer LNCS, 1996. To appear. ⊕.

[688] T. Mossakowski. Equivalences among various logical frameworks of partial algebras. In H. K. B. et al., editor, *Computer Science Logic. 9th Workshop, CSL'95. Paderborn, Germany, September 1995.* Springer LNCS, 1996. To appear ⊕.

[689] T. Mossakowski. Parameterized Recursion Theory – A Tool for the Systematic Classification of Specification Methods. *Theoretical Computer Science*, 1996. To appear. ⊕.

[690] T. Mossakowski. Using Colimits of Parchments to Systematically Construct Institutions of Partial Algebras. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 362–376. Springer–Verlag, 1996. To appear. ⊕.

[691] P. Mosses. Abstract semantic algebras. In *Proc. IFIP TC2 Working Conf. on the Formal Description of Programming Concepts II.* North-Holland, 1983.

[692] P. D. Mosses. Unified Algebras and Institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989. ⊕.

[693] P. D. Mosses. Unified algebras and modules. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.

[694] P. D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992. ⊕.

[695] P. D. Mosses. On the Action Semantics of Concurrent Programming Languages. In *Semantics: Foundations and Applications, Proc. REX Workshop, Beekbergen, The Netherlands, 1992. LNCS 666*, pages 398–424. Springer–Verlag, 1993. ⊕.

[696] P. D. Mosses. The Use of Sorts in Algebraic Specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, volume 655 of *LNCS*, pages 66–92. Springer–Verlag, 1993. ⊕.

[697] P. D. Mosses, editor. *Proc. 1st Intl. Workshop on Action Semantics, Edinburgh, Scotland, 1994*, number NS-94-1 in Notes Series. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994. ⊕.

[698] P. D. Mosses. Unified Algebras and Abstract Syntax. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, Spain, 1992, Selected Papers*, pages 280–294. Springer LNCS 785, 1994. ⊕.

[699] P. D. Mosses. A Tutorial on Action Semantics, Mar. 1996. 50pp. Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, Spain, 1994) and FME'96 (Formal Methods Europe, Oxford, England, 1996). ⊕.

[700] P. D. Mosses. Theory and Practice of Action Semantics. In *MFCS '96, Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, Lecture Notes in Computer Science. Springer–Verlag, 1996. To appear. ⊕.

[701] P. D. Mosses and M. Musicante. An Action Semantics for ML Concurrency Primitives. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Proc. FME'94 (Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods), Barcelona*, pages 461–479. Springer LNCS 873, 1994. ⊕.

[702] P. D. Mosses (editor). Initiative for a *Common Framework for Algebraic Specification. Bulletin of the EATCS*, 59, June 1996. Also available on the WWW: `http://www.brics.dk/Projects/CoF`.

[703] A. Mück. *Eine verifizierte Implementierung funktional logischer Sprachen.* PhD thesis, Ludwig-Maximilians-Universität München, 1996. ⊕.

[704] H. Munthe-Kaas and M. Haveraaen. Coordinate Free Numerics – Part 1: How to avoid index wrestling in tensor computations. Technical report 101, Department of Informatics, University of Bergen, N-5020 BERGEN, Norway, 1996. ⊕.

[705] H. Munthe-Kaas and M. Haveraaen. Coordinate free numerics, how to avoid index-wrestling in tensor computations. *Zeitschrift fur Angewandte Mathematik und Mechanik*, 1996. Proceedings of ICIAM'95, The Third International Congress on Industrial and Applied Mathematics. Hamburg, Germany, June 1995. ⊕.

[706] D. L. Musser. On Proving Inductive Properties of Abstract Data Types. In *Proceedings of the 7th Annual ACM Conference on the Principles of Programming Languages*, 1980.

[707] P. Narendran and M. Rusinowitch. Any ground associative–commutative theory has a finite canonical system. In R. Book, editor, *Proc. 4th Int. Conf. Rewriting Techniques and Applications.* Springer LNCS 488, 1991.

[708] M. Navarro and F. Orejas. Parameterized Horn clause specifications: proof theory and correctness. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 202–216. Springer LNCS 250, 1987.

[709] M. Navarro, F. Orejas, and A. Sanchez. Characterizing Confusion. *Bulletin of the EATCS*, 49:104–109, 1993. ⊕.

[710] M. Navarro, F. Orejas, and A. Sánchez. On the Correctness of Modular Systems. In G. Scollo, editor, *AMAST 93, Enschede, Netherlands*, 1993. ⊕.

[711] M. Navarro, F. Orejas, and A. Sánchez. On the Correctness of Modular Systems. *Theoretical Computer Science*, 140(1):139–178, 1995. ⊕.

[712] D. Nazareth. *A Polymorphic Sort System for Axiomatic Specification Languages.* PhD thesis, Technische Universität München, 1995. Technical Report TUM-I9515. ⊕.

[713] M. Nenninger and F. Nickl. Implementing data structures by composition of reusable components: a formal approach. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice*, pages 134–141, 1995. ⊕.

[714] F. Nickl. *Algebraic Specification of Semantic Domain Constructions.* PhD thesis, Universität Passau, 1987.

[715] F. Nickl. Ablaufspezifikation durch Datenflußdiagramme und Axiome. In B. Wolfinger, editor, *Innovationen bei Rechen-und Kommunikationssystemen*, pages 27–34, 1994. ⊕.

[716] F. Nickl and M. Wirsing. A Formal Approach to Requirements Engineering. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Formal Methods in Programming and Their Application*, pages 312–334. Springer LNCS 735, 1994. ⊕.

[717] R. Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47:65–69, Aug. 1993. ⊕.

[718] R. Nieuwenhuis. On Narrowing, Refutation Proofs and Constraints. In J. Hsiang, editor, *Rewriting Techniques and Applications. 6th International Conference, RTA-95, Kaiserslautern (Germany)*, pages 56–70. Springer LNCS 914, 1995. ⊕.

[719] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Brückner, editor, *European Symposium on Programming*, pages 371–390. Springer LNCS 582, 1992. ⊕.

[720] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering Constrained Clauses. In D. Kapur, editor, *11th International Conference on Automated Deduction, Saratoga Springs, New York*, pages 477–491. Springer LNAI 607, 1992. Extended version to appear in Journal of Symbolic Computation. ⊕.

[721] R. Nieuwenhuis and A. Rubio. AC-Superposition with constraints: No AC-unifiers needed. In A. Bundy, editor, *12th International Conference on Automated Deduction. Nancy, France.* Springer LNAI 814, June 1994. ⊕.

[722] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995. ⊕.

[723] T. Nipkow. Non-deterministic data types: models and implementations. *Acta Informatica*, 22:629–661, 1986.

[724] T. Nipkow and Z. Qian. Modular Higher-Order E-Unification. In R. Book, editor, *Proc. 4th Int. Conf. Rewriting Techniques and Applications*, pages 200–214. Springer LNCS 488, 1991. ⊕.

[725] T. Nipkow and Z. Qian. Reduction and Unification in Lambda Calculi with Subtypes. In D. Kapur, editor, *11th International Conference on Automated Deduction, Saratoga Springs, New York*, pages 66–78. Springer LNCS 607, 1992. ⊕.

[726] P. Nivela. *Semantica de comportamiento para especificaciones algebraicas.* PhD thesis, Universitat Politècnica de Catalunya, Barcelona, 1987.

[727] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specifications. In *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, pages 184–207. Springer LNCS 332, 1988.

[728] E. Nordhagen. An Object-Oriented Calculus. In *Proceedings from 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS, Paris, 4-6 March, 1996*. Chapman & Hall, 1996. To appear. ⊕.

[729] A. Oberschelp. Untersuchungen zur mehrsortigen Quantorenlogik. *Math. Annalen*, 145(1):297–333, 1962.

[730] P. Olveczki, P. Kosiuczenco, and M. Wirsing. Steamboiler specification problem: an algebraic object-oriented solution. Submitted for publication, 1996. ⊕.

[731] P. Ölveczky and O. Lysne. Order-sorted Termination: the Unsorted Way. In *Proceedings from NIK'95: Norwegian Conference on Informatics, Gran (Hadeland)*, pages 307–318. TAPIR, Nov. 1995. To be submitted to Algebraic and Logic programming (ALP'96). ⊕.

[732] F. Orejas. Characterizing composability of abstract implementations. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*. Springer LNCS 158, 1983.

[733] F. Orejas. A proof system for checking composability of implementations of abstract data types. In *Proc. of the Intl. Symp. on Semantics of Data Types*. Springer LNCS 173, 1984.

[734] F. Orejas. On implementability and computability in abstract data types. In *Algebra, Logics and Combinatorics in Computer Science*. North-Holland, 1985.

[735] F. Orejas. The role of abstraction in program development (response). In *Proc. IFIP Congress 1986*, pages 143–146. North-Holland, 1986.

[736] F. Orejas. A characterization of passing compatibility for parameterized specifications. *Theoretical Computer Science*, 51:205–214, 1987.

[737] F. Orejas, M. Navarro, and A. Sánchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 93–125. Springer LNCS 655, 1993. ⊕.

[738] F. Orejas, M. Navarro, and A. Sánchez. Algebraic Implementation of Abstract Data Types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science*, 1996. To appear ⊕.

[739] F. Orejas, V. Sacristán, and S. Clerici. Development of algebraic specifications with constraints. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical methods in Computer Science with aspects from Topology*, pages 36–49. Springer LNCS 393, 1989.

[740] O. Owe. A Comparison of Partial Logics. In *Proceedings of NIK'91: Norwegian Conference on Informatics*. TAPIR, 1991. ⊕.

[741] O. Owe. Axiomatic Treatment of Processes with Shared Variables Revisited. *Formal Aspects of Computing*, 4:323–340, 1992. ⊕.

[742] O. Owe. Partial Logics Reconsidered: A Conservative Approach. *Formal Aspects of Computing*, 5(3):208–223, 1993. ⊕.

[743] O. Owe. A Proposal for a Process Logic. Technical Report RR-187, Dept. of Informatics, Univ. of Oslo, 1994. ⊕.

[744] O. Owe. On Convergent Rewriting in Three-Valued Propositional Logic. In *Proceedings of the 7. Nordic Workshop on Programming Theory*, pages 298–309. Programming Methodology Group, Göteborg and Chalmers University, Technical Report 86, 1996. NWPT'95, Göteborg, 1-3 Nov. 1995. ⊕.

[745] O. Owe and O.-J. Dahl. Generator Induction in Order Sorted Algebras. *Formal Aspects of Computing*, 3(1), 1991. ⊕.

[746] P. Padawitz. Towards a proof theory of parameterized specifications. In *Proc. of the Intl. Symp. on Semantics of Data Types*, pages 375–391. Springer LNCS 173, 1984.

[747] P. Padawitz. Parameter preserving data type specifications. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 323–341. Springer LNCS 186, 1985.

[748] P. Padawitz. *Computing in Horn Clause Theories.* EATCS Monographs on Theoretical Computer Science, Vol. 16. Springer, 1988.

[749] J. Padberg. Algebraic high-level net transformation systems: A survey over theory and applications. *Bulletin of the EATCS*, pages 102–110, 1993. ⊕.

[750] F. Paris-Presicce and A. Pierantonio. An Algebraic Theory of Class Specification. *ACM Transactions on Software Engineering and Methodology*, 3:166–169, Apr. 1994. ⊕.

[751] F. Parisi-Presicce. Partial composition and recursion of module specifications. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 217–231. Springer LNCS 249, 1987.

[752] F. Parisi-Presicce. Product and iteration of module specifications. In *Proc. 1988 Colloq. on Trees in Algebra and Programming*, pages 149–164. Springer LNCS 299, 1988.

[753] F. Parisi-Presicce and A.Pierantonio. On the Algebraic Specification of Classes and Inheritance in OOP. In A. Miola and M. Temperin, editors, *Advances in Symbolic Computation Systems*, Texts and Monographs in Symbolic Computation. Springer, 1996. ⊕.

[754] F. Parisi-Presicce and A. Pierantonio. An Algebraic Framework to Compare and Derive the Inheritance Relation. Technical Report TR 5/92, Dipartimento di Matematica Pura ed Applicata, Università de L'Aquila, 1992. ⊕.

[755] F. Parisi-Presicce and A. Pierantonio. Towards the Algebraic Specification of Classes in Object Oriented Programming. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science. Essays and Tutorials*, pages 183–194. World Scientific Series in Computer Science — Vol. 40, 1993. ⊕.

[756] F. Parisi-Presicce and A. Pierantonio. Reusing Object Oriented Design: an Algebraic Approach. In E. Bertino and S. Urban, editors, *Proc. Object Oriented Methodologies and Systems*, pages 329–345. Springer LNCS 858, 1994. ⊕.

[757] F. Parisi-Presicce and A. Pierantonio. Structured Inheritance for Algebraic Class Specifications. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, 1992*, pages 295–309. Springer LNCS 785, 1994. ⊕.

[758] F. Parisi-Presicce and A. Pierantonio. Dynamical Behaviour of Object Systems. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 406–419. Springer LNCS 906, 1995. ⊕.

[759] F. Parisi-Presicce and S. Veglioni. Heterogeneous Unified Algebras. In A. M. Bozyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993. 18th International Symposium, MFCS '93, Gdaánsk, Poland, August/September 1993*, pages 618–628. Springer LNCS 711, 1993. ⊕.

[760] F. Parodi and G. Reggio. METAL: a Metalanguage for SMoLCS. Technical Report DISI–TR–94–13, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994. ⊕.

[761] H. Partsch. An exercise in the transformational derivation of an efficient program by joint development of control and data structure. *Science of Computer Programming*, 3:1–35, 1983.

[762] H. Partsch. Algebraic requirements definition: a case study. *Technology and Science of Informatics*, 5(1):21–36, 1986.

[763] H. Partsch. *Specification and Transformation of Programs — A Formal Approach to Software Development.* Springer, 1990.

[764] H. Partsch and M. Broy. Examples for changes of types and object structures. In *Proc. of the International Summer School on Program Construction*, pages 421–463. Springer LNCS 69, 1979.

[765] H. Partsch and A. Laut. From requirements of their formalization — a case study on the stepwise development of algebraic specifications. In *Proc. 7. Fachtagung Programmiersprachen und Programmentwicklung*, pages 117–132. Springer IFB 53, 1982.

[766] H. Partsch and P. Pepper. Abstract data types as a tool for requirements engineering. In D. Kroenig and G. Hommel, editors, *Requirements Engineering*, pages 42–55. Springer IFB 74, 1983.

[767] H. Partsch and P. Pepper. Program transformations expressed by algebraic type manipulations. *Technology and Science of Informatics*, 5(3):197–212, 1986.

[768] J. Peleska. Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In M. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Method Europe*, pages 39–59. Springer LNCS 1051, 1996.

[769] P. Pepper. A study on transformational semantics. In *Proc. of the International Summer School on Program Construction*, pages 322–405. Springer LNCS 69, 1979.

[770] P. Pepper. Algebraic techniques for program specification. In *Proc. Workshop on Program Transformations and Programming Environments, Munich. NATO ASI Series, Vol F8.*, pages 231–244. Springer, 1984.

[771] P. Pepper. Inferential techniques for program development. In *Proc. Workshop on Program Transformations and Programming Environments, Munich. NATO ASI Series, Vol F8.*, pages 275–290. Springer, 1984.

[772] P. Pepper. A simple calculus of program transformations (inclusive of induction). *Science of Computer Programming*, 9(3):221–262, 1987.

[773] P. Pepper. Deductive derivation of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*. Kluwer Academic Publishers, 1993. ⊕.

[774] P. Pepper. Program development in an algebraic setting. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development. IFIP TC2/WG 2.1 State-of-the-Art Report*, pages 225–262. Springer LNCS 755, 1993. ⊕.

[775] P. Pepper, R. Betschko, S. Dick, and K. Didrich. Realizing Sets by Hash Tables. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*. Springer LNCS 1009, 1995. ⊕.

[776] P. Pepper, J. Exner, and M. Südholt. Funktionale Programmierung für massiv parallele Systeme. In H. Reichel, editor, *Informatik — Wirtschaft — Gesellschaft*, Informatik aktuell, pages 551–556. Springer-Verlag, 1993. ⊕.

[777] P. Pepper and M. Wirsing. A Method for the Development of Correct Software. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, pages 27–57. Springer LNCS 1009, 1995. ⊕.

[778] H. Perdrix. Program synthesis from specifications. In *Proc. ESPRIT Conf. '85: Status Report of Continuing Work*, pages 371–385. North-Holland, 1986.

[779] M. Phalippou. *Relations d'implantation et hypothèses de test sur des automates à entrées sorties.* Nouvelle thèse, Université de Bordeaux I, 1994.

[780] A. Pierantonio. Making Static Dynamic: Towards an Axiomatization for Dynamic ADT's. In *Proc. International Workshop on Communication Based Systems*, Berlin, 1994. ⊕.

[781] F. Pinheiro and J. Goguen. An Object-Oriented Tool for Tracing Requirements. *IEEE Software*, pages 52–64, Mar. 1996. Special issue of papers from ICRE '96. ⊕.

[782] D. A. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65(2&3):182–215, 1985.

[783] A. Pliškevičiene, R. Pliuškevičius, M. A. Walicki, and S. Meldal. On specialization of derivations in axiomatic equality theories. In A. Nerode and Y. V. Matiyasevic, editors, *Proc. of the Symposium on Logical Foundations of Computer Scienc*, pages 291–303. Springer LNCS 813, July 1994. ⊕.

[784] D. Plump. Implementing Term Rewriting by Graph Reduction: Termination of Combined Systems. In *Proc. Conditional and Typed Rewriting Systems*, volume 516 of *Lecture Notes in Computer Science*, pages 307–317. Springer-Verlag, 1991. ⊕.

[785] D. Plump. Collapsed tree rewriting: Completeness, confluence, and modularity. In *Proc. Conditional Term Rewriting Systems*, pages 97–112. Springer LNCS 656, 1993. ⊕.

[786] D. Plump. *Evaluation of Functional Expressions by Hypergraph Rewriting*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, 1993. ⊕.

[787] D. Plump. Critical Pairs in Term Graph Rewriting. In *Proc. Mathematical Foundations of Computer Science 1994*, pages 556–566. Springer Lecture Notes in Computer Science 841, 1994. ⊕.

[788] A. Poigné. Another Look at Parameterization Using Algebraic Specifications with Subsorts. In *Proc. 1981 Symp. on Mathematical Foundations of Computer Science*. Springer LNCS 176, 1984.

[789] A. Poigné. Foundations are Rich Institutions, but Institutions Are Poor Foundations. In H. Ehrig, H. Herrlich, H. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science*, pages 82–101. Springer LNCS 393, 1989.

[790] Z. Qian. Higher-order order-sorted algebras. In *Proc. 2nd Intl. Conf. on Algebraic and Logic Programming*. Springer LNCS 463, 1990.

[791] Z. Qian. An Algebraic Semantics of Higher-Order Types with Subtypes. *Acta Informatica*, 30:569–607, 1993. ⊕.

[792] Z. Qian. Linear Unification of Higher-Order Patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development. 4th International Joint Conference CAAP/FASE*, pages 391–405. Springer LNCS 668, 1993. ⊕.

[793] Z. Qian. Another Look at Parameterization for Order-Sorted Algebraic Specifications. *Journal of Computer and System Sciences*, 49(3), 1994. ⊕.

[794] Z. Qian. Higher-Order Equational Logic Programming. In *Proc. 21st ACM Symp. Principles of Programming Languages*, pages 254–267. ACCM Press, 1994. ⊕.

[795] Z. Qian. Unification of Higher-Order Patterns in Linear Time and Space. *Journal of Logic and Computation*, 1996. ⊕.

[796] Z. Qian and B. Krieg-Brückner. Object-Oriented Functional Programming and Type Reconstruction. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 441–460. Springer–Verlag, 1996. To appear. ⊕.

[797] Z. Qian and B. Krieg-Brückner. Object-Oriented Functional Programming with Late Binding. In *Proc. 10th European. Conf. on Object-Oriented Programming*. Springer LNCS, 1996. ⊕.

[798] Z. Qian and B. Krieg-Brückner. TOFL: A Typed Object-Oriented Functional Programming Language with Late Binding. Technical report, FB Informatik, Universität Bremen, Mar. 1996. ⊕.

[799] Z. Qian and T. Nipkow. Reduction and Unification in Lambda Calculi with a General Notion of Subtype. *Journal of Automated Reasoning*, 12:389–406, 1994. ⊕.

[800] Z. Qian and K. Wang. Higher-Order E-Unification for Arbitrary Theories. In K. Apt, editor, *Proc. 1992 Joint Int. Conf. and Symp. on Logic Programming*, pages 52–66. MIT Press, 1992. ⊕.

[801] Z. Qian and K. Wang. Modular AC Unification of Higher-Order Patterns. In J.-P. Jouannaud, editor, *Proc. Int. Conf. on Constraints in Computational Logics*, pages 105–120. Springer LNCS 845, 1994. ⊕.

[802] U. Reddy. Term rewriting induction. In M. Stickel, editor, *10th International Conference on Automatic Deduction, Kaiserslautern (Germany)*, pages 162–177. Springer LNAI 449, 1990.

[803] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, 1994. ⊕.

[804] F. Regensburger. The calculus of SPECTRUM. TUM-I 9424, Institut für Informatik, Technische-Universität München, 1994. ⊕.

[805] G. Reggio. Entities: An Institution for Dynamic Systems. In *Recent Trends in Data Type Specification. 7th Workshop on Specification of Abstract Data Types*, pages 246–265. Springer LNCS 534, 1991.

[806] G. Reggio. Event logic for specifying abstract data types. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France 1991*, pages 292–309. Springer LNCS 655, 1993. ⊕.

[807] G. Reggio, D. Bertello, and A. Morgavi. The Reference Manual for the SMoLCS Methodology. Technical Report DISI–TR–94–12, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994. ⊕.

[808] G. Reggio and E. Crivelli. Specification of a hydroelectric power station: Revised tool-checked version. Technical Report DISI–TR–94–17, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1994. ⊕.

[809] G. Reggio and V. Filippi. Specification of a High-Voltage Substation: Revised Tool-Checked Version. Technical Report DISI-TR-95-09, Dipartimento di Informatica e Scienze dell'Informazione – Università di Genova, Italy, 1995. ⊕.

[810] H. Reichel. Initially restricting algebraic theories. In *Proc. 1980 Symp. on Mathematical Foundations of Computer Science*, pages 504–514. Springer LNCS 88, 1980.

[811] H. Reichel. Behavioural equivalence — a unifying concept for initial and final specification methods. In *Proc. 3rd. Hungarian Comp. Sci. Conference*, pages 27–39, 1981.

[812] H. Reichel. Behavioural validity of equations in abstract data types. In *Proc. of the Vienna Conference on Contributions to General Algebra*, pages 301–324. Teubner-Verlag, 1985.

[813] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, 1987.

[814] H. Reichel. An approach to object semantics based on terminal co–algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995. ⊕.

[815] H. Reichel. Specification Semantics. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications.* Chapman and Hall, 1996. To appear. ⊕.

[816] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[817] B. Reus and T. Streicher. Verifying Properties of Module Construction in Type Theory. In A. M. Borzyszkowski and S. Skolowski, editors, *Mathematical Foundations of Computer Science 1993*, pages 660–670. Springer LNCS 711, 1993. ⊕.

[818] L. Ribeiro, H. Ehrig, and J. Padberg. Formal Development of Concurrent Systems Using Algebraic High-Level Nets and Transformations. In *Proc. VII Simpósio Brasileiro de Engenharia de Software*, pages 1–16, Rio de Janeiro, 1993. ⊕.

[819] C. Roques. *Modularité dans les spécifications algébriques, Théorie et application.* PhD thesis, Thèse de Doctorat, Université Paris-Sud, 1994. ⊕.

[820] J. Rouyer. Développement d'algorithmes dans le calcul des constructions. Thèse de l'Institut National Polytechnique de Lorraine, Mar. 1994. ⊕.

[821] J. Rouyer and P. Lescanne. Verification and Programming of First-order Unification in the Calculus of Constructions with Inductive Types. *Journal of Logic and Computation*, 1994. To appear. ⊕.

[822] A. Rubio. *Automated deduction with ordering and equality constrained clause*. PhD thesis, Technical University of Catalonia, Barcelona, Spain, 1994. ⊕.

[823] A. Rubio. Extension Orderings. In *22th International Conference on Automata, Languages and Programming*, LNCS. Springer-Verlag, 1995. ⊕.

[824] A. Rubio and R. Nieuwenhuis. A Precedence-Based Total AC-Compatible Ordering. In C. Kirchner, editor, *Rewriting Techniques and Applications. 5th International Conference, RTA-93*, pages 374–388. Springer LNCS 690, 1993. ⊕.

[825] A. Rubio and R. Nieuwenhuis. A total AC-compatible ordering based on RPO. *Theoretical Computer Science*, 142(2):209–227, 1995. ⊕.

[826] R. Rubio. Theorem Proving modulo Associativity. In *Conference of the European Association for Computer Science Logic, Paderborn, Germany*, LNCS. Springer-Verlag, 1995. ⊕.

[827] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.

[828] C. V. Russo. Standard ML type generativity as existential quantification. Draft report, University of Edinburgh, ⊕, 1996.

[829] C. V. Russo. *Types for Program Modules*. PhD thesis, University of Edinburgh, 1996. Forthcoming, ⊕.

[830] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Specification and Stepwise Refinement. In J. de Meer, V. Heymer, and R. Roth, editors, *Proc. Open Distributed Processing, Berlin (D), 8.-11. Okt. 1991 (IFIP Transactions C: Communication Systems, Vol. 1)*, pages 99–121. North-Holland, 1992. ⊕.

[831] A. Salibra and G. Scollo. A Soft Stairway to Institutions. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification. Dourdan France, August 1991*, pages 310–329. Springer LNCS 655, 1993.

[832] SALSA (group composed of D. Bert and M. Bidoit and C. Choppy and R. Echahed and J.-M. Hufflen and J.-P. Jacquot and M. Lemoine and N. Levy and J.-C. Reynaud and C. Roques and F. Voisin). Opération SALSA: Structure d'Accueil pour Spécifications Algébriques. Final report, 1994. ⊕.

[833] D. Sannella. A Set–Theoretic Semantics for CLEAR. *Acta Informatica*, 21:443–472, 1984.

[834] D. Sannella. Static and logical correctness conditions in formal development of modular programs. Draft report, University of Edinburgh, ⊕, 1993.

[835] D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992. ⊕.

[836] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 67–77, 1985.

[837] D. Sannella and A. Tarlecki. Specifications in an Arbitrary Institution. *Information and Computation*, 76:164–210, 1988.

[838] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[839] D. Sannella and A. Tarlecki. Extended ML: Past, Present and Future. In *Recent Trends in Data Type Specification. 7th Workshop on Specification of Abstract Data Types*, pages 297–322. Springer LNCS 534, 1991. ⊕.

[840] D. Sannella and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Model–Theoretic Foundations. In W. Kuich, editor, *Automata, Languages and Programming*, pages 656–671. Springer LNCS 623, 1992. ⊕.

[841] D. Sannella and A. Tarlecki. Model-theoretic foundations for program development: basic concepts and motivation. Internal Report 791, Institute of Computer Science, Polish Academy of Sciences, 1995. ⊕.

[842] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge University Press, 1996. To appear. ⊕.

[843] D. T. Sannella and A. Tarlecki. Building specifications in an arbitrary institution. In *Proc. of the Intl. Symp. on Semantics of Data Types*, pages 337–356. Springer LNCS 173, 1984.

[844] D. T. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. In *Proc. Workshop on Category Theory and Computer Programming*, pages 364–389. Springer LNCS 240, 1986.

[845] D. T. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[846] 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*, pages 96–110. Springer LNCS 249, 1987.

[847] D. T. Sannella and A. Tarlecki, editors. *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*. Springer LNCS 332, 1988.

[848] D. T. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 375–389. Springer LNCS 352, 1989.

[849] D. T. Sannella and L. A. Wallen. A calculus for the construction of modular Prolog programs. In *Proc. 1987 IEEE Symp. on Logic Programming*, pages 368–378, 1987.

[850] D. T. Sannella and M. Wirsing. Implementation of parameterised specifications. In *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, pages 473–488. Springer LNCS 140, 1982.

[851] D. T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, pages 413–427. Springer LNCS 158, 1983.

[852] M. Sato. Towards a mathematical theory of program synthesis. In *Proc. 6th Intl. Joint Conf. on Artificial Intelligence*, pages 757–762, 1979.

[853] B. Schätz and K. Spies. Formale Syntax zur logischen Kernsprache der FOCUS-Entwicklungsmethodik. TUM-I 9529, Technische Universität München, 1995.

[854] W. L. Scherlis and D. S. Scott. First steps towards inferential programming. In *Proc. IFIP Congress 1983*, pages 199–212. North-Holland, 1983.

[855] K. Schmid and R. Fettig. Towards an efficient construction of test sets for deciding ground reducibility. In J. Hsiang, editor, *Rewriting Techniques and Applications. 6th International Conference, RTA-95, Kaiserslautern (Germany)*, pages 86–100. Springer LNCS 914, 1995. ⊕.

[856] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*. PhD thesis, Universität Kaiserslautern (Germany), 1987.

[857] P. Schmitz, M. Timm, and M. Windfuhr, editors. *Requirements Engineering '87*. GMD-Studien 121. GMD, 1987.

[858] O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, University Edinburgh, 1986.

[859] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh, 1987.

[860] O. Schoett. Behavioural Correctness of Data Representation. *Science of Computer Programming*, 14:43–57, 1990.

[861] K. Schütte. *Proof Theory*. Springer, 1977.

[862] D. S. Scott. Domains for denotational semantics. In *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, pages 577–613. Springer LNCS 140, 1982.

[863] A. Sernadas, J. Costa, and C. Sernadas. An Institution of Object Behaviour. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, 1992*, pages 337–350. Springer LNCS 785, 1994. ⊕.

[864] A. Sernadas and H.-D. Ehrich. What is an Object, After All? In R. Meersman, W. Kent, and S. Khosl, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 39–69. North-Holland, 1991. ⊕.

[865] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. Abstract Object Types: A Temporal Perspective. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Colloq. on Temporal Logic and Specification*, pages 324–500. Springer LNCS 398, 1989.

[866] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. The Basic Building Blocks of Information Systems. In E. Falkenberg and P. Lindgreen, editors, *Information Systems Concepts: An In-Depth Analysis*, pages 225–246. North-Holland, 1989.

[867] A. Sernadas and C. Sernadas. The use of ER abstractions in knowledge representation. In P. Chen, editor, *Entity Relationship Approach: The Use of ER Concept in Knowledge Representation*, pages 224–231. North-Holland, 1985.

[868] A. Sernadas and C. Sernadas. Conceptual modeling for knowledge-based DSS development. In C. Holsapple and A. Whinston, editors, *Decision Support Systems: Theory and Application*, pages 91–135. Springer, 1987.

[869] A. Sernadas and C. Sernadas. Object Certification. Research report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1994. Presented at DAISD Workshop 94 ⊕.

[870] A. Sernadas, C. Sernadas, and J. Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5):603–630, 1995. Available as Research Report since 1992 ⊕.

[871] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P. M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Data Bases VLDB'87*, pages 107–116. VLDB Endowment Press, 1987.

[872] A. Sernadas, C. Sernadas, and J. Ramos. A Temporal Logic Approach to Object Certification. *Journal of Data and Knowledge Engineering*, 1996. In print ⊕.

[873] C. Sernadas, P. Gouveia, J. Gouveia, A. Sernadas, and P. Resende. The Reification Dimension in Object-Oriented Data Base Design. In D. Harper and M. Norrie, editors, *Specification of Database Systems*, pages 275–299. Springer-Verlag, 1992. ⊕.

[874] C. Sernadas, P. Gouveia, and A. Sernadas. Refinement: Layered Definition of Conceptual Schemata. In E. Falkenberg, C. Rolland, and E. El-Sayed, editors, *Information System Concepts*, pages 19–51. North-Holland, 1992. ⊕.

[875] C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large Object-Oriented Design of Information Systems. In F. V. Assche, B. Moulin, and C. Rolland, editors, *The Object-Oriented Approach in Information Systems*, pages 209–232. North-Holland, 1991. ⊕.

[876] H. Shi. An efficient matching algorithm for convergent rewrite systems. In B. Krieg-Brückner, editor, *Programmentwicklung durch Spezifikation und Transformation, Band 2*, Informatik Bericht 10/94. Universität Bremen, 1994. ⊕.

[877] H. Shi. *Extended Matching with Applications to Program Transformation.* Dissertation. Universität Bremen, 1994. ⊕.

[878] H. Shi and B. Wolff. A finitary matching algorithm for constructor based theories. In B. Krieg-Brückner, editor, *Programmentwicklung durch Spezifikation und Transformation, Band 2*, Informatik Bericht 10/94. Universität Bremen, 1994. ⊕.

[879] H. Shi and B. Wolff. Second-order matching with matching combinators. In B. Krieg-Brückner, editor, *Programmentwicklung durch Spezifikation und Transformation, Band 1*, Informatik Bericht 1/94. Universität Bremen, 1994. ⊕.

[880] M. Sintzoff. Expressing program developments in a design calculus. In *Proc. of the International Summer School on Logic of Programming and Calculi of Discrete Design, Marktoberdorf. Nato ASI Series, Vol. F36.*, pages 343–365. Springer, 1987.

[881] R. D. Smith. Top-down synthesis of simple divide and conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[882] T. SMoLCS group. A User's Guide to the SMoLCS Toolset. Technical Report on Project "Sistemi Informatici e Calcolo Parllelo" 133, C.N.R., Italy, 1995. ⊕.

[883] G. Smolka. Order-Sorted Horn Logic Semantics and Deduction. Draft of September 12, 1986, 1986.

[884] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order sorted equational computation. In *Proc. Colloq. on Resolution of Equations in Algebraic Structures.* Academic Press, 1989.

[885] D. Soni, R. L. Nord, and C. Hofmeister. Software architectures in industrial applications. In *The 17th International Conference on Software Engineering.* ACM, Apr. 1995.

[886] J. M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[887] J. M. Spivey. *The Z Notation.* Prentice Hall, 1989.

[888] R. Steinbrüggen. Program development using transformational expressions. Report TUM-I8206, Technische Universität München, 1982.

[889] K. Stølen. A Refinement Relation Supporting the Transition from Unbounded to Bounded Communication Buffers. In B. Möller, editor, *Mathematics of Program Construction. Third International Conference, MPC'95 Kloster Irsee, Germany, July 1995*, pages 423–451. Springer LNCS 947, 1995.

[890] K. Stølen. Development of SDL Specifications in Focus. In R. Bræk, editor, *Proc. SDL Forum'95*, pages 269–278, 1995. To appear.

[891] K. Stølen. Assumption/Commitment Rules for Data-flow Networks — with an Emphasis on Completeness . In *ESOP'96*, 1996. To appear.

[892] K. Subieta and M. Missala. Denotational semantics of query languages. *Information Systems*, 12, 1987.

[893] P. A. Subrahmanyam. Nondeterminism in abstracta data types. In *Proc. 8th Intl. Colloq. on Automata, Languages and Programming*, pages 148–164. Springer LNCS 115, 1981.

[894] M. Südholt. Data Distribution Algebras — A Formal Basis for Programming Using Skeletons. In *Proceedings of the 4th International Conference on Programming Methods, Concepts, and Calculi (PROCOMET '94)*, pages 38–57. IFIP, Consiglio Nazionale delle Ricerche, June 1994. ⊕.

[895] W. Swartout. GIST english generator. In *Proc. of American Assoc. for Artificial Intelligence*, 1982.

[896] A. Tarlecki. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science*, 37:269–304, 1986.

[897] A. Tarlecki. Quasi-varieties in abstract algebraic institutions. *Journal of Computer and System Sciences*, 33:333–360, 1986.

[898] A. Tarlecki and M. Wirsing. Continuous abstract data types. *Fundamenta Informaticae*, 9:95–125, 1986.

[899] J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. In *Proc. 6th Intl. Colloq. on Automata, Languages and Programming*, pages 596–615. Springer LNCS 71, 1979.

[900] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Data type specification: parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems*, 4:711–773, 1982.

[901] Y. Toyama. Counterexamples to termination of the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.

[902] Y. Toyama. On the Church-Rosser Property for the Direct Sum of Term Rewriting Systems. *Journal of the Association for Computing Machinery*, 34(1):128–143, 1987.

[903] P. O. Tvete and S. Meldal. Causality Breakdown at Omega. In R.-J. R. Back and K. Sere, editors, *The Fifth Nordic Workshop on Program Correctness*, pages 194–203. Übo Akademi, 1994. ⊕.

[904] J. Underwood. *Aspects of the Computational Content of Proofs.* PhD thesis, Cornell University, 1994.

[905] J. Underwood. Tableau for Intuitionistic Predicate Logic as Metatheory. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods.* Springer LNAI 918, 1995. ⊕.

[906] J. Underwood. Typing Abstract Data Types. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 437–452. Springer LNCS 906, 1995. ⊕.

[907] J. Underwood. Algebraic and Type Theoretic Notions of Equality. Draft report, University of Edinburgh, ⊕, 1996.

[908] J. van de Pol. Termination Proofs for Higher-order Rewrite Systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting. First International Workshop, HOA'93. Amsterdam*, pages 305–325. Springer LNCS 816, 1994.

[909] N. van Diepen. Implementation of modular algebraic specifications. In *Proc. 1988 European Symp. on Programming*, pages 64–78. Springer LNCS 300, 1988.

[910] N. van Diepen and H. Partsch. Some aspects of formalizing informal requirements. In *Proc. METEOR Workshop*, 1989.

[911] P. van Eijk. Tools for LOTOS, a Lotosfere overview. Memoranda Informatica 91-25, Universiteit Twente - Faculteit der Informatica, Enscede, 1991.

[912] V. Čyras and M. Haveraaen. Programming with Data Dependencies: a Comparison of Two Approaches. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *The 6th Nordic Workshop on Program Theory, NWPT6. Aarhus, Denmark, October 1994*, number NS-94-6 in BRICS Notes, pages 112–126, December 1994. ⊕.

[913] I. Velle. Abstrakte datatyper p parallelle datamaskiner. Master's thesis, Universitetet i Bergen, 1993.

[914] A. I. Vermesan and S. Meldal. An algebraic approach to refinement of KBS specifications. In M. Ayel and M.-C. Rousset, editors, *EuroVaV'95*, pages 265–276. ADERIAS-LIA, June 1995. ⊕.

[915] A. I. Vermesan and S. Meldal. Verification Oriented Development of Knowledge Based Systems. *New Review of Applied Expert Systems*, 1996. To appear. ⊕.

[916] L. Vigneron. Associative Commutative Deduction with constraints. In A. Bundy, editor, *12th International Conference on Automated Deduction*, Nancy, France, June 1994. Springer-Verlag LNAI.

[917] L. Vigneron. *Déduction automatique avec contraintes symboliques dans les théories équationnelles.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, Nov. 1994.

[918] P. Viry. *La réécriture concurrente.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, Oct. 1992. ⊕.

[919] N. Vlachantonis. A Case Study for an Open CASE System: The TROLL *light* Development Environment. In V. Mařík, J. Lažanský, and R. Wagner, editors, *Proc. 4th Int. Conf. on Database and Expert Systems Applications (DEXA'93)*, pages 673–676. Springer LNCS 720, 1993. ⊕.

[920] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich. Towards Reliable Information Systems: The KORSO Approach. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93)*, pages 463–482. Springer LNCS 685, 1993. ⊕.

[921] F. Voisin. CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7(1):61–86, 1986.

[922] A. Wagner and M. Gogolla. Defining Operational Behavior of Object Specifications by Attributed Graph Transformations. *Fundamenta Informaticae*, 1995. accepted, ⊕.

[923] E. G. Wagner and H. Ehrig. Canonical constraints for parameterized data types. *Theoretical Computer Science*, 50:323–351, 1987.

[924] M. Walicki and S. Meldal. Computations: A deterministic semantics of nondeterminism. In S. Meldal and M. Haveraaen, editors, *The 4th Nordic Workshop on Program Correctness*, 1992. ⊕.

[925] M. Walicki and S. Meldal. Initiality + Nondeterminism Implies Junk. In M. Haveraaen, editor, *NIK'93*, pages 129–138. Tapir, Nov. 1993. ⊕.

[926] M. Walicki and S. Meldal. Sets and nondeterminism. In *ICLP'93 Workshop on Logic Programming With Sets*, 1993. ⊕.

[927] M. A. Walicki and S. Meldal. A complete calculus for the multialgebraic and functional semantics of nondeterminism. *ACM Transactions on Programming Languages and Systems*, 17(2):366–393, 1995. ⊕.

[928] M. A. Walicki and S. Meldal. Generated models and the omega-rule: the nondeterministic case. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, pages 424–438. Springer LNCS 915, 1995. ⊕.

[929] M. A. Walicki and S. Meldal. Multialgebras, Power Algebras and Complete Calculi of Identities and Inclusions. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 453–468. Springer LNCS 906, 1995. ⊕.

[930] M. A. Walicki and S. Meldal. Nondeterministic Operators in Algebraic Frameworks. Technical Report CSL-TR-95-664, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, Mar. 1995. ⊕.

[931] M. A. Walicki and S. Meldal. Singular and plural nondeterministic parameters. *SIAM Journal of Computation*, 1996. To appear. ⊕.

[932] T. Walsh. A Divergence Critic. In A. Bundy, editor, *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, pages 14–28. Springer LNAI 814, June 1994. ⊕.

[933] M. Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19:27–44, 1979.

[934] M. Wand. First-order identites as a defining language. *Acta Informatica*, 14:337–357, 1980.

[935] P. Watson and J. Dick. Least sorts in Order-Sorted Term Rewriting. Technical report, Royal Holloway and Bedford New College, University of London, Nov. 1989.

[936] H. Weber and H. Ehrig. Specification of modular systems. *IEEE Transactions on Software Engineering*, SE-12(7):784–798, 1986.

[937] A. Werner. A Semantic Approach to Order-Sorted Rewriting. In C. Kirchner, editor, *Rewriting Techniques and Applications. 5th International Conference, RTA-93*, pages 47–61. Springer LNCS 690, 1993.

[938] R. Wieringa. Equational Specification of Dynamic objects. Technical Report 91-1, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1991.

[939] D. Wile. Program developments: formal explanations of implementations. In W. A. Agresti, editor, *New Paradigms for Software Development*, pages 239–248. IEEE Computer Society Press/North-Holland, 1986.

[940] J. M. Wing. Specifying Avalon objects in Larch. In *Proc. Joint Conf. on Theory and Practice of Software Development*, pages 61–80. Springer LNCS 352, 1989.

[941] M. Wirsing. Structured Algebraic Specifications. A Kernel Language. *Theoretical Computer Science*, 42, 1986.

[942] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[943] M. Wirsing. Structured Specifications: Syntax, Semantics and Proof Calculus. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification. NATO ASI Series F: Computer and System Sciences. Volume 94*, pages 411–442. Springer, 1993. ⊕.

[944] M. Wirsing. Algebraic Specification Languages: An Overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 81–115. Springer LNCS 906, 1995. ⊕.

[945] M. Wirsing and J. A. Bergstra. *Algebraic Methods: Theory, Tools and Applications*. Springer 394, 1989.

[946] M. Wirsing and M. Broy. An analysis of semantic models for algebraic specifications. In *Proc. Intl. Summer School on Theoretical Foundations of Programming Methodology*. Reidel, 1982.

[947] M. Wirsing, R. Hennicker, and R. Stabl. MENU: an example for the systematic reuse of specifications. In *Proc. 2nd European Conf. on Software Engineering*, pages 20–41. Springer LNCS 387, 1989.

[948] M. Wirsing, F. Nickl, and U. Lechner. Concurrrent Object-Oriented Design Specification in SPECTRUM. In P.-Y. Schobbens, editor, *MeDiCis'94: Methodology for the Development of Computer System Specifications, Working Notes of a Workshop held in the chateau de Namur 1994*, pages 163–179, 1994. ⊕.

[949] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, and M. Broy. On hierarchies of abstract data types. *Acta Informatica*, 20:1–33, 1983.

[950] L. With. Completeness and Confluence of Order-Sorted Term Rewriting. In M. Rusinowitch and J. L. Rémy, editors, *Proceedings 3rd International Workshop on Conditional Term Rewriting Systems, Pont-à-Mousson (France)*, pages 393–407. Springer LNCS 656, July 1992.

[951] M. Wolczko. Semantics of Smalltalk-80. In *Proc. 1987 European Conf. on Object Oriented Programming*. Springer LNCS 276, 1987.

[952] D. A. Wolfram and J. Goguen. A sheaf semantics for FOOPS expressions (extended abstract). In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 81–98. Springer LNCS 612, 1992. ⊕.

[953] U. Wolter. An Algebraic Approach to Deduction in Equational Partial Horn Theories. *J. Inf. Process. Cybern. EIK*, 27(2):85–128, 1990.

[954] U. Wolter. Institutional Frames. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Abstract Data Types*, pages 469–482. Springer LNCS 906, 1995. ⊕.

[955] U. Wolter, K. Didrich, F. Cornelius, M. Klar, R. Wessäly, and H. Ehrig. Realizing Sets by Hash Tables. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*. Springer LNCS 1009, 1995. ⊕.

[956] U. Wolter, R. Wessäly, M. Klar, and F. Cornelius. Four Institutions — A Unified Presentation of Logical Systems for Specification. Bericht Nr. 94-24, Technische Universität Berlin, Fachbereich Informatik, 1994. ⊕.

[957] D. Wolz. Tool Design for Structuring Mechanisms for Algebraic Specification Languages with Initial Semantics. In *Recent Trends in Data Type Specification, Proc. 11th Workshop on Specification of Abstract Data Types, Oslo, Norway, September 1995, Selected Papers*, Lecture Notes in Computer Science, pages 536–550. Springer–Verlag, 1996. To appear. ⊕.

[958] H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17:3–21, 1994.

[959] A. Zendler and S. Gastinger. *Vergleichende Analyse von Werkzeugen zum Aufbau und Einsatz von Bibliotheken für wiederverwendbare Software-Dokumente*, volume 2 of *Reihe Softwaretechnik, FAST*. Tectum Verlag, Marburg, 1995.

[960] A. Zendler and S. Gastinger. Werkzeuge zum Aufbau und Einsatz von Bibliotheken zur Software-Wiederverwendung: Kriterien und Anforderungen. In F. Huber-Wäschle, H. Schauer, and P. Widmayer, editors, *GISI 95, Erste gemeinsame Jahrestagung der Gesellschaft für Informatik (GI) und der Schweizer Informatiker Gesellschaft (SI), Zürich, 1995*. Springer, 1995.

[961] E. Zuca. From Static to Dynamic Abstract Data-Types: an Institution Transformation. Technical Report DISI-TR-1996-1, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1996. Submitted for publication, ⊕.

[962] E. Zucca. Implementation of Data Structures in an Imperative Framework. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications. 10th Workshop on Specification of Absstract Data Types*, pages 483–498. Springer LNCS 906, 1995. ⊕.