Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A COMPARATIVE ASSESSMENT OF SYSTEM DESCRIPTION METHODOLOGIES AND FORMAL SPECIFICATION LANGUAGES

Algirdas Avižienis
Chi-Sharn Wu

September 1990
CSD-900030

FOREWORD

This is the Final Technical Report for the research project "Methodology and Languages for System Specification", carried out under Subcontract C/UB-24, Task No. B-9-3561, Delivery Order 0022 from the CALSPAN-UB Research Center, P. O. Box 400, Buffalo, NY 14225. Mr. Richard J. Daley served as the Subcontracts Administrator for this task.

The research has been performed under the direction of the Principal Investigator, Professor Algirdas Avižienis, Director of the Dependable Computing and Fault-Tolerant Systems Laboratory at the UCLA Computer Science Department. The research personnel were Dr. Chi-Sharn Wu and Dr. Jia-Hong Chen, both Post-Graduate Research Engineers, working under the guidance of Professor Avižienis.

The authors appreciate the comments and advice received from Professors R. Caflish, J. W. Carlyle, B. Ho and D. A. Rennels, who served on Dr. C.-S. Wu's doctoral committee. We are also grateful for many stimulating discussions with Dr. Michael R. Lyu, Dr. Jose Miro-Julia, and Ms. Ann Tai, all researchers at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory. Mr. Pat O'Neill from RADC contributed valuable comments during the six-month technical review on 19 April 1990 at UCLA and via several telephone discussions.

Our work has benefited from the excellent support of Ms. Elizabeth V. Johnson, UCLA Contract and Grant officer and Ms. Ann Goodwin, the Computer Science Department accountant. Ms. Jackie Trang and Ms. Susanna Reyn have provided most capable technical support.

Please address any comments or questions relating to this Technical Report to:

Professor A. Avižienis
Boelter Hall 4731
University of California, Los Angeles
Los Angeles, CA 90024-1596

Internet: aviz@cs.ucla.edu

September 28, 1990

A Comparative Assessment of System Description Methodologies and Formal Specification Languages

Abstract

A classification of formal specification techniques which is useful for a comparative assessment is presented. In this classification, formal specification techniques are grouped into three approaches: *operational, definitional* and *hybrid.* Depending on whether data abstraction or sequencing is emphasized, the operational and definitional approaches both can be further partitioned into two schools: *data paradigm* and *process paradigm.* Five categories are identified, and some representative formal specification techniques in each category are surveyed. A comparative assessment over these specification techniques is given based on a set of criteria, such as usability, verifiability, support for nonfunctional requirements, etc. Our experiences in using formal specification techniques are discussed. A real-time extension of the VDM method, designated RT-VDM, is presented as the most promising approach for further study. Additional recommendations for further study are also made.

Contents

List of Figures

List of Tables

1 Introduction

A complete specification system consists of *methods*, *languages* and *tools*. Specification languages can be classified into three groups according to the level of formality : *informal*, *semi-formal*, and *formal*. Informal specification languages, mainly referring to natural languages, can contain many deficiencies, like inconsistency and ambiguity, which are difficult to detect. Semi-formal languages have a well-defined syntax and partially defined semantics which make building automatic tools possible. SA [DeM78], SREM [Alf85], and PSL/PSA [TH77] are some of the well-known semi-formal specification systems; these systems are widely used in industry because the documents resemble those written in natural language and the semi-formal languages can be learned and understood with limited effort by people who did not have extensive training in formal methods. Formal specification languages, with well-defined syntax and semantics, have the advantage of being concise and unambiguous; they support formal reasoning about the functional specification, and provide a basis for verification of the resulting software product.

The objective of this report is to survey on some representative formal specification techniques and make an assessment of them. In order to provide a basis for systematic study and facilitate the following assessment, a classification for formal specification techniques is presented first in Section 2. In sections 3-5, the surveyed formal specification techniques are presented according to the classification, and a summary is given in section 6. In section 7, a comparative assessment over formal specification techniques is given based on a set of criteria, such as usability, verifiability, support for nonfunctional requirements, etc. Section 8 summarizes the trends in research related to formal specification techniques. Our experiencecs in using formal specification languages are stated in section 9, and a formal specification language, named **RT-VDM**, is proposed in section 10. Finally, some formal specification techniques are recommended for further study.

2 Classification

We can categorize the formal specification techniques into three different approaches: *operational*, *definitional* and *hybrid*. Using the operational approach, a system is described as an abstract model by which the behavioral properties exhibited are those desired for the specified system. Using the definitional approach, systems are specified by such behavioral properties directly. In a hybrid approach, a specification method is extended by combining with other formalisms for specifying more kinds of properties. In Figure 1(a), the fundamental difference between the operational and definitional approaches is illustrated by showing how to specify the behavior of *"stack"*. In the operational approach, a predefined data type, **sequence**, is used to model the stack: *push* is modeled by concatenating an element to the head of sequence, and *pop* is modeled by deleting the head element (using tail function) with the pre-condition that the stack is not empty. In the definitional approach, however, no explicit model (or data structure) is used; the behavior of stack is specified by two equational axioms: the first one states that the empty stack cannot be popped, and the second one describes the "last-in-first-out" property. In literature, researchers also call the operational approach *constructive* and the definitional approach *axiomatic*.

Operational Approach :
(using VDM notation)

$init : \rightarrow$ **seq of** $ELEM$

$init() \triangleq []$
$push :$ **seq of** $ELEM \times ELEM \rightarrow$ **seq of** $ELEM$

$push(stk, elem) \triangleq elem \frown stk$
$pop \ (stk:$ **seq of** $ELEM)\ stk':$ **seq of** $ELEM$
pre $stk \neq []$
post $stk' =$ **tl** stk

Definitional Approach :
(using algebraic notation, i.e. equational axioms)

init : $\rightarrow$ STACK
push : STACK, ELEM $\rightarrow$ STACK
pop : STACK $\rightarrow$ STACK

pop(init) = UNDEFINED
pop(push(stk,elem)) = stk

(a) Using Data Paradigm

Operational Approach :
(using CSP notation)

$STACK = P_{<>}$
where $P_{<>} = push?x \rightarrow P_{<x>}$
$P_{<x>\frown s} = (pop!x \rightarrow P_s \mid push?y \rightarrow P_{<y>\frown<x>\frown s})$

(b) Using Process Paradigm

Figure 1: Diverse formal specifications for the behavior of *stack*

In [Jac88], specification techniques are classified into the classes of *data paradigm* and *process paradigm*. Based on that viewpoint, both the operational and definitional approaches can be further split into two schools: *data* school, which advocates the primacy of data abstractions, and *process* school, which focuses on sequences of events or actions (operations). That is, the prime concerns of the approaches based on the data and process paradigms are *data* and *sequencing* respectively. The two examples shown in Figure 1(a) use the data paradigm. An example of process paradigm is shown in Figure 1(b) where CSP notation (an operational approach) [Hoa85] is used; the specification states that (1) the stack is empty initially; (2) when the stack is empty, it is ready to engage input event *push* for getting an element; and (3) when not empty, either *pop* (output event) or *push* can be engaged. Note that the sequencing of events is emphasized when using the process paradigm, but it is implicit when using the data paradigm.

Table 1 shows the classification for some formal specification techniques. Note that there is no way to formalize this classification since the distinction between different categories is not clear-cut. Sometimes for achieving a higher level of abstraction, the behavior of a model, although using the technique classified as operational approach, can be described by stating its properties. Also, most techniques incorporate both process and data paradigms to some degree since no practical technique can rely purely on data or process notions. However, we believe that such a classification can facilitate a comparative survey.

In general, both data and process paradigms are marred by their their biases. The data approaches do not handle concurrency well. The lack of data abstraction in the process approaches creates complexity and inflexibility to changes in data representation. So in [Jac88], Jackson contends that each school has much to offer, and that an effective approach to software development must contain ingredients from both schools, data and process, in a reasonable balance. As shown in Table 1, it is interesting to note that each hybrid approach indicated there is based on combining the techniques of both the data paradigm and the process paradigm.

3 Operational Approaches

3.1 Data Paradigm – VDM, FDM and Z

VDM (Vienna Development Method) was developed at IBM Vienna Research Laboratories during the 1970s. FDM (Formal Development Methodology) was developed by System Development Corporation (SDC), Santa Monica, California. **Z**[1] was developed by the Programming Research Group at the University of Oxford. In the literature, they are often referred as *model-oriented* specification mehods.

3.1.1 Methods

Model-oriented specification is a technique that relies on formulating a model of the system which defines a mathematical model of its *data* and also a corresponding animation of this model. The data domain is modeled using well understood mathematical entities

[1] Z is pronounced "Zed' not "Zee'.

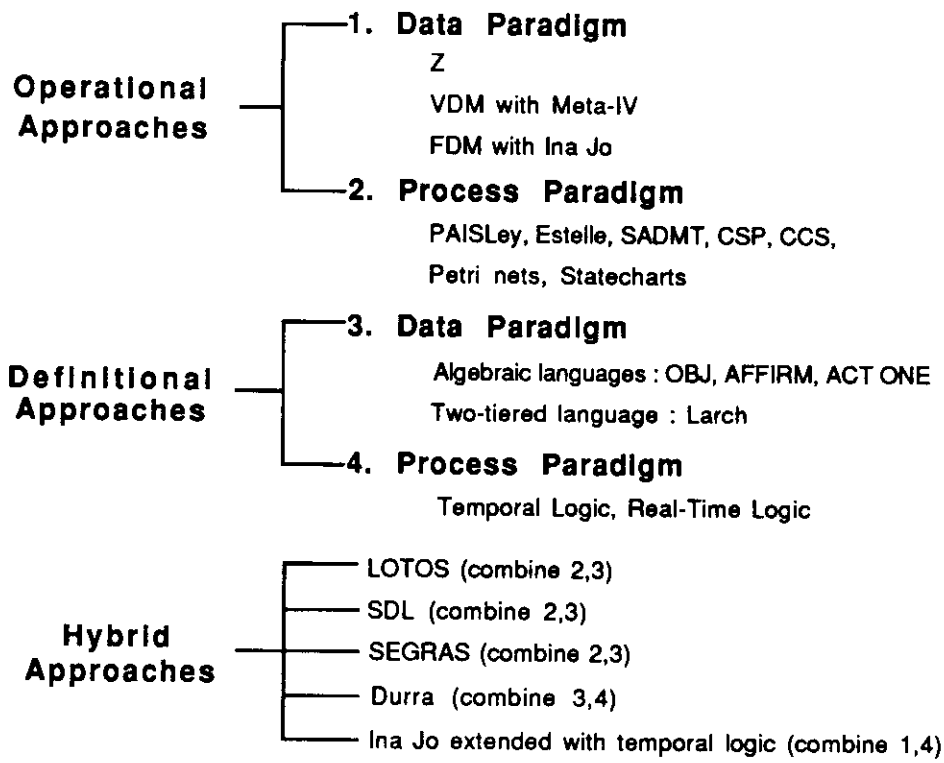

Table 1: A classification for formal specification techniques

such as sets and functions, and animation is specified under the form of a number of *operations* (*transform* in FDM term). The main guideline in constructing a model-oriented specification is to formalize data in such a way that the specification of the operations can be written in a straightforward manner.

Both **VDM** [Jon89] and **FDM** [Kem85, Ber87] are software development methods, based on their own specification languages, which adopt stepwise refinement techniques to derive final implementation from the abstract specification. **Z** [Hay87] is only a specification language, but a systematic refinement technique, which is suited for the style of Z and VDM, has been investigated [MR87, Mor90].

3.1.2 Languages and Tools

Z and the specification languages of VDM and FDM, named `Meta-IV` and `Ina Jo` respectively, are all based on first-order predicate logic and used to model the system as a state machine. These languages sacrifice executability for high-level abstraction. Meta-IV and Z provide a richer set of abstract data types and operators than Ina Jo for system modeling.

During the last 15 years the language Meta-IV has been used in many different variants. The different variants reflect that VDM has been a pragmatic approach where Meta-IV has been extended with properties that were needed for specific applications. But this diversity also hinders the development of tools to support VDM. The British Standards Institute (BSI) is currently working on harmonizing the different variants to produce a standard for Meta-IV, called the BSI/VDM Specification Language (BSI/VDM SL) [LA89]. Although

there are several ongoing projects for building VDM tools in Europe, the availability of sophisticated tools is still unknown. Some available VDM tools are:

- A LaTeX macro for writing VDM specifcation documents, called `vdm.sty` [Wol86].
- EPROS (Evolutionary PROtotyping System) [HI88], developed by S. Hekmatpour at University of Melbourne, is a UNIX-based prototyping environment which enables very rapid generation of working prototypes from a formal specification using VDM notation. The VDM notation is executable only if it is written in a constructive style.
- **Specbox**, developed by a company called "Adelard" in London, can accept the BSI draft standard and provide full static checking and a test harness.

In the development of Z, great emphasis is given to the readability of specifications. It has led to the development of the 'schema', a device for organizing the presentation of a Z specification, which is essentially a syntactic unit for expressing part of a specification. Schemas can be manipulated by operations for extension, restriction, inclusion and composition. Generic schemas can also be written. Some known Z tools are:

- Fuzz, developed by Mike Spivey at Oxford. This is a simple tool including LaTeX macro for documentation preparation, syntax checker and type cheker.
- A proof assistant, called B, developed at Oxford. The tool stores axioms, rules of inference, and application-oriented theories and proofs. Furthermore, it does much housekeeping, thus providing a secure environment for the interactive construction of proofs. In fact, B is a generic theorem prover, and it has been used in case studies using Z [Woo89] and VDM [LLS90].
- A Z toolset, resulting from a project called **Genesis**, developed by IST in UK. The toolset includes an editor, syntax checker, type checker and proof checking.

A FDM specification brings together a formal model of system correctness requirements and a hierarchy[2] of functional representations of the system design both expressed using the Ina Jo language. From this specification, the **Ina Jo processor**, a combined syntax checker and theorem generator, constructs a number of logical formulas which assert that the functional representations satisfiy the formal requirements model. These formulas must be submitted to the ITP[3] (interactive theorem prover) for proof, and the correctness of the specified system depends upon their provability. The program-design specification is the lowest level in the specification hierarchy, providing a module-by-module description of the system implementation and driving the code verification process. One VCG (verification condition generator) is required for each different implementation language used with the methodology. Other than an incomplete VCG for Modula, there has been no development of FDM tools to support code verification. It is not possible to complete the verification process to the code level without a VCG. Thus the FDM has been used exclusively for design

[2]A system specification in FDM is built as a sequence of state machines, progressing from the abstract to the concrete.

[3]The ITP incorporates some automatic features, but it is primarily a proof-checker and bookkeeper.

verification[4]. A execution tool called `Inatest` [EK85], developed by the Reliable Software Group at UCSB, allows Ina Jo specifications to be analyzed by symbolically executing the specifications [Kem85].

3.1.3 Applications

VDM stands for a relatively well propagated method for developing deterministic systems software, like compilers, database management systems, application program generators, etc., as well as major parts of non-deterministic, concurrent and distributed software, such as operating systems, local area nets, office automation systems, etc. Many applications of VDM can be found in [BJ87, B+88].

Some applications using Z are listed as follows:

- A number of projects in the application of Z, conducted at Oxford from 1978-1986, have been reported in [Hay87]. The applications include IBM's Customer Information Control System (CICS), UNIX filing system, and distributed operating system.
- A preventive cyclic retransmission version of the "sliding-window" protocol has been specified and verified [D+88].
- To develop the IEEE standard floating-point transputer for Inmos [Bar89]. That project is to develop the system from a Z specification to silicon implementation. Many proofs were conducted to ensure that the implementation was true to the specification.

The FDM tools have been used in the formal specification of a number of large systems; however, only the security properties of these systems have been verified. This is not due to any inherent weakness in the tools, but rather to the task at hand. In [Kem89], an encryption protocol used in a secure network is specified and analyzed.

3.2 Process Paradigm – PAISLey, Estelle and SADMT

In this category, two kinds of representation techniques can be distinguished:

- *text-based techniques*, such as PAISLey [ZS86], Estelle [D+89], SADMT [L+88], CCS [Mil80] and CSP [Hoa85].
- *graphics-based techniques*, including Statecharts [Har87] and Petri net-based methods, such as SARA [E+86] and PROTEAN [BWWH88].

All the techniques use only some primitive data types, similar to most programming languages, for data domain description. In the following, only some text-based techniques are discussed.

[4]In the past SDC has used a manual approach referred to as specification-to-code correlation [Sol82] to assure that the implementation is consistent with the lowest level Ina Jo specification.

3.2.1 PAISLey

PAISLey [ZS86], designed by Zave at AT&T Bell Labs for describing embedded systems, is a Process-oriented, Applicative, and Interpretable (executable) Specification Language. It is actually based on two computational models: *functional programming* and *asynchronously interacting concurrent processes.* A system specified in PAISLey consists of a set of asynchronous processes; some processes represent virtual objects within the proposed system, while others may be digital simulations of objects in its environment. Each PAISLey process has a state and goes through a never-ending sequence of a discrete state changes, and the state changes are defined in a functional style. A mechanism, called *"exchange functions"*, is provided as a powerful means of specifying asynchronous interactions.

A PAISLey specification is a model of a system that can be executed so as to simulate the behavior of the specified system. Such executability means that a specification can be debugged and verified by the analyst. A timing constraint can be attached to any operation in PAISLey specification. All timing constraints are honored by the interpreter in simulated time when it can and failures are reported when it can not, so that execution of a specification is automatically a performance simulation as well.

The PAISLey environment is a set of four programs that perform static analysis and execution of specification written in PAISLey: the `parser` is used to detect syntatic errors in a specification, the `cross-reference` is to help a user locate references to the stuff defined in a specification, the `consistency checker` is to detect all inconsistencies in a specification, and the `interpreter` is to execute specifications.

3.2.2 Estelle

Estelle [DV89, D$^+$89], is an ISO standard FDT (Formal Description Technique). In Estelle, a distributed system is specified as a hierarchy of communicating modules, which have parent-child relationships. A module's behavior is described by a nondeterministic, communicating, extended finite state machine[5], which uses Estelle primitives and Pascal statements. The Pascal typing system is extended to define Estelle objects, which are all strongly typed as in Pascal. The property enables to detect statically (i.e., during compilation) the specification inconsistencies.

The Estelle development effort was strongly influenced by the OSI architecture and by the desire to describe ISO protocols formally. An especially crucial requirement was the ability to compile Estelle code. Thus Estelle is the result of a compromise between using high-level constructs for the formal description and being able to easily and efficiently implement those constructs.

LOTOS, classified as a hybrid approach in this article, is another ISO standard FDT. The ESPRIT SEDOS (Software Environment for the Design of Open Systems) project, which ran from 1984 to 1988, aimed to further develop Estelle and LOTOS to descibe services and protocols for distributed architectures and to demonstrate their effectiveness as concretely as possible by deriving simulators and other support tools. The results and general comments about the project can be found in [DV89].

[5]Extended FSM is a FSM whose transitions also depend on some internal variables in addition to the input and cuurent state.

3.2.3 SADMT

The SDI[6] Architecture Dataflow Modeling Technique (SADMT) [L+88], developed at the Institute of Defense Analyses (IDA), provides an uniform formal notation, using the typing and functional facilities of the Ada programming language, for the description of SDI system architectures and BM/C3 architectures[7].

The basic building blocks of a SADMT description are the *processes* and *communication links* (i.e. port mechanism), which are the two kinds of abstract components built in Ada. These two components are used together to construct representations of **platforms** which model all physical entities such as a "ground station", a "sensor satellite", or a "fragment of debris". A platform consists of several processes; each of these processes may consist of several other processes, and so on. Each leaf process, which is not decomposable, is specified as an Ada task, and each nonleaf process is constructed from a set of subprocesses by specifying port connections for each subprocess. The platforms are then grouped together to form an initial configuration of the architecture, and a simulation is produced when a configuration of platforms is executed under the control of the `SADMT/SF` (SADMT Simulation Framework), which simulates the physical environment in which the proposed architecture operates.

The model used by the SADMT/SF provides two types of entities called **platforms** and **cones**. Cones are the mechanism by which platforms become aware of other platforms in the system. A platform may "emit a cone" by describing the geometry of the cone and its associated data. This facility is used for modeling communication, radar, and laser beam weapons. Given the physical characteristics of each platform, such as position, equation of motion, life time, and cones' type, the SADMT/SF can simulate the movement of all platforms and the emission of cones. SADMT modules are compiled by an Ada compiler, linked with the SADMT/SF which is also written in Ada, and executed to simulate the performance of the system. During the simulation, platforms can be created dynamically or destroyed.

Due to the requirement that the description be directly executable, SADMT process descriptions using complex Ada templates are significantly more verbose than if direct execution were not required. A tool, called `SAGEN` (for SADMT Generator) [K+88a], is designed to accept a less verbose language, which eliminates much of the drudgery of specifying the SADMT template, and automatically generates the required SADMT template.

SADMT also provides a facility for adding behavioral constraints to the Ada program representing a SADMT model in the form of annotations, similar to Anna (ANNotated Ada) [LH85], for run-time checks. A tool, called `ToolA`, has been built to assist in the validation of these annotations by producing an equivalent Ada program that provides notification if any of the constraints given in the annotations are violated during the simulation.

[6]SDI stands for Strategic Defense Initiative.

[7]BM/C3 stands for Battle Management and Command, Control, and Communications.

4 Definitional Approaches

4.1 Data Paradigm – OBJ and Larch

This subsection discusses two definitional approaches: OBJ, a pure algebraic specification system developed at UCLA and SRI International, and Larch, a two-tiered specification technique developed at MIT and DEC. Both are classified as data paradigm approaches.

4.1.1 OBJ : Algebraic Approach

OBJ is designed to support parameterized programming, using algebraic specification techniques [Gog84]. OBJ is a language based on *equational logic* which can be interpreted directly as rewrite rules, and several executable versions of OBJ (interpreter) have been implemented in USA and Europe. OBJ provides features to assist the development of correct specifications: 'objects' which allow large specifications to be broken down into 'mid-size' pieces, facilities for testing the pieces and their interconnections by executing test cases, strong typing, the systematic use of error conditions (factored out from normal behaviour) and semantic consistency checks. In some respects OBJ can be regarded as an applicative programming language, though efficient execution is not a design objective.

OBJ was originally designed in 1976 by Goguen [GT79] as a language for "error algebras," an attempt to extend algebraic abstract data type theory to handle errors and partial functions in a simple, uniform way. Several versions of an OBJ interpreter have been implemented, including OBJ0 and OBJT developed at UCLA, OBJ1 and OBJ2 [Fut85] at SRI. OBJ3 [GW88] is the latest implementation developed at SRI. Although the syntax of OBJ3 is close to OBJ2, it has a different implementation based on a simple approach to order sorted rewriting, and it also provides much sophisticated parameterized programming.

OBJ has been used for many applications, including debugging algebraic specifications [GCG90], rapid prototyping, defining programming languages in a way that immediately yield an interpreter [GPG81], specifying software systems (e.g, the GKS graphics kernel system, an Ada configuration manager, the MacIntosh QuickDraw program, and OBJ itself). OBJ is also one of the languages for programming a massively parallel machine to execute rewrite rules directly [G+87]; in fact, researchers believe that OBJ on such a machine should greatly out-perform a conventional language on a conventional machine, by directly concurrent execution of rewrite rules. OBJ3 is also applied to theorem proving and hardware verification [Gog88]. The stepwise refinement method, one of the most effective programming methods, based on OBJ was proposed in [NF89].

4.1.2 Larch : Two-tiered Approach

Larch [GHW85] is designed to specify the functionality of sequential programs, in particular the properties of abstract data types. A Larch specification has components written in two languages. A Larch *interface* language, e.g. Larch/Pascal, is used to describe the observable behavior of program modules written in a particular programming language, e.g., Pascal. The Larch *Shared* Language (LSL) is used to write *traits* that define the assertion language used in interface components. Essentially, interface specifications use first-order predicate logic (pre- and post- conditions) to describe state transformations, and

traits use equational axioms (algebraic approach) to describe fundamental abstractions, i.e., abstract data types, that are independent of state, and thus, of any programming language. The two-tiered approach separates the specification of underlying abstractions from the specification of state transformations. In this way, Shared Language components can be reused by different interface language components, and programming language dependent issues – such as side effects, error handling, and resource allocation – can be isolated into the interface language components [Win87]. Each Larch interface language is designed for a programming language, which influences everything from the modularization mechanisms to the choice of reserved word. Larch/CLU and Larch/Pascal are presently the only two moderately well-developed Larch interface languages.

The Larch Project has been designing and implemeting software tools as part of a specification environment, including a syntax and static-semantics checker for the LSL and a theorem prover for semantic checking, and the design of a syntax-directed editor and a specification library. LP [GG89], the Larch prover implemented at MIT, is more than a rewrite-rule engine, but not quite a general-purpose first-order logic theorem prover. The LSLC, the LSL Checker which serves as a front-end to LP, is to check the syntax and static semantics of LSL specifications and generate LP proof obligations. The method of debugging LSL specifications using both LSLC and LP has been studied in [GGH90].

Up to now, no Larch specification for large-scale application was found in the literature. LP has been used in some nontrivial applications, including circuit verification [G+88] and proving properties of an Avalon/C++ program[8] [WG89]. In [WG89], the program is first "encoded" into Larch specification prior to performing program proof with proof checker. With the conclusions indicated in [WG89], some comments from the experience using LP are quoted as follows:

> "The specificand domain is complex and no amount of machine assistance is going to make that less complex."
>
> "Using LP is like programming since the user designs a proof and lets LP execute it."
>
> "We used only a small subset of the full functionality of LP. To use LP at its fullest and perhaps more effectively than we have illustrated here, the user needs to understand concepts from rewrite-rule theory, e.g., confluence, termination, convergence, termination orderings."
>
> "A proof checker does not decrease the amount of thinking required on the user's part; it can alleviate some of the bookkeeping and symbol pushing, but no more."

4.2 Process Paradigm – Temporal Logic and Real-Time Logic

Temporal Logic (TL), a formal language for expressing temporal properties, provides a natural way of describing and reasoning safety properties and liveness properties of a system. A structure of states (e.g. a sequence or tree of states), generated by every individual run of a program, is the key concept that makes temporal logic suitable for program specification.

[8]The program is a highly concurrent atomic FIFO queue implementation. Avalon/C++, designed at CMU, is a programming language dealing with concurrency and faults.

A temporal axiom is an assertion about state sequences, using temporal operators such as □ (henceforth), ◇ (eventually). A temporal logic specification, consisting of a set of temporal axioms, specifies properties that must be true of all state sequences resulting from system execution. Several variants of different temporal logics, including different types of temporal semantics and different ways of real time extensions, have been studied by logicians and computer scientists [Gal87]. Temporal logic has been used for the specification and verification of concurrent program behavior [Lam83], reactive systems [Pnu86a], systems composed of real-time discrete event processes [Ost89b] and hardware design [Mos85].

Real-Time Logic (RTL) [JM86] is a formal language designed for reasoning about timing properties of real-time systems, especially for safety analysis. In contrast to temporal logic, RTL is intended to describe systems for which the absolute timing of events, and not only their relative ordering is important. Time is captured by the *occurrence function*; the notation @(e, i) is used to denote the time of the ith occurrence of event e. RTL formulas, which represent assertion over occurrence functions, are constructed using first-order logic. Given the timing specification of a system and a safety assertion to be analyzed, both in RTL formulas, the goal is to relate the safety assertion to the systems specification. If the safety assertion is a theorem derivable from the specification, then the system is safe with respect to the behavior denoted by the safety assertion, as long as the implementation is faithful to the requirements' specification [JM86].

5 Hybrid Approaches

LOTOS (Language of Temporal Ordering Specification) [DV89, EVD89] is one of the two FDTs developed within ISO for the formal specification of open distributed systems. A LOTOS specification contains two components: the description of process behaviors and interactions (*process abstraction*), and the description of data structure and value expressions (*abstract data types*). Process abstraction is based on many ideas from CSP and CCS, and abstract data types are described by an algebraic specification language based on ACT ONE [EM85].

SDL (Specification and Description Language) [BH89, SSR89], developed and standardized by CCITT, has been developed for use in telecommunication systems including data communication, but actually it can be used in real time and interactive systems. SDL has two paradigms: Abstract Data Type (described in ACT ONE) and Finite State Machines (for modeling system dynamic behavior). The user friendliness of SDL is partly due to the graphical representation, SDL/GR, in which *graphical syntax* is used to give an overview. SDL/GR is complemented by SDL/PR, a textual phrase representation using only *textual syntax*, since graphical symbols are missing (being unsuitable) for some concepts, e.g. abstract data type. SDL/GR and SDL/PR have a common subset of textual syntax, and thus overlap each other.

SEGRAS[9] [Kra87], a formal language for writing and analyzing specifications of distributed software systems that unifies algebraic specifications of abstract data types with high-level Petri net specifications of nonsequential systems in a common syntactic and semantic framework. The data structure of a system, the information content of its local

[9]SEGRAS is a registered trademark of GMD.

states, and static constraints to state changes are specified algebraically using positive conditional equations. Dynamic behavior is specified by high-level Petri nets.

Durra [BW87], intended for real-time applications, is a specification language which combines two formalisms: Larch used to specify functional behavior, and an event expression language used to specify timing behavior[10].

In [WN89], Ina Jo is extended with temporal logic to specify concurrency properties; this method is referred as "Ina Jo + TL" in the following discussion.

A growing field in software engineering, called *multiparadigm programming*, has been advocated for building systems while using as many paradigms as we need, each paradigm handling those aspects of the system for which it is best suited [Hal86, Zav89]. In the same spirit, applying different specification languages for different parts of a complex system in forming a composite specification is also considered as a promising way. Several efforts have been attempted in this direction. In [Ter87], a library problem is specified using a mixture of specification languages Z and CSP. In [ZJ89], the dynamic behavior of control-oriented systems is described by the combination of Statecharts and JSP/JSD structure diagrams, and data domain is described in VDM.

6 Summary of the Surveyed Techniques

Table 2 shows the summary of surveyed specification techniques. Some observations are made as follows:

1. All the techniques using the data paradigm, shown in Table 2(a), are designed for specifying sequential systems. As shown in Table 2(b) and (c), techniques using the process paradigm or using the hybrid approach can specify either (non-realtime) distributed systems or real-time systems.

2. Z, VDM and FDM have the capability for wide range of abstraction, i.e., the languages provide constructs for specifying systems in a wide spectrum ranging from the most abstract level to the concrete level which is closely akin to the final implementation. Based on such capability, stepwise refinement techniques for deriving implementation were developed. On the other hand, definitional approaches, such as OBJ, Larch, Temporal Logic and RTL, provide only the constructs of highest level of abstraction.

3. The representation style of specifications in Z, VDM and FDM can be either *prescriptive* (specify "how") or *descriptive* (specify "what"), depending on the desired level of abstraction and specifier's intention. The descriptive style is more abstract, and is less bound to implementational bias than the prescriptive style. The capability of using both styles also contributes to the wide range of abstraction. LOTOS, SDL and SEGRAS, on the other hand, specify dynamic behavior of the system in prescriptive style and specify the data domain, using algebraic techniques, in descriptive style.

[10]Real-time logic is used to define the semantics of the timing behaviors.

	Operational Approaches			*Definitional Approaches*	
	Z	VDM	FDM	OBJ	Larch
Formal model	set theory, state machine, first-order logic	set theory, state machine, first-order logic	state machine, first-order logic	algebraic ADT	algebraic ADT, first-order logic
Application area	sequential systems	sequential systems	sequential systems	sequential systems	sequential systems
Abstraction capability	high – low	high – low	high – low	high	high
Representation style	descriptive, prescriptive	descriptive, prescriptive	descriptive, prescriptive	descriptive	descriptive

(a) Data Paradigm

	Operational Approaches			*Definitional Approaches*	
	PAISLey	Estelle	SADMT	Temporal Logic	RTL
Formal model	async process, functional programming	async process, extended FSM	sync process, BM/C3 model, Ada	modal logic	first-order logic, event occurrence
Application area	real-time systems	distributed systems	BM/C3 architectures	concurrent systems	timing analysis of real-time systems
Abstraction capability	medium	medium – low	medium – low	high	high
Representation style	prescriptive	prescriptive	prescriptive	descriptive	descriptive

(b) Process Paradigm

	LOTOS	SDL	SEGRAS	Durra	Ina Jo + TL
Formal model	sync process, process algebras, algebraic ADT	async process, extended FSM, algebraic ADT	high-level Petri nets, algebraic ADT	Larch, RTL	Ina Jo, temporal logic
Application area	distributed systems	telecommunication, real-time systems	distributed systems	real-time systems	concurrent systems
Abstraction capability	high – medium	medium – low	high – medium	high	high – low
Representation style	prescriptive, descriptive	prescriptive, descriptive	prescriptive, descriptive	descriptive	prescriptive, descriptive

(c) Hybrid Approaches

1. "ADT" stands for "abstract data type".
2. "async" and "sync" are short for "asynchronous" and "synchronous".

Table 2: Summary of surveyed formal specification techniques

7 A Comparative Assessment

In this section, the surveyed formal specification techniques are assessed based on the following criteria:

1. General suitability for describing **BM/C3 architectures** at the highest level.
2. **Useability**. The specification language should be easy to learn and to use. Support for multiple representations is one way to improve the useability. For example, graphical representation can aid in explaining the specifications.
3. The capability for specifying **nonfunctional requirements**, such as concurrency, security, reliability, performance, fault tolerance, and time-out.
4. **Verifiability**. The specification methodology should provide capability for validation of completeness, consistency, and correctness with respect to both syntax and semantics. The desired supporting tools include: syntax checker, interpreter, theorem prover.
5. The capability for **equivalence checking**. The capability to study the equivalence between two independently created specifications will help to check the consistency of the understanding of the informal requirements. This capability can also help to prove the consistency between the specifications of different abstraction levels. The importance of equivalence notions in the context of formal descriptions of distributed systems has been widely recognized.
6. The support for **deriving implementation** from the specification, either automatically or through rigorous refinement steps.

In the assessment, Durra and Ina Jo + TL, instead of being treated as new languages, are mentioned as the extensions of Larch and FDM, respectively.

7.1 Suitability for Describing BM/C3 Architectures

Among these surveyed techniques, only Estelle and SDL have the potential for describing BM/C3 architectures since they provide, similar to SADMT, both the facility for describing a platform (as a subsystem in Estelle and a block in SDL) as a hierarchy of processes and the port mechanism for interprocess communication. SDL seems more appropriate for describing BM/C3 since it provides both graphical representation and mechanism for describing real-time features. SADMT is different from Estelle and SDL in two main aspects: (1) Estelle and SDL are less verbose, or higher level, languages since they are new-brand description languages instead of the one based on a complex programming language. Both Estelle and SDL describe a process as an extended finite state machine where states are explicitly declared, while SADMT describes a process as an Ada task using the normal syntax and semantics of Ada. (2) SADMT can be regarded as a general technique, like Estelle and SDL, for describing arbitrary systems composed of intercommunicating processes. However, SADMT/SF, which makes SADMT different from other techniques, is

specifically designed to provide mechanisms, such as cones, simulated time and space, and dynamic creation of platforms, for simulating the processes in BM/C3 architectures.

With some extensions, Estelle and SDL have the potential to become the representation techniques for early BM/C3 architectural development purposes. As an example, SDLSIM [SKG87] is an extended SDL which incorporates the performance analysis capabilities into the SDL by introducing some novel concepts, such as mission, resource, delay, scheduling mechanisms.

7.2 Useability

Understandability appears to be inversely proportional to the level of complexity and formality present. In the study of [Dav88], Statecharts, PAISLey, and Petri nets appear to be much more difficult to comprehend than the others (natural language, finite state machine, SA/RT, REVS, RLP and SDL) which are mostly less formal. Roughly speaking, reading and writing a definitional specification initially takes more practice than reading and writing an operational specification, because programmers trained in conventional languages tend to think imperatively. However, it is very difficult to determine whether one technique has higher useability than the other since many factors have to be considered, including human factors. In the following, we intend to evaluate the useability of each technique based on (1) modularity and reusability of components, (2) support of "human-friendly" form, such as diagrams and flow charts, and (3) management tools for specification construction.

Z, OBJ and Larch support modularity and encourage reuse of components by providing libraries and mechanisms for parameterization, renaming, export-import interface, etc. BSI/VDM SL will enchance original VDM for supporting modularity and parameterization [LA89]. In [Ber86], FDM was enchanced to support modularity. Most techniques using the process paradigm or the hybrid approach support modularity to some extent, but do not encourage reuse of components; RTL and temporal logic are two exceptions which do not support modularity. RTL describes timing properties of a system in a global way; temporal logic was traditionally used in a global, non-modular and non-compositional way since it reasons about the global state of the program, but some researchers have been investigating methods of the modular specification using temporal logic [BKP84, Lam83]. Modularity is supported only a little in PAISLey since process is the unique structuring unit. SDL and LOTOS support modularity and use the algebraic technique, which encourages reusability, for specifying abstract data types. SEGRAS supports modularity and encourages reuse in both system structure (for dynamic behavior) and data structure.

Most surveyed specification techniques lack the support of "human-friendly" form, and only SDL and SEGRAS support graphical representation. A tool, called GROPE (Graphical Representation Of Protocols in Estelle), was prototyped with the intention to animate Estelle specifications in graphical form [NA90].

Syntax-directed editors exist for some languages, such as Larch, Estelle, LOTOS, SDL and SEGRAS. In ESPRIT SEDOS project, the workstations of Estelle and LOTOS, for increasing efficiency and productivity in the development phases, have been prototyped [DV89]. SDL is the language in widest use in industry by specifiers and developers of telecommunication systems, such that diverse commercial supporting tools have been developed; YAST (Yet Another SDL Tool) [Z+89], for example, is a set of tools that sup-

port the use of full SDL'88, including the graphical editor and on-line SDL tutorial. The SEGRAS laboratory, an interactive specification environment, is designed to support the stepwise development of large specifications in SEGRAS [Kra87].

7.3 Nonfunctional Requirements

Most nonfunctional requirements, also called *constraints*, are difficult to specify formally. Some constraints (e.g., response to failure, fault tolerance) are related to design solutions that are not known at the time the requirements are written. Many constraints (e.g., maintainability) are not formalizable, given the current state of the art, and many other are not explicit [Rom85]. Given the current state of the art, only some types of constraints are addressed by current formal specification techniques, such as *security*, *concurrency*, and *timing constraints*.

Some security properties can be verified with respect to functional specifications using the techniques with data paradigm, although they do not intend to provide constructs for specifying nonfunctional requirements. A large portion of the current formal verification work has been dominated by security-related projects, and FDM is one of techniques which have been used extensively in this area [C+81, Kem89]. In [WN89], Ina Jo was extended with temporal logic for specifying concurrency. Larch has been used to demonstrate the applicability of specifying some nonfunctional properties, such as synchronization [BHL87], persistence and atomicity [WG89]. In the work of Durra [BW87], Larch was extended with an event expression language for specifying timing behavior.

All the techniques that use the process paradigm or the hybrid approach deal with concurrency. When describing timing behavior we usually want to be able to specify that (a) "something should occur within a certain time, otherwise .....", or that (b) "after a certain time something must occur". In Estelle, a delay mechanism, which can specify the delay time for each enabled transition in a finite state machine, is provided for modeling *time-out behavior* (related to (a)) or a *waiting function* (related to (b)). In SDL, timing behavior is described by setting a watch-dog timer which can be made in three different ways: (1) using a timing device, within or outside of the system that takes care of waking up the process at appropriate time, (2) using a continuous signal[11], (3) using the SDL construct of timer[12]. Both PAISLey and SADMT provide only method (1) of SDL for modeling time-out behavior. SADMT, PAISLey and SDL provide facilities for describing some kinds of performance requirements; using SADMT, each platform can be associated with a resource assignment module which define the values for transit and processing delays honored during simulation; PAISLey provides a timing attribute attached to functions in the functional specification, referring to the evaluation time of the functions in the form of a random variable with lower/upper bounds, mean, or the distibution; different types of transmission delay can be modeled in SDL [SSR89]. Estelle, temporal logic, LOTOS and SEGRAS support the representation of time ordering aspects and handle concurrency well, but provide no construct for time measures; some real-time extensions of temporal logic

[11]Built-in operator **NOW** can be used in any expression yielding the value of the current time.

[12]SDL timer is an entity which can be activated by the **SET** statement and produce a signal upon the expiring of the time set.

[PH88, Ost89b] and LOTOS [QF87] for expressing time quantitatively have been proposed. RTL is designed specifically for specifying timing behavior and performance requirments.

7.4 Verifiability

Most techniques with data paradigm, based on either an extension of first-order predicate calculus or equational logic, provide a proof theory; all the theorem provers, primarily are proof-checkers and bookkeepers that provide an environment for supporting formal reasoning where the humans guide proof creation using their insight into the problem domain. OBJ is the only exception which provides an interpreter instead of theorem prover; despite being incomplete, testing using the interpreter is a practical way of increasing confidence in the correctness of the specification. In [Gog88], the technique for proving theorems using OBJ and its interpreter was developed.

Simulators for dynamic behavior checking exist for PAISLey, Estelle, SADMT, LOTOS, SDL and SEGRAS[13]. Extensive efforts spent on the verification techniques for standard FDTs have produced some other verification tools for Estelle [D+89], LOTOS [EVD89] and SDL [FM89]. Temporal logic provides sound *global* proof systems[14] for reasoning the properties of *entire* systems, but the technique to support compositional proof systems based on modular specifications is still under intensive investigation [BKP84]. In order to use temporal logic itself as a tool for programming and simulation, two programming languages based on Interval Temporal Logic (ITL), named *Tempura* [Mos86] and *Tokio* [F+86], were designed and implemented. A decision procedure for RTL formulas, although inherently computationally expensive, has been proposed in [JM86] for safety analysis.

7.5 Equivalence Checking

Among the surveyed techniques, only LOTOS, which is based on a process algebra derived from CCS, has developed the equivalence theories, using the notions of behavioral equivalence, and implemented the tool for equivalence checking[BC89]. In the work of [VB89], the mapping of SDL processes and queues onto an extended version of CCS was proposed to make equivalence checking possible[15]. The equivalence notion of algebraic specifications, which could be applied to OBJ and Larch, has been addressed in several studies [BW88].

7.6 Support for Deriving Implementations

Both VDM and FDM support stepwise refinement techniques for deriving implementations. Some theoretical studies have been done on the stepwise refinement methods for Z [MR87, Mor90] and OBJ [NF89]. The translation of OBJ notations into an efficient implementation

[13]An interactive Petri net simulator of SEGRAS was under developement as indicated in [Kra87], but its current status is unknown.

[14]The proof systems are referred as *global* since they are only applicable to entire systems, and cannot be applied to components of systems.

[15]CCS is a process algebra which has a clearly defined equivalence relation between processes. SDL, however, is a language and not an algebra, such that there is no way of telling whether two SDL specifications are equivalent (apart from their being extually identical).

is still a big challenge; in [Shu89], a development strategy was designed for translating OBJ into the MALPAS intermediate language which is then refined until it is easily translated into code. Larch, based on a two-tiered specification technique, allows some implementation issues, such as modular decomposition and exception handling, to be specified in interface languages.

The transformational methods for translating a PAISLey specification into implementation, as proposed in [Zav84], where an operational approach for software development was advocated, are not available yet. A generator of C source code for Estelle was built for both simulation and implementation purposes [RC89]. A set of tools, called *LIW* (LOTOS Implementation Workbench) [M+89], has been designed for providing an interactive process to refine a LOTOS specification into a C source code; the very high level abstraction of LOTOS precludes a direct compilation. For SDL, several implementation tools have been built, including code synthesizer generating C++ source code from SDL-PR, and CHILL[16] source code generators [FM89]. The development methodology for reactive systems based on temporal logic was discussed a little in [Pnu86b], and the need for a compositional proof system as a prerequisite for such methodology was pointed out. The formal grounds of stepwise implementation using SEGRAS are currently ongoing research.

8 Trends in Research

Some active research issues on formal specification techniques are summarized as follows:

1. **Hybrid Specification Methods.** It is interesting that LOTOS, SDL and SEGRAS all adopt algebraic techniques for describing properties of the data domain at the lower tier, similar to Larch, and use other formalisms (the operational approach) for specifying dynamic system behavior at the upper tier. This kind of two-tiered approach seems to become a general solution for combining the best from the worlds of data paradigm and process paradigm. Several other efforts have been attempted to deal with practical complex systems using the hybrid approach. In [ZJ89], the dynamic behavior of control-oriented systems is described by the combination of Statecharts and JSP/JSD structure diagrams, and data domain is described in VDM. The RAISE[17] project [NG88] is intended to extend VDM in several areas, including concurrent processes based on CSP, the use of algebraic axioms for higher level of abstraction, and the support of modularity and parameterization.

2. **Model Checking Techniques.** *Model checking* has become a well known method to carry out automatic verification of distributed systems. In this method, a model representing the behavior of the system is described using a certain operational approach (serving as a behavioral specification), and the desired properties of the system are specified in temporal logic formulas (serving as a requirement specification). **EMC** [CES86] and **XESAR** [R+87], for example, are two typical systems which use a subset of CSP and a variant of Estelle, respectively, for implementing the behavioral

[16]CHILL (CCITT High Level Language), a general-purpose language suited for programming embedded systems, supports concurrency and interprocess communication.

[17]RAISE is an acronym for "Rigorous Approach to Industrial Software Engineering".

model, and use branching time temporal logic for specifying the desired properties. For verification, a complete state graph representing all the behaviors of the system is generated from the behavioral model first, and then a model checking algorithm is applied to check if the state graph satisfies the temporal logic formulas. The limits of model checking method using XESAR were discussed in [G+89b]. A model checking technique for verifying real-time systems is proposed in [Ost89a, Ost89b], where a timed transition model is provided as a generic computational model for real-time systems, and a real-time temporal logic is used for specifying the properties to be verified.

3. **Object-oriented Specification.** A more recent paradigm for system structuring is the object-oriented approach, in which the system is divided into objects, each of which has its own set of operations. The benefits of object orientation include support for modular design, code sharing, and extensibility. The object-oriented approach offers one of the most promising ways of structuring a system in a way which increases cohesion within its parts and reduces the coupling between them. It is therefore important that the specification should be able to reflect this structuring. Although most object-oriented languages have been developed for implementation and prototyping, efforts on object-oriented extensions for several specification languages have been attempted, including Z [DD90], VDM [Bea88], Estelle [SG88], LOTOS [CRS90], SDL [BMPD87, HH89]. Algebraic languages are in a sense 'naturally' object oriented; OBJ [GW88] is an example of a language based on object-oriented ideas which uses equational specifications to define the behavior of objects. Without object-oriented extension, both Z [GI90] and SDL [Mor89] have been used, by integrating with some object-oriented design methodology, for developing complex software systems. In recent years, object orientation has gained importance as a design methodology for distributed systems, thus there is a strong need to formally specify object-oriented systems; LOTOS has been shown to be well suited for specifying object-oriented systems in [May89]. In [J+90], an object-oriented specification method for reactive systems is presented.

4. **Specification Languages for Real-time Systems.** The timing specification is a particularly important issue in specifying the requirements and in design of real-time systems. In addition to the specification languages specifically designed for real-time systems, several existing specification languages have been enhanced with timing expressions, including LOTOS [vHTZ90, QAF90], CSP [K+88b, Zic90], Petri-nets [CR83, G+89a], temporal logic [KdR83, Ost89b]. In [JG89], different formal models for real-time systems are reviewed. Formal models of real-time programs are still at an early stage of developement, and a great deal of work remains to be done on aspects such as designing language constructs for real-time behavior and developing proof systems and associated semantic models.

5. **Theory and Practice of the Compositional Method and Stepwise Refinement Technique.** Compositionality [Zwi89] and stepwise refinement techniques [BdRR90] have attracted a lot of research recently in the context of specification and design of distributed systems. It is of great value for a specification method to be

compositional so that the specification (and the properties) of a composed system can be constructed (and deduced) from the specification of its components without referring to their internal details. The stepwise refinement method postulates a system construction route that starts with some relatively high-level specification, goes through a number of provably correct development steps, each of which replaces some declarative, non-executable (or merely inefficient) aspects of the specification by imperative executable constructs, and ends with an (efficiently) executable program. A compositional method for real-time distributed computing is illustrated in [HR90]. In [Bac90], a refinement calculus for sequential programs is applied to stepwise refinement of both parallel programs and reactive programs.

9 Our Experiences in Using Formal Specification Languages

In order to facilitate experimental investigations into the design and evaluation of Muti-Version Software (MVS) [Avi85] as a means of achieving fault-tolerant systems, a distributed MVS supervisor and testbed, called the DEsign DIversity eXperiment (DEDIX) system [A+85], has been designed and implemented by the UCLA Dependable Computing and Fault-Tolerant Systems research group. Research was initiated to produce a formal specification for DEDIX after implementing the prototype DEDIX in C language. For the purpose of validation, the specification should be executable or provable.

The design of the prototype DEDIX is based on a set of hierarchically structured layers, shown in Figure 2, to reduce complexity and facilitate the inevitable modifications. Each site has an identical set of layers and entities. These layers, from the bottom to the top, are *Trasport Layer*, *Synchronization Layer*, *Executive Layer* and *Version Layer*.

Being the first attempt, Larch was chosen to specify the synchronization protocol of DEDIX which had been described using an Extended Finite State Machine in [GP85]. Since Larch does not support the notion to express timing constraints and concurrency, the effort did not get satisfactory results. In addition, we found that it is difficult to specify the protocol directly in Larch since our understanding of the protocol is based on the state transition model which is usually viewed as a mutually exclusive alternative to the axiomatic approach[18]. A protocol description language, named SPEX [Sch82], based on the model of a non-deterministic state machine, is thus used to specify the protocol at the first step, and then SPEX is translated into Larch Shared Language[19]. In this piece of work, SPEX and Larch can complement each other, since SPEX is more readable and Larch provides equational axioms for theorem proof. We have a strong feeling that the use of equational axioms becomes cumbersome, but it makes mechanical proof possible, when dealing with the state transition systems for which operational approaches are well suited. LOTOS and SDL, two hybrid techniques that use operational approaches at the

[18]As pointed out in [S+82], the notion of specifying state transition machines axiomatically seems relatively unexplored, although Flon and Misra [FM79] hint at it.

[19]In [Sch82], a set of translation rules to generate algebraic specifications from SPEX are proposed, and the algebraic specification is treated as the semantics of SPEX. Since the Larch Shared Language is an algebraic approach, the translation rules are adopted with little modification in our work.

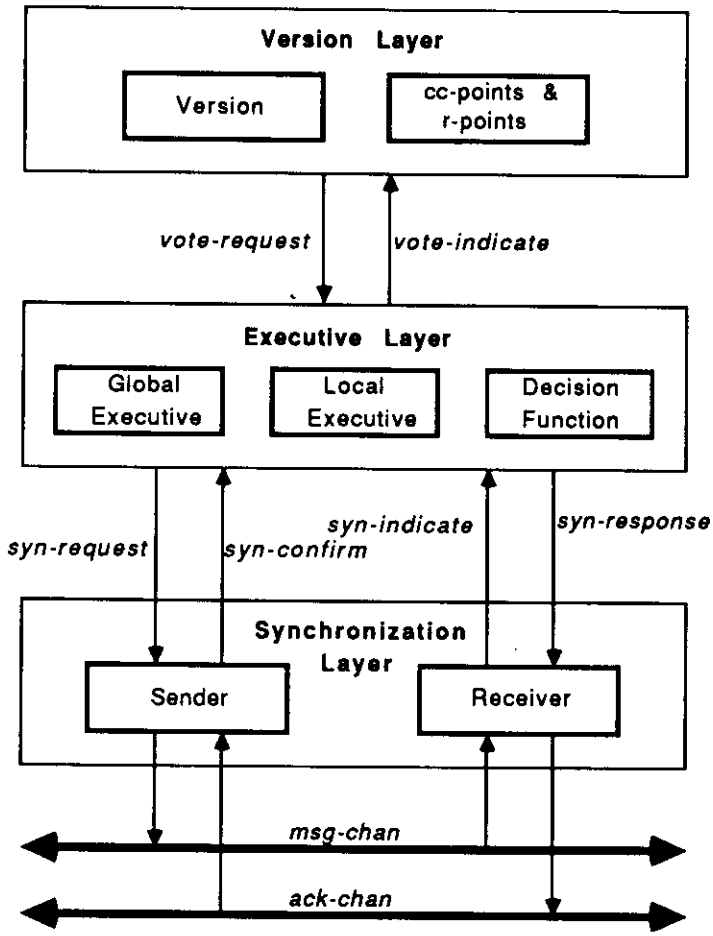

Figure 2: The architecture design of DEDIX

upper tier for specifying dynamic process behavior, are two-tiered approaches which try to apply algebraic technique in the appropriate domain, i.e., data domain at the lower tier.

Later, PAISLey [ZS86] was considered as an appropriate language to specify DEDIX because it provides the following features: *interprocess interactions*, *executability*, *interface to the C language*[20], and *expressions for timing constraints*. Being the second attempt, an effort was initiated to specify the synchronization layer of DEDIX in PAISLey. The successful experience with the synchronization layer then encouraged us to use PAISLey to specify the remaining parts of DEDIX. Like programming languages, no specification language is suitable for all kinds of problem domains. The fact has been recognized later that PAISLey is appropriate for specifying the local executive and global executive, but not appropriate for specifying the decision function. After some evaluations, Prolog [CM81] was chosen to specify the decision function as a form of prototyping. This piece of work demonstrates that specifying a software system in two different specification languages is feasible, as long as there is a clear-cut interface between the subsystems specified in different languages. The specification of the decision function is validated by a Prolog interpreter, and the PAISLey specification of the other parts can tolerate the incompleteness caused by the undefined decision function. The size of the specification is medium; there are about nine hundred lines in PAISLey and two hundred lines in Prolog, and the total size of the specification (including comments) is about 1,700 lines [Wu88]. DEDIX was implemeted in about 14,000 lines of C code.

A system specified in PAISLey consists of *processes* that run in asynchronous parallel and interact over virtual communication channels. The structure of the PAISLey specification for DEDIX is shown in Figure 3, where processes are represented by ellipses. The specification is divided into five modules: *synchronization/transport layer*, *local executive*, *global executive*, *voter* and *version*. The synchronization/transport layer at each site is specified by six processes to describe the behaviors of broadcasting, time-out[21] and synchronization. Both local executive and global executive are specified by one process. Although the decision function is specified in Prolog, the timing behavior of the voter and its interactions with other modules are specified by a process which leaves the decision algorithm undefined. Figure 3 shows the sequence of communications among the processes when a cc-point is executed by DEDIX. The communications are implemented with the *exchange functions* in PAISLey. Although exchange functions allow bidirectional information flow, only one direction is utilized in our specification.

With the experience of specifying DEDIX using PAISLey, some disadvantages of PAISLey have been recognized as follows:

1. *Lack of support for modularity.* As shown in Figure 3, the hierarchical structure is not supported in PAISLey since *process* is the unique structuring unit.

2. *No construct for describing time-out behavior in a succinct way.* Each timer has to

[20]Functions left undefined in a PAISLey specification may be defined instead by functions written in the C language. So those functions which are awkward or useless to specify in PAISLey can be defined by functions in C (or even left undefined for those better suited to another formalism, since PAISLey can tolerate incompleteness).

[21]Two timer processes for the sender and the receiver, respectively, are not shown in Figure 3 for simplicity.

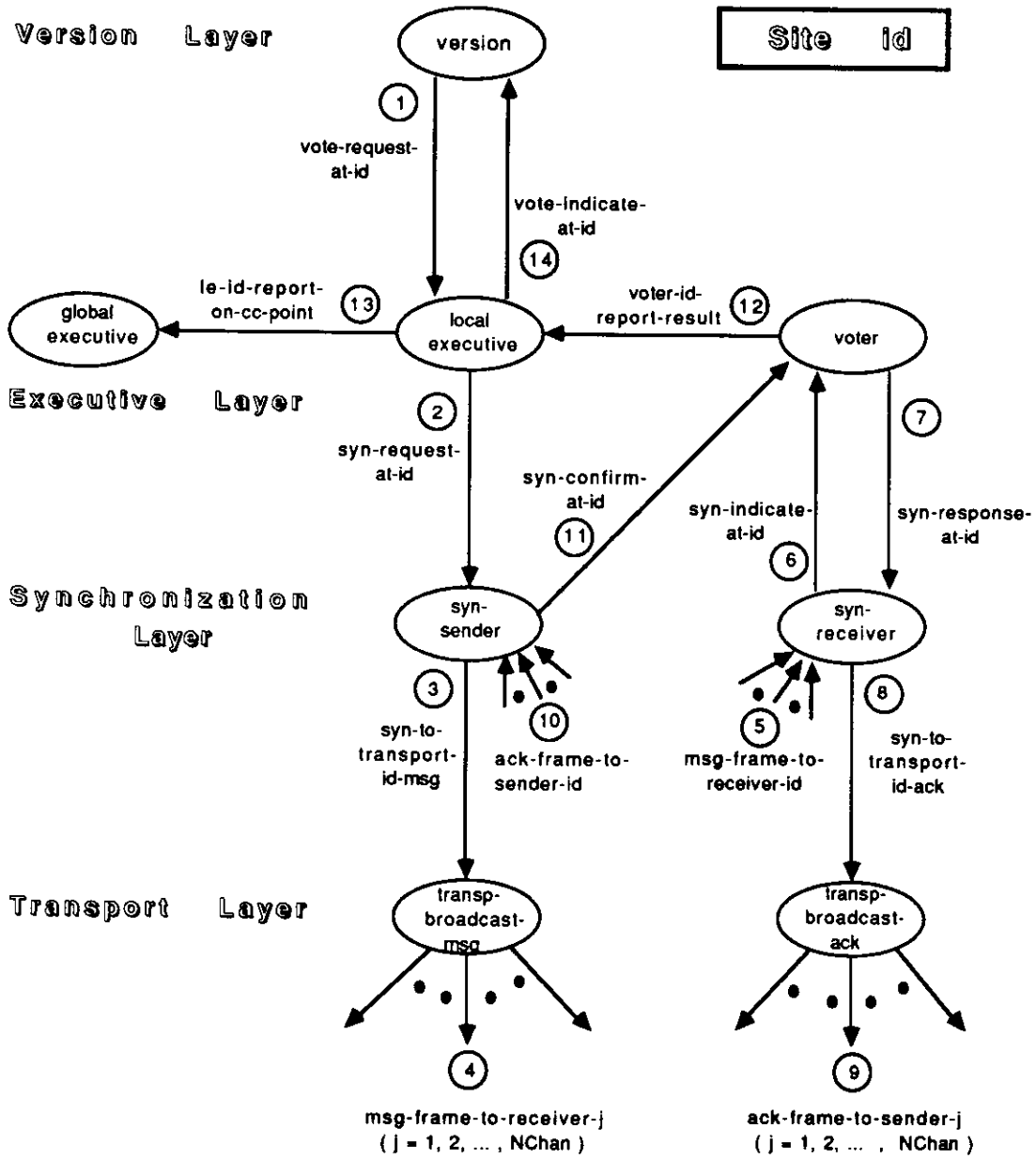

Figure 3: DEDIX specified in PAISLey: a scenario of the normal execution of cc-point at site *id*

be modeled as a process which communicates with the process to be wakened up.

3. *Lack of reasoning capability.* PAISLey provides no formalism for theorem proof, and its simulation cannot guarantee the correctness of the specification.

4. *Functional style is the only style for behavioral description.* There are three disadvantages of such limitation. First, requiring a specification notation to be directly executable restricts the forms of specification that can be used, and thus the level of abstraction using PAISLey is limited. Second, PAISLey does not have the power of typical functional programming languages, and its functional style is cumbersome for describing some complicated operations. Third, since each process is specified by a function definition, different events which affect the behavior of the process are mixed, and thus not explicit, in a single function definition.

In the following section, a VDM-based specification language, called RT-VDM, is proposed. We claim that the above disadvantages of PAISLey can be avoided when RT-VDM is used, and DEDIX can be specified as a whole in a modular way using RT-VDM.

10 A VDM-based Formal Specification Language: RT-VDM

As mentioned in previous sections, VDM [Jon89, B+88] is a *model-oriented* approach, and it provides a rich set of notations to formulate a model of the system which defines a mathematical model of its *data* and also corresponding *operations* on the data. For real-time applications, timing behavior is as important as functional behavior. Over the past few years, VDM has been shown to be well suited to developing sequential software. However, its lack of modularity and the absence of a method for handling temporal aspects of real-time systems are shortcomings which limit its application in complex real-time systems. In this section, a VDM-based specification language, called **RT-VDM** (*Real Time* extension of VDM), is designed for specifying real-time systems. In order to integrate the formal specifications of functionality and real-time requirements in an unified and modular way, RT-VDM enhances VDM with the following features:

1. *Modularity.* A structuring mechanism is supported to model a complex system in a hierarchical, modularized structure.

2. *Concurrency.* The notion of asynchronous processes is incorporated to express the concurrent execution.

3. *Timing Constraints.* Expressions are provided for describing timing constraints, including system performance, time-out, and transmission delay.

In the following subsections, the language constructs of RT-VDM are described. Some detailed case studies of RT-VDM and their verifications can be found in [Wu90].

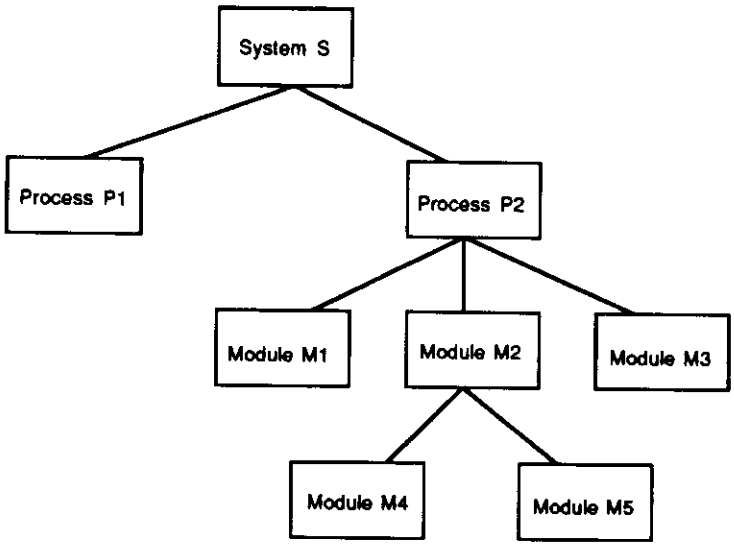

Figure 4: An example of system structure described in RT-VDM

10.1 Overview of RT-VDM

The behavior of a system is composed of the combined behavior of a number of *asynchronously interacting processes* in the system. A process is a state transition machine which runs autonomously and concurrently with other processes. The communication among the processes is through *interaction points* which are modeled as shared variables. To cope with complexity, a process (or a module) can be further partitioned into *modules* with interaction points interconnecting among them. With the structuring mechanism, the overall structure of a system specification may be organized in a hierarchical fashion. Figure 4 shows an example of the system structure where P1 is the process without further partition and M2 is the module further decomposed into two modules M4 and M5. In such hierarchical structure, three kinds of nodes are distinguished:

1. The root, called **system node**, describes the top level structuring of the system.
2. The non-leaf nodes, called **composite nodes**, comprise the children nodes.
3. The leaf nodes, called **primitive nodes**, describe the primitive operations which constitute the system behavior.

Note that a process can be specified as either a primitive node or a composite node. In the following, we use **node** as a general term to stand for either primitive node or composite node.

In many applications, a system may contain several identical processes or identical modules in different processes. So in RT-VDM specifications, the **types** of nodes are specified, and nodes are declared as **instances** of nodes' types.

The specification method using RT-VDM can be classified as an operational approach in the sense that a real-time system is specified as an explicit model of the proposed system interacting with an explicit model of the environment. Both the proposed system and the environment are modeled as a set of RT-VDM processes.

SYSTEM *system-name*
Need *package-name-list*
Process Type
process-type-list
Process Instance
process-instance-declarations
Interconnect
process-interconnection-description
Performance Requirement
performance-statements
END *system-name*

Figure 5: A template for the description of system nodes

10.2 Construct for System Node

Figure 5 shows the template for describing a system node. The `Need` section lists the names of the *packages* from which some useful information is imported. The packages are usually formed by grouping the constants, data types, and functions which are commonly needed in the descriptions of different nodes' types. The processes, which constitute the system, are defined in `Process Instance` section with the types declared in `Process Type` section. The `Interconnect` section describes the interconnections among these processes. The system performance requirements, which should be enforced by the implementation, are specified in `Performance Requirement` section. The construct is intended to separate performance requirements from the functional specification. Of these sections, `Process Type`, `Process Instance` and `Interconnect` are mandatory sections.

10.3 Construct for Composite Nodes

Figure 6 shows the template for describing the type of a composite node whose instances are either processes or modules. The `Interface` section declares the *interface variables*, representing the interaction points, which are used for interacting with the outside world. The children nodes, which constitute the composite node, are defined in `Module Instance` section with the types declared in `Module Type` section . The `Interconnect` section describes the interconnections among the children nodes. The `Attach` section describes the attachment of the interaction points of children nodes to those of the composite node; through the attachment, the children nodes can communicate with the outside world. The `Performance Requirement` section has the same meaning as that in the system node except with a different scope of description; the scope is over the combination of the descendant nodes. Of these sections, `Module Type` and `Module Instance` are mandatory sections.

10.4 Construct for Primitive Nodes

Figure 7 shows the template for describing the type of a primitive node. The `Constant`, `Data Type`, and `State Variable` sections define the constants, data types, and local state

[PROCESS TYPE *node-type-name*]
[MODULE TYPE *node-type-name*]
Need *package-name-list*
Interface
interface-variable-declarations
Module Type
module-type-list
Module Instance
module-instance-declarations
Interconnect
module-interconnection-description
Attach
attachment-description
Performance Requirement
performance-statements
END *node-type-name*

Figure 6: A template for the description of composite nodes

variables, respectively, which can be accessed only at the local node. The initial values of local state variables are defined in the `Initial State` section, and the interface variables have default initial values. In the `Operation` section, which is the core part of the specification, the primitive operations which constitute the system behavior are specified in VDM notation; each operation is specified in two parts: the pre-condition defines the condition which **enables** the operation, and the post-condition defines the changes on the local state variables and interface variables once the operation is executed. The `Auxiliary Function` section defines the functions which are auxiliary for specifying the operations.

In order to express the requirement that some enabled operations must be delayed for execution, the VDM notation is extended with a *delay* statement which is an optional in the operation definition. A delay statement for operation *OP* is in the form "**delay** [E1, E2]" where E1/E2 is the minimum/maximum amount of time the operation *OP*, once enabled, must/may be delayed for execution. Infinity can be represented in E2 using the symbol ∞. A simple form "**delay** [E]" stands for "**delay** [E, E]". If the enabled operation is still enabled when the delay period elapses, then it becomes *fireable*. Fireable operations are the candidates to be executed next. The operations without delay statements become fireable immediately once they are enabled. Within one process, several operations can become fireable simultaneously, but some of them could be *disabled* later by the execution of a certain fireable operation.

In the `Fairness Constraint` section, some operations are declared to hold a fairness constraint. An operation, which has the potential of being disabled by the execution of other operations, holds a fairness constraint if the operation has to be executed eventually after being enabled and disabled a finite number of times. In the literature, different notions of fairness has been proposed [QS83, LPS81]. The notion of fairness used in RT-VDM is *strong fairness*, i.e., the operation is executed infinitely many times if it is enabled infinitely many

[**PROCESS TYPE** *node-type-name*]
[**MODULE TYPE** *node-type-name*]
Need *package-name-list*
Interface
    interface-variable-declarations
Constant
    constant-declarations
Data Type
    data-type-declarations
State Variable
    state-variable-declarations
Initial State
    initial-value-declarations
Operation
    VDM-operation-definitions
Auxiliary Function
    VDM-function-definitions
Fairness Constraint
    operation-list
Always
    operation-names
Performance Requirement
    performance-statements
END *node-type-name*

Figure 7: A template for the description of primitive nodes

times. Fairness constraints are generally crucial to the satisfaction of liveness properties[22].

The **Always** section declares those operations which should not be disabled by other operations (i.e., must be executed eventually) once they have been enabled.

10.5 Expression of Performance Requirement

In [Das85], timing constraints are classified into two categories: *performance constraints*, which set limits on the response time of the proposed system, and *behavioral constraints*, which make demands on the rates at which the environment applies stimuli to the system. In RT-VDM, using the same constructs for modeling both the environment and the proposed system blurs the distinction. Instead, two types of timing constraints are distinguished as follows:

- *delay constraints* which are specified in delay statements as a part of the operation definitions. Delay constraints are suitable for modeling some kinds of timing behaviors, like time-out and event occurrence rate. The delay constraints can be treated as a part of functional requirements, since they can be interpreted in the functionality simulation using the algorithm which simulates Estelle specifications with delay constraints [DB87].

- *performance requirements* which appear in Performance Requirement sections. This category is related to real-time constraints which should be enforced by the implementation, and thus are independent of the functionality part.

The event-action model proposed in [Mok85] is adopted for specifying the performance requirements. The event-action model is used to specify the causal and temporal relationship of the computational *actions* that must be taken in response to *events* in a real-time application; it is also a formal language which can be translated in real-time logic (RTL) [JM86] for reasoning about the timing properties of a design and for safety analysis. In order to fit the event-action model into the RT-VDM model, the correspondence between them is recognized as follows:

- *Actions*, which are schedulable units of work in event-action model, correspond to *operations* in RT-VDM.

- An *event* serves as a temporal marker, i.e., the occurrence of an event marks a point in time which is of significance in describing the timing behavior of the system. In RT-VDM, an event can be referred to either as completing the execution of an operation, using the notation $\downarrow OP$, which may trigger the execution of other operations, or simply as the pre-condition of the triggered operation becoming true, using the notation *pre-OP*. External events, like pushing a button or turning a switch on, which are a part of environment, are modeled as the completing of the execution of operations of environment processes which manipulate the interaction points.

[22]For example, a successful transmission will eventually occur after transmission loss has occurred consecutively a finite number of times; the operation of successful transmission holds a fairness constraint.

In order to specify the performance requirements, expressions for the execution sequence of operations are required. The syntax in BNF is given as follows:

$$E_{op} = operation \mid E_{op}; E_{op} \mid E_{op} \,|||\, E_{op} \mid E_{op} [] E_{op}$$

where ';', '|||', and '[]' are the operators representing *sequential*, *interleaving*, and *nondeterministic selection* executions, respectively. Given A ; ((B [] C) ||| D) as an example, ABD, ADB, ACD, and ADC are the four possible execution paths. The only restriction on choosing the execution path is that the pre-condition of each operation on the path must be satisfied.

The performance requirements are described using the following forms which are the same as in [JM86]:

1. **while** <state predicate>, **execute** <E_{op}> **with period** = <time>, **deadline** = <time>
2. **when** <event>, **execute** <E_{op}> **with deadline** = <time>, **separation** = <time>

The first form is referred to as a *periodic* timing constraint which requires some operations to be executed at fixed intervals while some state predicates are true; <state predicate> should be true during the execution of <E_{op}>, i.e., it should be satisfied by the pre-condition of each operation in <E_{op}>. The second form is referred to as a *sporadic* timing constraint which requires some operations to be executed before a specified deadline elapses after the occurrence of a certain event; the separation parameter, as an optional, specifies a lower bound on the length of an interval separating two successive occurrences of the triggering event.

10.6 Remarks

Basically, RT-VDM takes advantage of some important features from three independent techniques:

1. **VDM** [Jon89]. The domain constructors available in VDM (like sets, maps, sequences and trees) are used as generic abstract data types which are powerful enough for describing the data domain of complicated software systems. The notion of operation in VDM is adopted in a natural way to define the state transitions in distributed systems. The succinctness of the RT-VDM specifications illustrated in [Wu90] is mainly attributed to the use of VDM notations.

2. **Estelle** [D+89]. The delay statement, the notion borrowed from Estelle, facilitates some timing expressions, such as time-out. Both RT-VDM and Estelle model the systems in a hierarchical structure. Similar to the semantics of Estelle, concurrency in RT-VDM is modeled by the interleaving of atomic transitions.

3. **Event-action model** [JM86]. The model is incorporated in RT-VDM for specifying the performance requirements. The event-action model is to capture the temporal ordering of the computational actions (operations in RT-VDM) that must be taken in response to events (completion or enabling of operations in RT-VDM) in a real-time application.

11 Recommendations for Further Study

VDM and Z are promising techniques for the specifying of sequential systems [BHL90]. VDM and Z have been used extensively in a variety of applications, especially in the developement of sequential software. Both VDM and Z, sharing a common philosophy, insist that the specification of requirements should be formulated from the beginning at the highest possible level of abstraction, using all the available power of mathematics to describe the desirable, observable and testable properties of the product which is to be implemented. The British Standards Institute (BSI) is currently formulating a standard for the VDM Specification Language – a standard that will also be proposed to the ISO. That standard will contain fully formal, mathematical definitions of the semantics of the BSI VDM SL.

For specifying distributed systems, we considered LOTOS [EVD89] as the most promising technique among the surveyed techniques. LOTOS has the merit (and takes the risks) of being based on relatively new and powerful theories (CCS and ACT ONE), which so far have mainly been confined to academic environments. The great promise of LOTOS lies in the fact that it allows as many levels of refinement as needed, through the use of two language operators: parallel composition and restriction. The LOTOS specifications that have been produced so far (see references in [BB89]) indicate that such quite complex systems can be specified with an intuitively appealing structure, and be relatively concise (when compared with other FDTs). One interesting feature of LOTOS is that there are four main styles for writing LOTOS specifications, as identified in [VSvS88], i.e., the monolithic style, the state-oriented style, the resource-oriented style and the constraint-oriented style. At the Tenth International Symposium on Protocol Specification, Testing and Verification (held in Canada on 12-15 June 1990), the fact that eight LOTOS papers were published shows that LOTOS has attracted a lot of research work.

SDL [BH89, FM89], a promising language for specifying real-time systems, is estimated to be known by more than 10,000 telecommunication engineers throughout the world. Key features of SDL which make it a success are: *support for graphics, hierarchical development and refinement*, *supporting tools*, and *standardization*.

STATEMATE [H+90] provides a promising approach for specifying the complex reactive systems. STATEMATE is a set of tools, with a heavy graphical orientation, intended for the specification, analysis, design, and documentation of large and complex reactive systems. Using STATEMATE, the system is described from three interrelated points of view, capturing *structure*, *functionality*, and *behavior*. These views are represented by three graphical languages, the most intricate of which is the language of *statecharts* [Har87], used to depict reactive behavior over time.

References

[A+85] A. Avižienis et al. The UCLA DEDIX system : A distributed testbed for multiple-version software. In *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, pages 126–134, June 1985.

[Alf85] M. Alford. SREM at the age of eight: The distributed computing design system. *IEEE Computer*, 18(4):36–46, April 1985.

[Avi85] A. Avižienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[B+88] R. Bloomfield et al., editors. *VDM'88: VDM - The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988. VDM Europe Symposium 1988.

[Bac90] R. J. R. Back. Refinement calculus, Part II: Parallel and reactive programs. In J. W. Bakker et al., editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.

[Bar89] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, SE-15(5):611–621, May 1989.

[BB89] T. Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk et al., editors, *The Formal Description Technique : LOTOS*, pages 23–73. North-Holland, 1989.

[BC89] T. Bolognesi and M. Caneve. Equivalence verification : Theory, algorithms and a tool. In P. H. J. van Eijk et al., editors, *The Formal Description Technique : LOTOS*, pages 303–326. North-Holland, 1989.

[BdRR90] J. W. Bakker, W.-P de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. Proc. REX Workshop, Mook, Netherlands, 1989.

[Bea88] S. Bear. Structuring for the VDM specification language. In R. Bloomfield et al., editors, *VDM'88: VDM - The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*, pages 2–25. Springer-Verlag, 1988.

[Ber86] D. M. Berry. Adding modularity and separate subsystem specification support to the formal development methodology (FDM). Technical Report SP-4361, System Development Corp., Santa Monica, CA, March 1986.

[Ber87] D. M. Berry. Towards a formal basis for the formal development method and the Ina Jo specification language. *IEEE Transactions on Software Engineering*, SE-13(2):184–200, February 1987.

[BH89] F. Belina and D. Hogrefe. The CCITT-Specification and Description Language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, 1988/89.

[BHL87] A. Birrell, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. *Proc. of the Eleventh ACM Symposium on Operating Systems Principles*, pages 94–102, 1987. ACM/SIGOPS.

[BHL90] D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors. *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. VDM Europe Symposium 1990.

[BJ87] D. Bjørner and C. B. Jones, editors. *VDM'87: VDM – A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987. VDM Europe Symposium 1987.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 51–63, 1984.

[BMPD87] D. Belsnes, B. Møller-Pedersen, and H. P. Dahle. Rationale and tutorial on OSDL: An object-oriented extension of SDL. In R. Saracco and P. A. J. Tilanus, editors, *SDL '87 : State of the Art and Future Trends*, pages 413–426. North-Holland, 1987.

[BW87] M. R. Barbacci and J. M. Wing. Specifying functional and timing behavior for real-time applications. In *Proc. of the Parallel Architecture and Languages Europe, Vol II*, pages 124–140. Lecture Notes in Computer Science, vol. 259, 1987.

[BW88] F. L. Bauer and M. Wirsing. Crypt-equivalent algebraic specifications. *Acta Informatica*, 25(2):111–153, February 1988.

[BWWH88] J. Billington, G. R. Wheeler, and M. C. Wilbur-Ham. PROTEAN : A high-level Petri net tool for the specification and verification of communication protocols. *IEEE Transactions on Software Engineering*, SE-14(3):301–316, March 1988.

[C+81] M. H. Cheheyl et al. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[CR83] J. E. Coolahan and N. Roussopoulos. Timing requirements for time-driven systems using augmented Petri Nets. *IEEE Transactions on Software Engineering*, SE-9(5):603–616, September 1983.

[CRS90] E. Cusack, S. Rudkin, and C. Smith. An object-oriented interpretation of LOTOS. In S. Vuong, editor, *Formal Description Techniques (Proc. FORTE'89)*. North-Holland, 1990.

[D+88] R. Duke et al. Protocol specification and verification using Z. In S. Aggarwal and K. Sabnani, editors, *Proc. 8th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 33–46, 1988.

[D+89] M. Diaz et al. *The Formal Description Technique Estelle*. North-Holland, 1989.

[Das85] B. Dasarathy. Timing constraints of real-time systems : Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[Dav88] A. M. Davis. A comparison of techniques for the specification of external system behavior. *Communications ACM*, 31(9):1098–1115, September 1988.

[DB87] P. Dembinski and S. Budkowski. Simulating Estelle specifications with time parameters. In H. Rudin and C. H. West, editors, *Proc. 7th IFIP Conference on Protocol Specification, Testing and Verification*, pages 265–279, 1987.

[DD90] D. Duke and R. Duke. Toward a semantics for Object-Z. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 244–261. Springer-Verlag, 1990.

[DeM78] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[DV89] M. Diaz and C. Vissers. SEDOS : Designing open distributed systems. *IEEE Software*, 6(5):24–33, November 1989.

[E+86] G. Estrin et al. SARA (System ARchitects Apprentice): Modeling,analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[EK85] S. Eckmann and R. A. Kemmerer. INATEST : An interactive environment for testing formal specifications. *ACM SIGSOFT Software Engineering Notes*, 10(4):17–18, August 1985. Third Workshop on Formal Verification.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*. Springer-Verlag, 1985.

[EVD89] P. V. Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. North-Holland, 1989.

[F+86] M. Fujita et al. Tokio : Logic programming language based on temporal logic and its complilation to prolog. In E. Shapiro, editor, *Proc. 3rd International Conference on Logic Programming*, pages 695–709. Lecture Notes in Computer Science, vol. 225, Springer-Verlag, 1986.

[FM79] L. Flon and J. Misra. A unified approach to the specification and verification of abstract data types. In *Proc. Specifications of reliable Software Conf.*, pages 162–169, Cambridge, MA, April 1979.

[FM89] Ove Færgemand and M. M. Marques, editors. *SDL '89 : The Language at Work.* North-Holland, 1989.

[Fut85] K. Futatsugi. Principles of OBJ2. In *ACM Symposium of Principles of Programming Languages*, pages 52–66, Arlington, Virginia, 1985.

[G+87] J. Goguen et al. OBJ as a language for concurrent programming. In Steven Kartashev and Svetlana Kartashev, editors, *Proc. Second International Supercomputing Conference, Volume I*, pages 195–198, St. Petersburg, FL, 1987.

[G+88] S. J. Garland et al. Verification of VLSI circuits using LP. In *Proc. IFIP WG 10.2, The Fusion of Hardware Design and Verification.* North Holland, 1988.

[G+89a] C. Ghezzi et al. A general way to put time in Petri Nets. *ACM SIGSOFT Software Engineering Notes*, 14(3):60–67, May 1989.

[G+89b] S. Graf et al. What are the limints of model checking methods for the verification of real life protocols ? In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 275–285. Lecture Notes in Computer Science, vol. 407, Springer-Verlag, 1989.

[Gal87] A. Galton, editor. *Temporal Logics and their Applications.* Academic Press, 1987.

[GCG90] C. P. Gerrard, D. Coleman, and R. M. Gallimore. Formal specification and design time testing. *IEEE Transactions on Software Engineering*, 16(1):1–12, January 1990.

[GG89] S. J. Garland and J. V. Guttag. An overview of LP : the Larch Prover. In *Proc. 3rd International Conference on Rewriting Techniques and Applications*, pages 137–151. Springer-Verlag, 1989.

[GGH90] S. J. Garland, J. V. Guttage, and J. J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering*, SE-16(9):1044–1057, September 1990.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, July 1985.

[GI90] R. Di Giovanni and P. L. Iachini. HOOD and Z for the development of complex software systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 262–289. Springer-Verlag, 1990.

[Gog84] J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gog88] J. A. Goguen. OBJ as theorem prover with applications to hardware verification. Technical Report SRI-CSL-88-4R2, SRI Computer Science Lab, August 1988.

[GP85] P. Gunningberg and B. Pehrson. Protocol and verification of a synchronization protocol for comparison of results. In *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, pages 172–177, Ann Arbor, MI, June 1985.

[GPG81] J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In *Formalizing Programming Concepts*, pages 292–309. Lecture Notes in Computer Science, vol. 107, Springer-Verlag, 1981.

[GT79] J. A. Goguen and J. J. Tardo. An introduction to OBJ : A language for writing and testing formal algebraic program specifications. In *Proc. Specifications of Reliable Software Conf.*, pages 170–189, Cambridge, MA, April 1979.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI Computer Science Lab, August 1988.

[H+90] D. Harel et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, SE-16(4):403–414, April 1990.

[Hal86] B. Halipern, editor. *Special Issue on Multiparadigm Languages and Environments.* IEEE Software, January 1986.

[Har87] D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hay87] I. Hayes, editor. *Specification Case Studies.* Prentice Hall, Inc., 1987.

[HH89] T. Hauge and Ø. Haugen. OST – An Object Oriented SDL Tool. In Ove Færgemand and M. M. Marques, editors, *SDL '89 : The Language at Work*, pages 179–188. North-Holland, 1989.

[HI88] S. Hekmatpour and D. Ince. *Software Prototyping, Formal Methods and VDM.* Addison-Wesley, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Inc., 1985.

[HR90] J. J. M. Hooman and W. P. De Roever. Design and verification in real-time distributed computing: an introduction to compositional methods. In E. Brinksma et al., editors, *Proc. 9th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 37–56, 1990.

[J+90] H.-M. Järvinen et al. Object-oriented specification of reactive systems. In *Proc. 12th International Conference on Software Engineering*, pages 63–71, Nice, France, March 1990.

[Jac88] D. Jackson. Composing data and process descriptions in the design of software systems. Technical Report MIT/LCS/TR-419, Department of Computer Science, MIT, May 1988.

[JG89] M. Joseph and A. Goswami. Formal description of realtime systems: A review. *Information and Software Technology*, 31(2):67–76, March 1989.

[JM86] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[Jon89] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, Inc., second edition, 1989.

[K+88a] M. R. Kappel et al. SAGEN user's guide. Technical Report IDA Paper P-2028, Institute for Defense Analyses, April 1988.

[K+88b] R. Koymans et al. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

[KdR83] R. Koymans and W.-P. de Roever. *Examples of a Real-time Temporal Logic Specification*, volume 207 of *Lecture Notes in Computer Science*, pages 231–252. Springer-Verlag, 1983.

[Kem85] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[Kem89] R. A. Kemmerer. Analyzing encryption protocols using formal verification techniques. Technical Report TRCS89-4, Department of Computer Science, University of California, Santa Barbara, Feburary 1989.

[Kra87] B. Kramer. SEGRAS – a formal and semigraphical language combining petri nets and abstract data types for the specification of distributed systems. In *Proc. 9th International Conference on Software Engineering*, pages 116–125, 1987.

[L+88] J. L. Linn et al. Strategic defense initiative architecture dataflow modeling technique, version 1.5. Technical Report IDA Paper P-2035, Institute for Defense Analyses, April 1988.

[LA89] P. G. Larsen and M. M. Arentoft. Towards a formal semantics of the BSI/VDM specification language. In *Information Processing 89*, pages 95–100. IFIP, 1989.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[LH85] D. C. Luckham and F. W. Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[LLS90] C. Lafontaine, Y. Ledru, and P.-Y. Schobbens. An experiment in formal software development: Usng the B theorem prover on a VDM case study. In *Proc. 12th International Conference on Software Engineering*, pages 34–43, Nice, France, March 1990.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness: The ethics of concurrent termination. In *Automata, Languages, and Programming*, pages 265–277. Lecture Notes in Computer Science, vol. 115, Springer-Verlag, 1981.

[M+89] José A. Mañas et al. The implementation of a specification language for OSI systems. In P. H. J. van Eijk et al., editors, *The Formal Description Technique : LOTOS*, pages 409–421. North-Holland, 1989.

[May89] T. Mayr. Specification of object-oriented systems in LOTOS. In K. Turner, editor, *Formal Description Techniques (Proc. FORTE'88)*, pages 107–119. North-Holland, 1989.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mok85] A. K. Mok. SARTOR – a design environment for real-time systems. In *Proc. of 9th IEEE COMPSAC*, pages 174–181, October 1985.

[Mor89] R. Moretti. SDL and object oriented design: A way for producing quality software. In Ove Færgemand and M. M. Marques, editors, *SDL '89 : The Language at Work*, pages 387–394. North-Holland, 1989.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice Hall, Inc., 1990.

[Mos85] B. C. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, February 1985.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[MR87] C. Morgan and K. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.

[NA90] D. New and P. D. Amer. Adding graphics and animation to Estelle. *Information and Software Technology*, 32(2):149–161, April 1990.

[NF89] A. T. Nakagawa and K. Futatsugi. Stepwise refinement process with modularity : an algebraic approach. In *Proc. 11th International Conference on Software Engineering*, pages 166–177, 1989.

[NG88] M. Nielsen and C. George. The RAISE language, method and tools. In R. Bloomfield et al., editors, *VDM'88: VDM - The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*, pages 376–405. Springer-Verlag, 1988.

[Ost89a] J. S. Ostroff. Automated verification of timed transition models. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 247–256. Lecture Notes in Computer Science, vol. 407, Springer-Verlag, 1989.

[Ost89b] J. S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press LTD., 1989.

[PH88] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98. Lecture Notes in Computer Science, vol. 331, Springer-Verlag, 1988.

[Pnu86a] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems. In J. W. deBakker et al., editors, *Current Trends in Concurrency*, pages 510–584. Lecture Notes in Computer Science, vol. 244, Springer-Verlag, 1986.

[Pnu86b] A. Pnueli. Specification and development of reactive systems. In H. J. Kugler, editor, *Information Processing 86*, pages 845–858, 1986.

[QAF90] J. Quemada, A. Azcorra, and D. Frutos. A timed calculus for LOTOS. In S. Vuong, editor, *Formal Description Techniques (Proc. FORTE'89)*. North-Holland, 1990.

[QF87] J. Quemada and A. Fernandez. Introduction of quantitative relative time into LOTOS. In H. Rudin and C. H. West, editors, *Proc. 7th IFIP Conference on Protocol Specification, Testing and Verification*, pages 105–121, 1987.

[QS83] J. P. Queille and J. Sifakis. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[R+87] J. Richier et al. Verification in XESAR of the sliding window protocol. In H. Rudin and C. H. West, editors, *Proc. 7th IFIP Conference on Protocol Specification, Testing and Verification*, pages 235–248, 1987.

[RC89] Jean-Luc Richard and T. Claes. A generator of c-code for Estelle. In M. Diaz et al., editors, *The Formal Description Technique : Estelle*, pages 397–420. North-Holland, 1989.

[Rom85] G.-C. Roman. A taxonomy of current issues in requirements engineering. *IEEE Computer*, 18(4):14–22, April 1985.

[S+82] C. A. Sunshine et al. Specification and verification of communication protocols in AFFIRM using state transition models. *IEEE Transactions on Software Engineering*, SE-8(5):460–489, September 1982.

[Sch82] D. Schwabe. *Formal Techniques for the Specification and Verification of Protocols.* PhD thesis, UCLA Computer Science Department, Los Angeles, September 1982.

[SG88] R. Sijelmassi and P. Gaudette. An object-oriented model for Estelle. In K. Turner, editor, *Formal Description Techniques (Proc. FORTE'88)*, pages 91–105. North-Holland, 1988.

[Shu89] R. N. Shutt. A rigorous development strategy using the OBJ specification language and the MALPAS program analysis tool. In C. Ghezzi and J. A. McDermid, editors, *ESEC '89, 2nd European Software Engineering Conference*, pages 260–291. Lecture Notes in Computer Science, vol. 387, Springer-Verlag, 1989.

[SKG87] M. Sredniawa, B. Kakol, and P. Gumulinski. SDL in performance evaluation. In R. Saracco and P. A. J. Tilanus, editors, *SDL '87 : State of the Art and Future Trends*, pages 127–135. North-Holland, 1987.

[Sol82] J. Solomon. Specification-to-code correlation. In *Proc. of the 1982 IEEE Symposium on Security and Privacy*, Oakland, CA, April 1982.

[SSR89] R. Saracco, J. R. W. Smith, and R. Reed. *Telecommunications Systems Engineering using SDL.* North-Holland, 1989.

[Ter87] A. Teruel. On mixing formal specification styles. In *Fourth Internatonal Workshop on Software Specification and Design*, pages 28–33, Monterey, CA, April 1987.

[TH77] D. Teichroew and E. A. Hershey. PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on Software Engineering*, 3(1):41–48, January 1977.

[VB89] Pieter H. A. Venemans and Rob A. Beukers. An experiment with algebraic reduction. In Ove Færgemand and M. M. Marques, editors, *SDL '89 : The Language at Work*, pages 325–334. North-Holland, 1989.

[vHTZ90] W. H. P. van Hulzen, P. A. J. Tilanus, and H. Zuidweg. LOTOS extended with clocks. In S. Vuong, editor, *Formal Description Techniques (Proc. FORTE'89).* North-Holland, 1990.

[VSvS88] C. A. Vissers, G. Scollo, and M. van Sinderen. Architectural and specification style in formal description of distributed systems. In S. Aggarwal and K. Sabnani, editors, *Proc. 8th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 189–204, 1988.

[WG89] J. M. Wing and C. Gong. Machine-assisted proofs of properties of Avalon programs. Technical Report CMU-CS-89-172, Department of Computer Science, Carnegie Mellon University, August 1989.

[Win87] J. M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[WN89] J. M. Wing and M. R. Nixon. Extending Ina Jo with temporal logic. *IEEE Transactions on Software Engineering*, SE-15(2):181–197, February 1989.

[Wol86] M. Wolczko. *Typesetting VDM with* LaTeX. 1986.

[Woo89] J. C. P. Woodcock. Calculating properties of Z specifications. *ACM SIGSOFT Software Engineering Notes*, 14(5):43–54, July 1989.

[Wu88] Chi-Sharn Wu. Final report on the formal specification of DEDIX. Internal Report of UCLA Dependable Computing and Fault-Tolerant Systems research group, October 1988.

[Wu90] Chi-Sharn Wu. *Formal Specification techniques and Their Applications in N-Version Programming.* PhD thesis, UCLA Computer Science Department, Los Angeles, October 1990.

[Z+89] M. Zorič et al. Tool set development and the use of SDL. In Ove Færgemand and M. M. Marques, editors, *SDL '89 : The Language at Work*, pages 77–86. North-Holland, 1989.

[Zav84] P. Zave. The operational versus the conventional approach to software development. *Communications ACM*, 27(2):104–118, February 1984.

[Zav89] P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25, September 1989.

[Zic90] J. J. Zic. Some thoughts on communication system performance specification. In D. B. Hoang and E. Chew, editors, *Proc. Open Distributed Processing Workshop*, Sydney, January 1990.

[ZJ89] P. Zave and D. Jackson. Practical specification techniques for control-oriented systems. In *Information Proccessing 89, Proc. of the IFIP 11th World Computer Congress*, pages 83–88, San Francisco, USA, August 28 – September 1 1989.

[ZS86] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, SE-12(2):312–325, February 1986.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.