KORSO Reference Languages
Concepts and Application Domains

H.-D. Ehrich (ed.)

Abstract. This paper gives an overview of the three KORSO reference languages SPECTRUM, TROLL *light*, and SPECIAL, exposing their motivation and background, language concepts, and typical application domains. The presentation of the different languages is followed by a discussion to what extent these languages may complement each other in the software development process.

1 Introduction

H.-D. Ehrich

Among the KORSO project partners, there are quite a variety of specification and programming languages in use. In order to facilitate cooperation and communication, and to focus on concepts, unnecessary variety has to be avoided. However, no single approach was felt to be able to capture all requirements in all application areas. In order to span the range of approaches around, three conceptually different specification languages were selected and given the status of *reference languages* within the KORSO project.

These specification languages are the functional language SPECTRUM (TU Munich), the object-oriented language TROLL *light* (TU Braunschweig), and the process language SPECIAL (U Oldenburg). As an implementation backend to SPECTRUM, the functional programming language OPAL (TU Berlin) was also given the status of a KORSO reference language.

SPECTRUM (TU Munich) is the only language that has been newly developed within the KORSO project. It is based on axiomatic algebraic specification. It integrates approved functional programming concepts like functions as objects, non-strict functions, parametric polymorphism, and sort classes. The main application area is to develop correct complex algorithms, employing functional programs for expressing the algorithms. In a sense, SPECTRUM plays a central role in the KORSO project since it serves as a representative for the majority of algebraic approaches in the project. The SPECTRUM program development process is carried out to the level of functional programming where OPAL is supposed to take over.

TROLL *light* (TU Braunschweig) is a dialect of TROLL, a broad-spectrum object-oriented specification language for information systems under development at TU Braunschweig. The approach integrates the traditions of data modeling, algebraic data specification and process modeling with structuring principles of object-orientation. The TROLL *light* dialect is specifically designed for studying certification issues in information systems design and development.

SPECIAL (U Oldenburg) aims at developing applications in which communication and distribution aspects play a central role. The target language of program development is Occam. The approach offers several language levels, ranging from time diagrams close to intuition via MCTL, a variant of temporal logic, to explicit specifications in trace logic. There are mathematically well founded transformations between these levels.

The KORSO reference languages cover a wide spectrum of applications. However, an obvious question is how to use the languages pragmatically: for which application domains should which language be used? Are the languages well suited for their respective application domains? Are there application domains not well supported by any of these languages? Is it feasible to use more than one language in one application? If so, how can partial specifications with different languages be integrated?

Questions like these are discussed in chapter 5. It goes without saying that not everything is clear, further investigation is needed.

Of course, languages cannot be judged without a thorough discussion of a methodology for using them, and not without going through many case studies either. These issues are addressed in companion KORSO reports [Men+93, Wir+92, CHL94, LL94].

2 SPECTRUM

F. Regensburger, W. Grieskamp, C. Gerke

2.1 Motivation and background

The specification language SPECTRUM is one of the KORSO reference languages. Its development is part of the Technical University of Munich's contribution to the KORSO Project. SPECTRUM is an axiomatic specification language. Its design was influenced by experiences that the supervisor of the KORSO project, Manfred Broy, gained in earlier projects such as CIP [BBB+85] and PROSPECTRA [KBH91]. SPECTRUM is intended for use in the process of specification and development of software and hardware systems.

In computer science there are two main trends concerning formal methods and techniques used in the process of specification and development. The first is the algebraic school, surveyed in [Wir90]. The second is the family of type-theoretic languages, represented for example by the systems for the non–constructive type theories, HOL [Cam89] and LCF [WGM79, Pau87], and for the constructive type theories, Nuprl [C+86] and LEGO [LPT89]. Both families have a long tradition which reaches back to the beginning of this century, and all formal methods for specification and development are based on them to some extent. We will discuss their influence on SPECTRUM more thoroughly in the next section.

SPECTRUM is a language which permits – within reasonable limits – the description on the syntactic level of as many as possible of a data type's intended properties. The properties that can only be described using model theoretic techniques should be minimized. This shift towards the syntactic level emphasizes the role of logic as an instrument in the development process. As a consequence, the required logic must have at least the expressiveness of full first-order logic with induction principles. It should for example be possible to describe not only abstract data types and their corresponding functions, but also to some extent relations between specifications. A method which makes use of this feature and which is a special instance of the KORSO methodology is described in [Bro91].

Describing as much as possible on the syntactic level has the advantage that most of the reasoning required for establishing relations between specifications can be done as a pure calculation in the logical calculus associated with the specification language. And this verification in the calculus can be supported by a machine, a fundamental prerequisite for the development of realistic applications.

Correct software has to be feasible as far as executability and efficiency are concerned. One possible target implementation language for SPECTRUM is a functional language such as OPAL. OPAL is the KORSO reference language for functional implementations. It has been developed independently from the KORSO Project at the Technical University of Berlin under the supervision of Peter Pepper. OPAL is influenced by the same historical roots as SPECTRUM, but emphasizes the aspects of implementation design and highly efficient compilation. In this paper we will concentrate on SPECTRUM while outlining concepts and applications of OPAL only in so far as it is a target language of SPECTRUM-based software developments.

2.2 Language concepts

It became apparent that in order to get a specification language that fits our purposes, concepts from algebraic languages and type-theoretic languages had to be combined. Roughly speaking, SPECTRUM is an association between an algebra-oriented axiomatic language like Pannda-S, developed in the project PROSPECTRA [KBH91], and a more type theory oriented language like LCF [WGM79, Pau87]. An informal presentation with many examples illustrating its properties is given in [BFG+93a, BFG+93b]. A description of the formal semantics of its kernel language can be found in [GR93]. We now briefly summarize SPECTRUM's principal characteristics.

Influences from algebra in SPECTRUM. In SPECTRUM specifications the influence of algebraic techniques is evident. Every specification consists of a signature and an axioms part. However, in contrast to most algebraic specification languages, the semantics of a specification in SPECTRUM is loose, i.e. it is not restricted to initial or even term generated models. Moreover, SPECTRUM is not restricted to equational or conditional equational axioms, since it does not primarily aim at executable specifications. One can use full first order predicate logic to write very abstract and non-executable specifications, or only use its constructive part to write specifications which can be understood as programs.

Loose semantics leaves a large degree of freedom for later implementations. In special cases it allows a simple definition of refinement as the reduction of the class of models. This reduction is achieved by imposing new axioms which result from design decisions occurring in the step-by-step development of the algorithms. The well-known technique of relating specifications and their models with different signatures via signature morphisms and reducts in an institution framework provides a general means for refinement of data and algorithms.

Since writing well-structured specifications is one of our main goals, a flexible language for structuring specifications has been designed for SPECTRUM. This structuring is achieved by using so-called specification building operators which map a list of argument specifications into a result specification. The language for these operators was originally inspired by ASL [SW83]. The current version also borrows concepts from Haskell [HJW92], LARCH and PLUSS [Gau86].

Influences from type theory in SPECTRUM. The influence from type theory is twofold. On the type level SPECTRUM uses shallow predicative polymorphism with type classes in the style of Isabelle [Nip91, NP93]. The theory of type classes was introduced by Wadler and Blott [WB89] and originally realized in the functional programming language Haskell. Type classes may be used in SPECTRUM to model both overloading [CW85, Str67] as well as many instances of parameterized specifications. As in object-oriented languages type classes can be organized in hierarchies such that every class inherits properties from its parent classes. This gives SPECTRUM an object-oriented flavour.

The other influence of type theory can be seen in the language of terms and their underlying semantics. SPECTRUM incorporates the entire notation for typed λ-terms. The definition of the semantics and the proof system was strongly influenced by LCF. Thus, SPECTRUM supports a notion for partial and non-strict functions as well as higher-order functions in the sense of domain theory. The models of SPECTRUM specifications are assumed to be certain continuous algebras. All the statements about the expressiveness of LCF due to its foundation in domain theory apply to SPECTRUM.

In addition to type classes there are another two features in the SPECTRUM logic which distinguish SPECTRUM from LCF. SPECTRUM uses three-valued logic and also allows in a restricted form the use of non-continuous functions for specification purposes. These non-continuous functions are an extension of predicates and allow to express facts in a functional style that would otherwise have to be coded as relations. The practical usefulness of these features is investigated in case studies.

Functional sublanguages and OPAL. SPECTRUM already provides the entire notation for typed λ-terms, and thus contains the base of any functional language as a special case. However, pure functional languages like HOPE, ML, HASKELL and OPAL [Per88, HMM86, HJW92, Gro91, DFG+94] differ with respect to the syntactic sugar around the λ-calculus, the type system used, and the concepts for structuring in the large. Nevertheless, due to its generality SPECTRUM is powerful enough to cope with any of these languages as a target implementation language.

For example, the distinguishing concepts of OPAL are on the level of structuring in the large and in the type system. OPAL does not provide polymorphic types, but instead an enhanced concept of parameterization. The parameterization concept is inherited from traditional algebraic specification languages such as CIP-L or ACT ONE, and is augmented by the possibility of letting signature morphisms for actualizations be automatically inferred, much as type instantiations are inferred in languages with polymorphic type systems. The parameterization concept of OPAL can be modelled in SPECTRUM by its own parameterization concept. The effect of automatic inference of actualizations can be partially modelled by the use of type classes.

On the level of structuring in the large, OPAL emphasizes a discipline oriented towards the implementation of already established design specifications of large software systems. In OPAL a software system is a collection of structures which are connected by persistent import relations, such that each structure can be implemented independent of its application context. A structure itself is split into two parts: a signature part provides the external visible view, and the implementation provides the hidden internal view. The semantic relation between internal and external view is established by a hardwired forget-restrict-identify scheme, as described for example in [Wir90]. This structuring discipline can be modelled in SPECTRUM by its ASL-like specification building operations.

A more detailed discussion of OPAL as a functional sublanguage of SPECTRUM can be found in another contribution to this volume.

2.3 Application domains

SPECTRUM is a general-purpose specification language not streamlined for any special domain of application. It can be used for the specification of basic data types, functional programs, database systems or distributed systems. On an appropriate level of abstraction SPECTRUM can be applied in all of the above examples as a specification language, provided its basic theory is augmented by suitable theories for the particular issue at hand.

Basic Data Types. Several case studies show the adequacy of SPECTRUM for the development of basic data types. In the case study [SHNR93] the step-by-step development of a functional program for the data type of AVL-trees is shown. The development starts with a very abstract requirement specification of AVL-trees which takes advantage of the full first-order predicative logic supplied by SPECTRUM. This specification is then refined in several steps to the well-known algorithmic realisation of AVL-trees [Wir76]. A purely syntactic translation yields a program in the functional language SML.

Functional Programs. Due to its syntax and semantics SPECTRUM is best suited for the specification of purely functional software without implicit state, using a functional language such as OPAL as the implementation language. For example, in the LEX case study an efficient generic lexical scanner is formally developed [BDDG93]. The starting point is a requirement specification which resembles an informal description of the scanner as it is provided by the manual pages of the operating system UNIX. This requirements specification is formally transformed into a more abstract one which recovers the underlying problem domain more clearly. The resulting requirements are then transformed into a design specification, which is later on developed into an executable functional specification. A syntactic translation yields a program in the functional language OPAL. Execution benchmarks show, that this program is only 7-9 times slower than the quite elaborated generic scanners available under UNIX.

Distributed Systems. SPECTRUM may be augmented by a theory of streams and stream-processing functions, serving then as a convenient base for the modelling of distributed systems [Bro88]. In this framework a distributed system is modelled as a network of stream-processing functions. This technique was successfully used in the HDMS-A case study [Nic94] which shows that stateless languages like SPECTRUM with higher-order concepts are not restricted to applications of a purely sequential nature. The connection between this kind of modelling distributed or concurrent systems and other description formalisms like SPECIAL (described in a later chapter of this report) and Petri nets [Rei85] is still a subject of current research.

Database Systems and Semi-Formal Methods. Case studies in the context of the HDMS-A project also show how SPECTRUM can be used to model problems in the area of database systems [Het93] or more generally how SPECTRUM can serve as a basis for the formalisation of semi-formal methods like SSADM [DCC92, Hus93].

Tools for the formal development of software using SPECTRUM are still under development. Besides a language analyser for SPECTRUM, prototypical machine support for the verification task has been implemented at the Technical University of Munich. There is an instance of the SPECTRUM logic [Reg94] in the generic theorem prover Isabelle [Nip89] and a backend for the SPECTRUM parser [Pus94] that translates SPECTRUM specifications into Isabelle theories.

For OPAL, a comprehensive compilation system which produces highly efficient and portable code has been developed at the Technical University of Berlin. The intention is to integrate this system as a backend in an overall development system based on SPECTRUM and the KORSO methodology.

3 TROLL *light*

R. Herzig, M. Gogolla, G. Denker

3.1 Motivation and background

An information system can be understood as a model of some slice of reality. Such a model must be able to capture the things which exist as well as the things which may happen.

Information system development can be roughly split up into two important phases. The aim of the requirements engineering phase is to obtain a syntactical description of a conceptual model of the information system in mind. This description, often called a conceptual schema, is used as input for the design engineering phase in which a running system is built under consideration of further nonfunctional constraints [Lou92].

In traditional conceptual modeling structural properties of a desired system are described in terms of some kind of semantic data model [HK87, PM88] (e.g. the Entity-Relationship (ER) model [Che76]). The schema of a semantic data model consists of a set of object types with associated attributes and concepts expressing special relationships among object types. By such a schema the state space of an information system is fixed. In many semantic data models the state space of a system can be further restricted by static integrity constraints which may be formulated in a predicate logic style.

Semantic data models do not attempt to model system behavior. In fact behavioral aspects are very often considered first on implementation level when writing application programs. In an improved approach as reported in [EGH+92] dynamic aspects are made subject to conceptual modeling by adding an evolution layer on top of an object layer established by an extended ER model. In this layer possible state evolutions of a system can be restricted by means of dynamic integrity constraints expressible by temporal logic formulae [Eme90, MP92], and possible state transitions can be described by event modeling. Thereby dynamic integrity constraints are particularly suited to express everything that is forbidden while event modeling is useful to make explicit what is allowed.

In event modeling it is often convenient to connect events with event preconditions. Event preconditions may refer to the current state of a system as well as to the present state evolution. Hence predicate logic and past tense temporal logic may be used to formulate event preconditions. Another aspect of event modeling is the specification of behavior patterns, e.g. the specification of allowed event sequences or event alternatives. This task may be supported by process describing formalisms like process algebras [Hen88, BW90] or Petri nets [Rei85].

In the approach of [EGH+92] static and dynamic properties are described in different layers. The separate description of structural and dynamic aspects leads to the case that aspects which determine object behavior are dispersed over many global events which means to be a serious impediment for schema modularization. This can be solved by an integrated description of static and dynamic properties in the framework of objects [SSE87, FSMS92, EDS93]. Semantic foundations for this approach have been and are being studied in the ESPRIT BRA IS-CORE [ES91].

At the beginning of the KORSO project we looked for an *object-oriented specification language* meeting the following requirements. On the one hand it had to be simple enough to make the study of verification and validation aspects feasible, on the other hand it had to be expressive enough to carry out some nontrivial case studies. Examples of languages for object-oriented conceptual schema specification we had to consider were Oblog [SSG+91], Cmsl [Wie91] and Troll [JSHS91].

Since we did not want to re-invent the wheel, we took the specification language Troll as a basis for our language. Troll is a voluminous language which offers a large variety of modeling concepts to satisfy the user needs for expressing real world facts and laws. With this aim in mind it also offers some partially redundant concepts. Thus it was necessary to evaluate which concepts are redundant and could be disregarded for our language purposes. Other concepts, like temporal logic for specifying dynamic constraints, were excluded for pragmatic reasons (so as to simplify prototyping and to make verification manageable). Finally we obtained a language with a small number of basic concepts and hence called it Troll *light* [CGH92].

3.2 Language concepts

Troll *light* is a language for formulating object descriptions called *templates* having the following general structure.

`TEMPLATE` <*name of an object description*>	
`DATA TYPES`	import of data types
`TEMPLATES`	import of other object descriptions
`SUBOBJECTS`	declaration of local subobjects
`ATTRIBUTES`	declaration of attributes
`EVENTS`	declaration of event generators
`CONSTRAINTS`	static integrity conditions (invariants)
`DERIVATION`	rules for derived attributes
`VALUATION`	effects of events on attributes
`INTERACTION`	synchronization of events in different objects
`BEHAVIOR`	restriction of possible event sequences
`END TEMPLATE`	

A template describes static and dynamic properties, i.e. valid states and valid state evolutions, of a prototypical object. In an actual object community this prototypical object may be instantiated by any number of actual objects.

In principle, the attribute state of an object is given by a number of data values. In contrast to objects as instances of templates, values as instances of data types are considered as stateless items. A preferred approach for the implementation-independent description of abstract data types is algebraic specification [EM85, EGL89, Wir90].

Specifications of standard and user-defined data types can be referred in the *data types* clause of a template. An object description may also depend on other object descriptions. These can be imported in the *templates* clause of a template.

The declaration of local subobjects, attributes and event generators describes the interface of objects to other objects. By subobject relationships objects are arranged into an object hierarchy, attributes hold the directly observable state information of an object, and events are abstractions of state-modifying operations on objects. Subobject, attribute and event symbols define a signature over which axioms concerning static and dynamic properties can be specified.

With respect to static properties we have to mention constraints and derived attributes. Constraints are static integrity conditions (invariants) restricting the possible states of an object to admissible ones. In the same direction the contents of a derived attribute is determined by derivation rules. However, while the values of non-derived attributes must be explicitly set in consequence of event occurrences, the values of derived attributes are computed indirectly from other stored or derived information. For stating static integrity conditions and derivation rules TROLL *light* includes a SQL-like query calculus.

The query calculus is also needed in the formulation of dynamic properties. For instance, the effect of event occurrences on attributes is specified by valuation rules, events in different objects can be synchronized by interaction rules, and possible event sequences can be restricted to admissible ones by simple process descriptions which may include event preconditions formulated as expressions in the query calculus.

Templates can be used to describe simple objects as well as composite objects having simple objects as subobjects [HCG94]. In order to install an actual object community a certain template must be designated as the schema template and a certain object belonging to the schema template must be chosen as the schema object.

The schema object induces a tree of object identifiers building the basic structure on which the evolution of an object community can take place. In a certain state of an object community each object identifier is associated with an attribute state consisting of a complex value and a behavior state. State transitions are induced by finite sets of simultaneously occurring events in different objects. These sets have to be closed against interaction rules. Attribute states are altered according to valuation rules and behavior states are changed according to process descriptions. Hence an object community is a system of concurrently existing and interacting objects.

A first sketch of an operational semantics of TROLL *light* appeared in [GCH93]. A semantics based on graph grammars is being under consideration.

3.3 Application domains

The language TROLL *light* combines several well-known techniques to achieve conceptual modeling of information systems.

- Algebraic specifications can be included by importing data types.
- Ideas of semantic data models are adopted by providing objects with complex attributes.
- Predicate logic plays the central role of the SQL-like query calculus.
- Principles of process modeling had their influence on the specification of objects (objects are processes) and behavior patterns.

In principle TROLL *light* might be used whenever object communities of concurrently existing and interacting objects are to be described. The general method is as follows: Identify objects connected with states and place them into an object hierarchy. Describe attributes, events as state-modifying operations, and the effects of events on attributes. Restrict possible attribute states by constraints and possible event sequences by behavior patterns. Synchronize events in different objects by interaction rules.

The result is a fully formalized description of the system which is to be designed. With respect to the methodical framework for software development in KORSO [Wir+92] this corresponds to a formal requirements specification. Such a specification should be analyzed with respect to certain properties such as unambiguity, consistency and completeness. Another aspect is validation which means to test a formal specification against the informal user requirements.

Schema analysis and validation should be supported by corresponding software tools. In this respect we designed and implemented a proof support system for verifying formally described propositions about specifications, and an animation system allowing rapid prototyping of specifications (see [GCD+94] in this volume).

Because TROLL *light* is a language for the specification of information systems and databases it is only natural to regard database systems or programming languages which allow for persistence as candidates for the implementation basis. Currently we are working with an object-oriented database system but it is also imaginable to employ a relational database system together with an appropriate embedding into a modern programming language. The steps which lead from the TROLL *light* specifications to executable programs can surely be done in a systematic way, but one cannot expect a complete automatic transformation because there are still many design decisions which are open to debate. The general role of TROLL *light* in a development environment for information systems is discussed in [VHG+93].

There exist some case studies for TROLL *light* ranging from the specification of a simple database system [Con93] through the specification of a dictionary for TROLL *light* [Her93] object descriptions to the specification of a production cell [HV94] and some other examples being all collected in [Ehr93].

For instance, the specification of a production cell showed that, although a production cell is not a typical information system, basic structural and dynamic aspects of such a cell could be described by TROLL *light* in a straightforward manner. However, other special aspects, including the description of activity, liveness conditions or real-time conditions, could not. Here it comes to the fore that TROLL *light* was developed for the specification of information systems as typical interactive systems. Hence we claim that TROLL *light* might be used whenever static and dynamic aspects of complex-structured interactive systems are to be described.

The specification of such systems may be supported by more expressive language features as provided by TROLL [JSHS91], for instance by conceptually different object types, temporal logic, more expressive process specifications.

4 SPECIAL

E.-R. Olderog

4.1 Motivation and background

The purpose of a reactive system is to continuously react to stimuli from its environment so that this is kept in a certain desirable condition. To this end, the system may communicate with its environment via input and output channels. An adequate specification of reactive components within software systems requires a formalism incorporating such concepts like channel, communication, trace, environment, synchronization and concurrency. These concepts cannot easily be embedded in specification formalisms that are oriented towards sequential programming like SPECTRUM or towards an object-oriented view like TROLL *light*. Therefore a more specific specification formalism is needed to deal with reactive system components.

The KORSO team at Oldenburg aims at developing an integrated specification formalism that satisfies the following criteria:

(1) it should be intuitively understandable for a user,
(2) it should support verification of implementations by automatic procedures (model checking) or – where this is impossible or too inefficient – by interactive tactical reasoning based on compositional verification rules,
(3) it should – at least for subsets – allow a systematic or even automatic synthesis of correct implementations via transformation rules.

To date there is no single specification formalism for reactive systems satisfying all these criteria. Rather a variety of specification formalisms have been developed, among them Temporal Logic [MP92], iterative programs like action systems [Bac90] or UNITY programs [CM88], input/output automata [LT89], high-level Petri nets, process algebras [Mil89, BW90] with realistic extensions like LOTOS, and stream processing functions [Bro87].

Therefore the KORSO team at Oldenburg uses a specification language for reactive systems, called SPECIAL, that comprises three language levels describing reactive systems at different levels of abstraction. The target implementation language is fixed as an OCCAM-like programming language [IN88] where the data types are kept as simple as possible.

The language levels of SPECIAL are based on previous work within the ESPRIT projects 415 on "Parallel Architectures and Languages for Advanced Information Processing – a VLSI-directed approach" [Bak89] and BRA 3104 on "Provably Correct Systems" (ProCoS) [BLH93]. They are developed further within the current ESPRIT projects FORMAT and ProCoS II.

The aim within KORSO was to integrate these language levels. To this end, semantic links between these different levels have been defined and automatic procedures for achieving some of these links have been implemented. These implementations are part of the TRAVERDI system [BH94]. First case studies with

SPECIAL have been carried out (see section 4.3) but for a final assessment of its concepts more experience is needed.

4.2 Language concepts

The three language levels of SPECIAL are timing diagrams, temporal logic and trace logic.

Symbolic Timing Diagrams. At the top-level the graphical specification language of *timing diagrams* [DS93, SH93] is used. Timing diagrams represent an informal specification style which is well understood by hardware designers.

Since in KORSO we are not interested in low level hardware design but rather in a system level view where components interact via OCCAM-style message passing, we use timing diagrams *symbolically* for expressing qualitative temporal constraints between events in the sense of partial orders rather than quantitative timing constraints. This approach enables us to connect timing diagrams with the well understood semantic world of temporal logic specifications [MP92].

In SPECIAL a timing diagram consists of an interface of channel and variable declarations and of a partial order on events. Each event represents a state change of an interface channel or variable. States are described by logical formulas. To specify temporal dependencies between different events, directed arrows are used. A distinction between two types of arrows is made: *weak arrows* express safety constraints and *strong arrows* express liveness constraints.

Timing diagrams are given a formal automata-based semantics and an equivalent one via a translation into Temporal logic (MCTL). The details of the syntactic and semantic definition of timing diagrams have evolved through a number of small case studies.

Temporal Logic. At an intermediate level a branching time temporal logic called MCTL standing for *Modular Computation Tree Logic* is used. MCTL is a logical calculus developed by Josko [Jos87] as an extension of CTL [CES86] to deal with reactive systems.

"Modular" refers to an assumption/commitment style logic: each MCTL specification consists of an *assumption part* dealing with the environment and a *commitment part* dealing with the desired system reactions. The assumption part is restricted to a subset of linear time temporal logic whereas the commitment part allows arbitrary positive CTL formulas. CTL formulas describe the full branching structure of computations arising from nondeterministic choices and consist of subformulas dealing with the computation paths and the states on such paths. Optional is an additional *fairness part* in MCTL specifications.

The advantages of MCTL are threefold: it has a well-defined semantics in terms of Kripke structures, an efficient model checking procedure for finite data types, and compositional proof rules for verifying concurrent implementations without running into the problem of *state explosion* [Jos89]. Based on a Kripke structure semantics of OCCAM, the model checking procedure can be used to automatically verify MCTL properties of OCCAM programs.

Moreover, in [DS93] a translation of timing diagrams into a subset of MCTL is given which provides a well-defined temporal logic semantics to the graphical specification language of timing diagrams. Within KORSO both the translation of timing diagrams into MCTL and the model checking procedure had to be adapted to the setting of synchronous communication as present in OCCAM.

Trace Logic. At a level close to the OCCAM programming paradigm we use a specification language SL that combines ideas from trace logic [Zwi89, Old91a], process algebra and assertional specification methods. SL was developed in the ESPRIT project ProCoS [ORSS92, BLH93].

The idea of SL is to split the description of the desired system behaviour into a *trace part* and a *state part*. The trace part specifies the sequencing constraints on the system channels but ignores the communication values. This is done by means of regular expressions over a communication alphabet. The state part specifies the communication values by means of local state variables and a pre-post style assertional specification of the state transition that each communication invokes. Altogether SL can be seen as extending Z specifications [Spi89], UNITY programs [CM88] or action systems [Bac90] by explicit communications and regular expressions to control their occurrence.

SL has a well-defined *predicative semantics* based on a combined *state-trace-readiness model* for reactive systems [ORSS92]. A strong point about SL is that it is well suited as a departure point for a transformational design of sequential and concurrent OCCAM-like programs. In fact, a large subset of SL can be transformed in a fully automatic way into sequential programs.

Within KORSO we have defined a semantic link of SL with MCTL and hence timing diagrams.

4.3 Application domains

The language SPECIAL is designed for the specification of reactive components within software systems. It comprises of three language levels.

The envisaged use of these three language levels is that a user starts to specify the requirements as a set of timing diagrams. As a graphical specification language timing diagrams are thought to be *intuitively understandable*, for example to users with an engineering background. Timing diagrams are then automatically translated into the temporal logic MCTL. Given an MCTL specification one can proceed in two ways: either come up with an OCCAM program thought to implement that specification and *verify* this implementation relation using compositional verification rules and model checking [Hun93], or come up with a lower level SL specification which can be proven correct against the MCTL specification. SL is close to the programming level and allows a systematic *transformational design* of OCCAM-like programs from SL specifications [OR93]. The TRAVERDI system [BH94] supports this dual transformation and verification approach to the design of distributed reactive systems.

Typical applications for the above formalism are the specification of communication protocols or the design of control programs – both sequential and concurrent ones. Within Korso the case study of a communication processor has been performed [Kle94, HR93]. Also a specification and transformational design of a part of the object communication mechanism in the Korso case study Hdms-A has been achieved [Sch94].

5 Discussion

H.-D. Ehrich (moderator)

There are many perspectives along which specification languages can be evaluated and compared: underlying concepts and paradigms, expressive power, syntax, semantics, the way these are described, ease of use, scopes of application, etc. Moreover, a language in isolation is hardly enough to judge its usefulness, be it as promising as it may be. An appropriate set of tools, an adequate methodology and some experience in using it are essential. The latter can be given by appropriate case studies.

All these aspects of the Korso reference languages, and others, have been discussed during their development, on general meetings, on special workshops, on working meetings within groups and among groups, and via long-distance means of communication. Methodology, tools and case studies were addressed by separate Korso working groups, cf. [Men+93, Wir+92, LL94, CHL94].

The main focus of the discussion was on correctness issues, especially formal and machine supported verification. However, other goals like usability, functionality, reliability, security, adaptability, etc., were also discussed at times.

In the initial phase of the project, the role of the Korso reference languages within the project was subject to lively discussion. There was some agreement that the purpose of the reference languages was to support communication among partners without restricting them too much in their work. Also, the reference languages were supposed to serve as an integration kernel for the various partner activities. However, a single language was not felt enough to cover the concepts and paradigms represented in the project. Finally, the partners agreed on the languages described in this report.

Appropriate modelling, structuring and description concepts are essential for avoiding errors as much as possible and to localize the errors which inevitably remain. In particular, precise syntactic and semantic descriptions played a major role in the discussion, due to the emphasis on correctness issues.

A comprehensive discussion of all these aspects is beyond the scope – and page limit – of this report. Rather, the purpose of the current section is to give some arguments about the pragmatic roles of the Korso reference languages: For which application domains should which language be used? Are the languages well suited for their respective application domains? Are there application domains not well supported by any of these languages? Is it feasible to use more than one language in one application? If so, how can partial specifications with different languages be integrated?

Looking at SPECTRUM and OPAL as one complex, we have three views of system design represented among the KORSO reference languages: the functional view of SPECTRUM and OPAL, the object-oriented view of TROLL *light*, and the reactive view of SPECIAL. The target application areas towards which the languages were designed are (functional) programs, information systems, and reactive systems, respectively.

Roughly speaking, the three approaches are rooted in the traditions of program specification and transformation, conceptual data modeling and database design, and concurrency, respectively. However, the separation is by no means clearcut: SPECTRUM has concepts to cope with concurrency, and there are activities to explore its usefulness in the field of entity-relationship modeling. TROLL *light* draws from the tradition of the latter, but goes beyond classical conceptual data modeling by incorporating dynamic aspects, and it also inherits quite a few ideas from abstract data type theory. SPECIAL is the most specialized of these languages: it concentrates on concurrency issues and distributed systems where it is more powerful than the other languages.

Looking at the main emphasis of the approaches, one might say that SPECTRUM integrates functional program design with concurrency, TROLL *light* integrates conceptual data modeling with concurrency, and SPECIAL specializes on concurrency.

Programming paradigms and underlying logics offer other perspectives for comparison. SPECTRUM is a functional language based on data abstraction and employing higher-order predicate logic. TROLL *light*, being object-oriented, adopts the imperative paradigm, i.e., it is state-oriented. For describing system dynamics, TROLL *light* employs a CSP-like way of process description. SPECIAL uses other means of describing concurrent processes, namely timing diagrams for the higher abstraction level, a variant of temporal logic for an intermediate abstraction level, and trace logic for the lower abstraction level.

Still another comparison perspective is the implementation platform envisaged. SPECTRUM specifications can be transformed to OPAL programs which in turn translate to efficient C-code. The typical implementation platform for TROLL *light* specifications are (object-oriented) database systems with application programs for the dynamic parts. SPECIAL is designed for using OCCAM as its operational basis.

At the beginning of the project, OPAL was the only reference language which was ready to use. The other three, i.e., the high-level specification languages, were either new developments within the KORSO project or major adaptations from existing prototypes. Accordingly, language design and implementation activities took much time, leaving little time for investigating pragmatic application aspects. The consequence was that the reference languages could not be used from the beginning in case studies, and that reference language concepts entered the methodology discussion only after a while.

This point raised considerable discussion, namely that SPECTRUM, TROLL *light* and SPECIAL are still in a preliminary status. Especially the formal semantics of the languages are not yet fully documented. Consequently, detailed analysis and comparison of the languages' semantics could not be performed yet. On the other hand, there is a variety of languages developed and used by KORSO partners which are in a more mature status concerning formal semantics, tool support and case studies, whose concepts, though, may be not as advanced as those of the reference languages. These topics, however, are highly important for correct software development and hence for KORSO.

Other algebraic specification languages that have been used by KORSO partners and that were mentioned or discussed at times are ACT ONE [CEW93, Cla92, EM85], OBSCURE [LL87], and ASL with the RAP system [Hus85]. Corresponding developments outside KORSO were also given due attention, e.g., ASSPEGIQUE with the PLUSS language [BC85], OBJ [GCG92] and FOOPS [GM87, RS92].

Returning to the KORSO reference languages, there was some discussion on whether it is feasible to use several specification languages based on different paradigms for real systems specification, addressing each partial aspect with the most appropriate concepts. This way of "taking the best from several worlds" might look attractive for specifiers, but there is a problem: how can overlapping system properties be proved if they are not formulated in one language, i.e., they are not tractable by one proof system? Research is needed whether and how proofs across paradigms can be developed from partial proofs within paradigms.

The above argument might not bother practitioners too much since, in many cases if not most, formal correctness proofs will not be feasible, too expensive or not necessary for the reliability and security standard envisaged. In many such cases, semiformal techniques can do a good job. In fact, semiformal proofs – in about the sense as mathematics itself is semiformal – can achieve a high level of correctness.

However, semiformal techniques are largely guided by intuition and cannot be better than the intuition of the best experts in the field. Semiformal techniques tend to be more reliable in cases where it is known how to do *in principle* a strictly formal proof, because such knowledge helps to develop intuition. Moreover, while the amount of work using a semiformal technique might be much less than that needed for applying a formal technique, the amount of *human* work (handwork or, rather, headwork) might be more for the former, depending on the availability of machine support for the techniques in question.

6 Conclusion

There is some confidence that the KORSO reference languages cover the possible application domains to a large extent. However, it would be interesting to receive reactions from application areas which do not feel well represented. Possible such areas are real-time systems and embedded systems – they were not represented in the KORSO project.

It was a highly rewarding experience to cooperate on the diverse approaches to formal specification and verification represented in the project, to see the languages develop and to discuss their many facets and interrelationships.

At the end of a successful research project with new insights and results, there use to be more questions open than at the beginning. This project is no exception. Perhaps the most urgent need is to make the approaches under consideration more practical, finishing and consolidating the language design and implementations, integrating more tools, designing appropriate methodologies, performing ever larger case studies, and give the users more guidance on how to use the systems. But also the impact on basic research is obvious. Especially the interplay between multiple paradigms and between formal and semiformal techniques need further study before they can be safely put to practice.

Acknowledgements

The editor is grateful for the inspiring atmosphere in which work and discussion reported here took place. Thanks to all who contributed, by authoring or coauthoring a chapter or by participating actively in the discussion. Special thanks are due to Heinrich Hussmann who gave a survey of the reference languages on the KORSO Workshop in Munich in March 1993. Thanks, too, to Uwe Wolter and Hartmut Ehrig who gave critical comments on an earlier draft of this paper.

References

[Bac90] R.J.R. Back, Refinement Calculus, Part II: Parallel and Reactive Programs, in: [BRR89] 67-93.

[Bak89] J.W. de Bakker, Ed., Languages for Parallel Architectures – Design, Semantics, Implementation Aspects (Wiley & Sons, 1989).

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, and Krieg-Brückner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L.*, volume 183 of *L.N.C.S.* Springer, 1985.

[BC85] M. Bidoit and C. Choppy. *ASSPEGIQUE: An Integrated Environment for Algebraic Specifications.* Proc. 1st Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'85), pages 246–260. LNCS 186, Springer-Verlag, Berlin, 1985.

[BDDG93] 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, {kd,wg}@cs.tu-berlin.de, October 1993.

[BFG+93a] 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. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, Fakultät für Informatik, TUM, 80290 München, Germany, May 1993.

[BFG+93b] 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. Technical Report TUM-I9312, Technische Universität

München. Institut für Informatik, Fakultät für Informatik, TUM, 80290 München, Germany, May 1993.

[BG80] R. M. Burstall, J. A. Goguen The semantics of CLEAR, a specification language. In D. Bjørner, editor, *Proc. Advanced Course on Abstract Software Specification.* LNCS, Springer, 1980.

[BH94] J. Bohn, H. Hungar, TRAVERDI – Transformation and Verfication of Distributed Systems. This volume.

[BLH93] D. Bjørner, H. Langmaack, C.A.R. Hoare, Eds., Provably Correct Systems (Tech. Report, DTH Lyngby, 1993).

[Bro87] M. Broy, Specification and top-down design of distributed systems, J. Comput. System Sci. 34 (1987) 236-265.

[Bro88] M. Broy. Requirement and Design Specification for Distributed Systems. *LNCS*, 335:33–62, 1988.

[Bro91] M. Broy. Methodische Grundlagen der Programmierung. In M. Broy, editor, *Informatik und Mathematik*, pages 355–365. Springer-Verlag, 1991.

[BRR89] J.W. de Bakker, W.-P. de Roever, G. Rozenberg, Eds., Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, LNCS 430 (Springer-Verlag, 1990).

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[C+86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Cam89] J. Camilleri. The HOL System Description, Version 1 for HOL 88.1.10. Technical Report, Cambridge Research Center, 1989.

[CES86] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic Verification of finite-state concurrent systems using temporal logic specifications, ACM TOPLAS 8 (1986) 244-263.

[CEW93] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development — The ACT Approach.* World Scientific Publishing, AMAST Series in Computing, 1993, to appear.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[Che76] P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[CHL94] F. Cornelius, H. Hussmann, and M. Löwe. *Software Engineering with Formal Methods. A Medical Information System: The* KORSO *Case Study.* This volume.

[Cla88] I. Classen. Semantik der revidierten Version der algebraischen Spezifikationssprache ACT ONE. Technical Report 88/24, TU Berlin, 1988.

[Cla92] I. Claßen. *ACT System – User Manual.* Internal Report, TU Berlin, April 1992.

[CM88] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, 1988.

[Con93] S. Conrad. Spezifikation eines vereinfachten Datenbanksystems. In Ehrich [Ehr93].

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[DCC92] E. Downs, P. Clare, and I. Coe. *Structured systems analysis and design method (2nd ed).* Prentice-Hall, 1992.

[DFG+94] K. Didrich, A. Fett, C. Gerke, W. Grieskamp. P. Pepper. OPAL *: Design and Implementation of an Algebraic Programming Language*. accepted for Conference on Programming Languages and System Architecture

[DS93] W. Damm, R. Schlör, Specification and verification of system-level hardware designs using timing diagrams, in: Proc. European Design Automation Conference, Paris, 1993.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT'93: Theory and Practice of Software Development*, pages 453–467. Springer LNCS 668, 1993.

[EGH+92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual Modelling of Database Applications Using an Extended ER Model. *Data & Knowledge Engineering*, 9(2):157–204, 1992.

[EGL89] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen – Eine Einführung in die Theorie.* Teubner, Stuttgart, 1989.

[Ehr93] H.-D. Ehrich, editor. *Beiträge zu* KORSO*– und* TROLL *light–Fallstudien.* Informatik-Bericht 93–11, Technische Universität Braunschweig, 1993.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer, 1985.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 995–1072. North-Holland, Amsterdam, 1990.

[ES91] H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. In G. Saake and A. Sernadas, editors, *Information Systems — Correctness and Reusability, Proc. ESPRIT BRA IS-CORE Workshop, London*, pages 1–24. Informatik-Bericht 91–03, Technische Universität Braunschweig, 1991.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer. Principles of OBJ2. In *Proc. POPL*, 1985.

[FSMS92] J. Fiadeiro, C. Sernadas, T. Maibaum, and A. Sernadas. Describing and Structuring Objects for Conceptual Schema Development. In Loucopoulos and Zicari [LZ92], pages 117–138.

[Gau86] M.-C. Gaudel. Towards Structured Algebraic Specifications. *ESPRIT '85', Status Report of Continuing Work (North-Holland)*, pages 493–510, 1986.

[GCD+94] M. Gogolla, S. Conrad, G. Denker, R. Herzig, N. Vlachantonis, and H.-D. Ehrich. TROLL *light*: The Language and Its Development Environment. This volume.

[GCG92] J. Goguen, D. Coleman, and R. Gallimore. *Applications of Algebraic Specifications using OBJ.* Cambridge, 1992.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems DISCO*, pages 17–32. Springer LNCS 722, 1993.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical report, Digital, Systems Research Center, Paolo Alto, California, 1985.

[GM87] J.A. Goguen and J. Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics.* Research Directions in Object-Oriented Programming, B. Shriver, P. Wegner, (eds.), pages 417–477. MIT Press, 1987.

[GR93] R. Grosu and F. Regensburger. The logical framework of SPECTRUM. Interner Bericht, Technische Universität München, 1993.

[Gro91] OPAL Language Group. The Programming Language OPAL. Technical Report 91-10, Technische Universität Berlin, 1991.

[HCG94] 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).* Cepadues Society Press, France, 1994.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Her93] R. Herzig, *Spezifikation der abstrakten Syntax von TROLL light mit TROLL light.* In Ehrich [Ehr93], pages 43–49.

[Het93] R. Hettler. Übersetzung von E/R-Schemas nach SPECTRUM. Interner KORSO-Bericht, 1993.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2).* ACM SIGPLAN Notices, May 1992.

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[HMM86] R.W. Harper, D.B. MacQueen, and R.G. Milner. Standard ML. *Report ECS-LFCS-86-2, Univ. Edinburgh*, 1986.

[HR93] H. Hungar, G, Rünger, Verification of a communication network. A case study in the use of temporal logic, Manuscript, Oldenburg/Saarbrücken, 1993.

[Hun93] H. Hungar, Combining model checking and theorem proving to verify parallel processes, in: C. Courcoubetis, Ed., Computer Aided Verification (CAV '93), LNCS 697 (Springer-Verlag, 1993) 154–165.

[Hus85] H. Hussmann. *Rapid Propotyping for Algebraic Specifications — RAP-System User's Manual.* Tech. Report MIP–8504, Universität Passau 1985 (revised edition 1987).

[Hus93] H. Hussmann. Synergy between formal and pragmatic software engineering methods. Technical Report TUM-I9323, Institut für Informatik, Technische-Universität München, September 1993. (Submitted to ESOP 94).

[HV94] R. Herzig and N. Vlachantonis. *Specification of a Production Cell with TROLL light*, Case Study "Production Cell": A Comparative Study in Formal Specification and Verification (C. Lewerentz and T. Lindner, eds.), Chapter 14. FZI Publication 1/94, Forschungszentrum Informatik, Karlsruhe (Germany), 1994.

[IN88] INMOS Ltd., OCCAM 2 Reference Manual (Prentice Hall, 1988).

[Jos87] B. Josko, Modelchecking of CTL formulae under liveness assumptions, in: Proc. ICALP 87, LNCS 267 (Springer-Verlag, 1987) 280-289.

[Jos89] B. Josko, Verifying the correctness of AADL modules using model checking, in: [BRR89] 386-400.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91–04, Technische Universität Braunschweig, 1991.

[KBH91] B. Krieg-Brückner and B. Hoffmann, editors. *PROgram development by SPECification and TRAnsformation: Vol. I: Methodology, Vol II: Language Family, Vol III: System.* PROSPECTRA *Report M.1.1.S3-R-55.2, -56.2,h - 57.2.* (to appear in LNCS), 1991. Universität Bremen (1990).

[Kle94] S. Kleuker, Case study: stepwise development of a communication processor using trace logic, to appear in: D.J. Andrewa, J.F. Groote, C.A. Middelburg, Eds., Semantics of Specification Languages (SoSL), Workshops in Computing, Springer-Verlag, 1994.

[LL87] T. Lehmann and J. Loeckx. *The Specification Language of OBSCURE.* Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specification of Abstract Data Types, pages 131–153. LNCS 332, Springer-Verlag, Berlin 1987.

[LL94] C. Lewerentz and T. Lindner. *Case Study 'Production Cell': A Comparative Study in Formal Specification and Verification.* This volume.

[Lou92] P. Loucopoulos. Conceptual Modeling. In Loucopoulos and Zicari [LZ92], pages 1–26.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO. *Department of Computer Science, University of Edinburgh*, 1989.

[LT89] N.A. Lynch and M.R. Tuttle, An introduction to input/ouput automata, CWI-Quaterly 2(3), CWI, 1989.

[LZ92] P. Loucopoulos and R. Zicari, editors. *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development.* John Wiley & Sons, 1992.

[Men+93] W. Menzel et al. *Bestandsaufnahme und Klassifikation der* KORSO-*Werkzeuge.* Technischer Bericht, Universität Karlsruhe 1993.

[Mil89] R. Milner, Communication and Concurrency (Prentice-Hall, 1989).

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification.* Springer, 1992.

[Nic94] F. Nickl. Ablaufspezifikation durch Datenflußmodellierung und stromverarbeitende Funktionen. Technical Report, Institut für Informatik, Technische-Universität München, 1994.

[Nip89] T. Nipkow. Term rewriting and beyond — theorem proving in Isabelle. *FAC*, 1:320–338, 1989.

[Nip91] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[NP93] T. Nipkow and C. Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418, 1993.

[Old91a] E.-R. Olderog, Nets, Terms and Formulas (Cambridge University Press, 1991).

[OR93] E.-R. Olderog and S. Rössig, A case study in transformational design of concurrent systems, in: M.-C. Gaudel, J.-P. Jouannaud, Eds., Proc. TAPSOFT '93, LNCS 668 (Springer-Verlag, 1993) 90–104.

[ORSS92] E.-R. Olderog and S. Rössig and J. Sander and M. Schenke, ProCoS at Oldenburg: The Interface between Specification Language and OCCAM-like Programming Language, Bericht 3/92, Univ. Oldenburg, Fachbereich Informatik, 1992.

[Pau87] L.C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1987.

[Per88] N. Perry. Hope+. Internal report IC/FPRC/LANG/2.51/7, Dept. of Computing, Imperial College London, 1988.

[PM88] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, 1988.

[Pus94] C. Pusch. Übersetzung von SPECTRUM Spezifikationen nach Isabelle. Fortgeschrittenenpraktikum, Technische Universität München, to appear, 1994.

[Reg94] F. Regensburger. Der SPECTRUM Kalkül in Isabelle. Interner Bericht, Technische Universität München, to appear, 1994.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer, 1985.

[RS92] L. Rapanotti and A. Socorro. *Introducing FOOPS.* Technical Report, Programming Research Group, Oxford University, 1992.

[Sch94] M. Schulte, Spezifikation und Verifikation von kommunizierenden Objekten in einem verteilten System, Diplomarbeit, Fachbereich Informatik, Univ. Oldenburg, März 1994.

[SH93] R. Schlör, J. Helbig, Layered timing diagrams – visual constraint programming for system level design, submitted to publication, 1993.

[SHNR93] C. Sudergat, R. Hettler, D. Nazareth, and F. Regensburger. AVL-Trees: A case study for software development in SPECTRUM. Interner Bericht der TU München, 1993.

[Spi89] J.M. Spivey, The Z Notation: A Reference Manual (Prentice Hall, 1989).

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stocker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Data Bases VLDB*, pages 107–116. Morgan-Kaufmann, 1987.

[SSG+91] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, and J. Gouveia. OBLOG — Object-Oriented Logic: An Informal Introduction. Tech. Report, INESC, Lisbon, 1991.

[ST87] D. Sannella, A. Tarlecki On Observational Equivalence. *Journal of Comp. and Sys. Science*, 34, 1987.

[Str67] C. Strachey. Fundamental Concepts in Programming Languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.

[SW83] D.T. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, September 1983.

[Tur85] D. A. Turner. Miranda — a Non-Strict Functional Language with Polymorphic Types. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer Verlag, 1985.

[VHG+93] 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, *Advanced Information Systems Engineering, Proc. 5th CAiSE'93, Paris*, pages 463–482. Springer LNCS 685, 1993.

[WB89] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[WGM79] C. Wadsworth, M. Gordon, and R. Milner. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[Wie91] R.J. Wieringa. A Conceptual Model Specification Language (CMSL, Version 2). Technical Report IR-267, Vrije Universiteit Amsterdam, 1991.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976.

[Wir90] M. Wirsing. Algebraic Specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 677–788. North-Holland, Amsterdam, 1990.

[Wir$^+$92] M. Wirsing et al. *A Framework for Software Development in* KORSO. Informatik-Bericht, LMU München, 1992.

[Zwi89] J. Zwiers, Compositionality, Concurrency, and Partial Correctness – Proof Theories for Networks of Processes and Their Relationship, LNCS 321 (Springer-Verlag, 1989).