REIHE INFORMATIK
3/93

Efficient Implementation of Estelle Specifications

B. Hofmann und W. Effelsberg
Universität Mannheim
Seminargebäude A5
6800 Mannheim

Efficient Implementation of Estelle Specifications

Bernd Hofmann and Wolfgang Effelsberg
University of Mannheim
P.O. Box 10 34 62
D-6800 Mannheim 1
Germany
Phone: +49 (6 21) 2 92 - 50 53
Fax: +49 (6 21) 2 92 - 57 45
hofmann@pi4.informatik.uni-mannheim.de

Abstract

Efficient implementation of communication software is of critical importance for high-speed networks. We analyze performance bottlenecks in existing implementations and propose two techniques for improvements: The first exploits parallelism not only in the actions of the FSMs, but also in the runtime system of the protocol stack. The second integrates adjacent layers leading to considerable savings in inter-layer interface handling and in the number of transitions occurring in the FSMs. Both techniques are discussed in the context of OSI upper layers, and are based on protocol specification in Estelle.

1 Introduction

Existing protocol suites such as the ISO/OSI [ISO84] or the INTERNET protocols [Com88] were designed with relatively slow networks in mind. The end systems were fast enough to process complex protocols because the data transmission time was long compared to the time needed for protocol execution.

Meanwhile, fast transmission media based on fiber optics are available, and the end systems are now too slow for high performance communication. Communication software has become the major bottleneck in high speed networks [Svo89b, CT90]. Efficient implementation of the protocol stack is of crucial importance for the networking future.

For specification purposes, three formal description techniques – Estelle, SDL and LOTOS – were standardized [ISO89, CCI87, ISO87]. They improved the correctness of specifications by avoiding ambiguities and by enabling formal verification. In addition, they allow semiautomatic code generation.

This has several advantages: The code can be maintained more easily, since the system is specified in an abstract, problem-oriented language. It is also much easier to port an implementation to another system. But one of the major problems is the performance of implementations produced automatically from a formal specification.

Existing code generators (such as the NIST Estelle-C-compiler [FHSW89, NIS89]) were made to easily get rapid prototypes for simulation purposes. Executable specifications lead to a better understanding of protocol behaviour. Since performance aspects are not essential for a simulation, these Estelle tools are designed for validation rather than for the generation of efficient code for high performance implementations.

Our measurements show that 60–80% of the runtime of automatically generated implementations are spent in the runtime system of the compiler. This gives rise to hopes that much more efficient implementations can be derived by code generators if the runtime system can be improved.

In this paper, we describe a more efficient runtime system for Estelle which avoids some of the weak points of the original runtime system. It makes use of parallelism and therefore is intended to run under the operating system MACH [Tan92].

Besides improving the runtime system there is another possibility to achieve more efficient implementations. Almost always layered communication subsystems are implemented according to the layered structure of the specification. This implies communication overhead, especially when functions already available in lower layers have to be passed through the layers in between to make them accessible for the application. In particular, this applies to the upper layers; they have relatively simple protocol machines and provide complex functions.

We describe a methodology how to integrate adjacent layers in Estelle. Thus, state transitions as well as communication can be saved leading to better performance.

This paper is organized as follows: Section 2 describes why implementations generated with the NIST compiler are inefficient. Results of runtime measurements are presented and possibilities of performance improvements are shown. In section 3, the parallel runtime system is described. The integration of Estelle modules is shown in section 4. Section 5 concludes the paper.

2 The NIST Estelle-C-Compiler

Formal description techniques are now widely used for the specification and validation of standardized communication protocols. They have a well-defined syntax and semantics and allow the precise and unambiguous definition of protocol behaviour.

For practical purposes the formal description techniques SDL and Estelle are most popular. Both are based on the *Extended Finite State Machine* model (in the following: FSM). Since our work concentrates on upper layers (i. e. layers 5 to 7 of the Reference Modell) we decided to use Estelle; a large number of upper layer specifications is already available in Estelle, including Session and Presentation layers, and more are being developed. The implementation techniques described in this paper are based on Estelle input, and all protocols described in Estelle can thus benefit immediately from the performance improvements.

An Estelle specification describes communicating FSMs which are connected through bidirectional buffering channels. FSMs are represented by active modules, which can be defined in a hierarchical manner together with passive ones. Passive modules only define interfaces and hide underlying modules according to the block structure of PASCAL which Estelle is derived from.

There is a broad variety of tools for Estelle specifications. Besides those for e. g. deadlock recognition, state minimization, test case derivation, simulation, verification there is a range of code generators. In the following, we refer to the NIST compiler exclusively.

The main purpose of the NIST compiler is to make an Estelle specification **executable**. This is done by generating C code which implements the FSM, predicates and actions of the protocol. An executable specification allows the protocol designer to "play" with the protocol machines, to gain experience and find bugs. In most cases the designer will generate two FSM instances from the same specification and execute them back-to-back.

We have extended the NIST compiler to produce **implementation code**. Compared to an executable specification, implementation code for a layer must be more efficient, and must interface with adjacent layers and the operating system. But of course the FSM is the same in both cases, and since our implementation code is based on a thoroughly validated specification it is correct, well structured and portable. The main concern with this approach is the efficiency of the generated code.

The NIST Compiler mainly generates two C-files (besides two header files) for each module of an Estelle specification. One file implements the predicates and actions specified by the transitions of the underlying FSM. The other one is a template for user-defined functions and types etc.

The FSMs are represented internally in the form of state tables and are mapped to data structures. A processing unit is missing. Table processing is implemented by the runtime environment of the compiler. It mainly consists of a scheduler and procedures for enqueueing events, creating new module entities, evaluating predicates etc. Normally, the scheduler is called in an endless loop, watching all queues for incoming events and looking up the actions to be executed in the state tables. If several transitions are possible for a given event, predicates (derived from the *when*-clauses in the Estelle specification) have to be evaluated to determine the transition to be executed.

The interesting question now is: How much time is spent by the actions defined by the FSMs (i. e. the action blocks of the transitions) and how much by the scheduler? To answer this question, we implemented a tool for runtime measurement of compiled Estelle specifications. The NIST compiler was modified to produce additional code which writes a timestamp on disk at the beginning and at the end of each generated subroutine. In addition, all runtime routines were also modified in this way [Lie92].

Thus, runtime measurement is easily possible for existing specifications by only recompiling them with the modified compiler. No changes are necessary neither to the Estelle specification nor to the handcoded parts. Of course, the writing of the timestamps itself influences the measurement. So, no exact execution times can be taken, but since all routines are influenced in the same way, a comparison of the runtime of different routines or different versions of the same specification is possible.

As an example, we used Estelle specifications of the ISO Presentation and Session layers. Both were extracted from a specification described in [Web91] which in turn was derived from [FLGL89] and [MM89] resp. The Presentation layer consists of the functional unit *kernel* without ASN.1-encoding/decoding, the Session layer comprises the *Basic Combined Subset*.

Different constellations were measured: Presentation layer alone, Session layer alone, then the combination of both within *one* Estelle specification. The results of the processing of a P-CONNECT.request and a S-CONNECT.request are shown in Tab. 1.

Two facts can be observed from these results:

1. The total execution time (last column) of both events in the combination of Presentation and Session layer (180 ms) lasts longer than the sum of the times in the single layers (150 ms).
2. The scheduler needs much more time than the execution of the actions of the FSMs.

Ad 1: As described above, the runtime system must periodically determine the fireable transitions. This is done by the scheduler, which examines the data structure containing the state tables. The data structure is organized as a tree reflecting the hierarchy of the modules. Since this tree gets more complex with an increasing number of modules, the access time to the transitions increases also.

Ad 2: The fact that approx. 60-80% of the execution time are spent by the scheduler shows that it is much more important to improve the efficiency of the runtime system than to speed up the FSMs. If we assume in the best case a portion of only 60% for the scheduler the speedup gained by only parallelizing the FSMs is limited to 1.67 ($= \frac{1}{60\%}$). This is a consequence of *Amdahl's law*[Qui88]: If f is the inherently sequential fraction of a computation to be solved by p processors, then the parallelization speedup s is limited according to

$$s \leq \frac{1}{f + \frac{1-f}{p}}$$

Table 1: Execution times of P-CONNECT.request and S-CONNECT.request (in ms)

	Actions of FSM	Scheduler	Σ
Session alone	20	80	100
Presentation alone	20	30	50
Σ	40	110	150
Combined Pres. & Session	40	140	180

The maximum speedup s_{max} can be estimated by letting p grow to infinity:

$$s_{max} = \lim_{p \to \infty} \frac{1}{f + \frac{1-f}{p}} = \frac{1}{f}$$

Two conclusions can be drawn from our observations:

1. The main problem opposing an efficient implementation is the runtime system. Hence, the main emphasis should be put on speeding up the runtime system before trying to speed up the FSMs.
2. The scheduler should process small and simple data structures.

How these postulations can be achieved is described in the next section.

3 Parallelism

One promising approach for improving the performance of a system is the use of parallelism [HE92, HEHK92, RDF89, UD90, Zit92]. For the implementation of Estelle specifications using the NIST compiler, there are two ways of using parallelism:

1. **Parallel execution of the runtime system**. The determination of executable transitions in one (active) module depends only on the elements of the input queues and the value of the state variable of this module and is therefore independent of all other modules, i. e. can be done in parallel for all modules.

   Then each module has its own scheduler. All schedulers operate in parallel. Each scheduler also operates much more efficiently than the original scheduler since the data structures to be searched are much smaller.

2. **Parallel execution of the specification**. Another potential of parallelism is provided by the language Estelle itself: It is possible to specify parallelism explicitly by decomposing the parallel tasks into separate Estelle modules.

   Estelle defines precisely which modules can be processed in parallel. For this purpose, active modules are attributed with the keywords *process*, *activity*, *system process* or *system activity*. Modules attributed as *system process* or *system activity* can be processed concurrently with other modules. Child modules of a *process* module can be active at the same time whereas child modules of an *activity* module must be processed in a sequential way. Modules in an ancestor/descendant relationship must not execute in parallel; a parent module takes precedence over its child modules.

Before we can describe how an Estelle specification can be implemented on a parallel computer, we give a brief introduction into parallel hardware.

There is a wide range of parallel computer architectures, and different classifications have been proposed [Fly66, HB84, Qui88]. The most important criteria are the use of instruction and data stream (SIMD, MIMD) and the organization of the memory (shared vs. local). The optimal architecture depends on the problem to be solved; there is no general-purpose parallel computer.

For our purposes it is clear that SIMD architectures (Single Instruction Multiple Data) are inapplicable. They are well suited for problems which need the same operations on all data objects simultaneously such as vector or matrix manipulation. Since an Estelle specification never describes a set of identical FSMs, a MIMD structure (Multiple Instruction Multiple Data) is appropriate.

The question of the memory structure needs further discussion. Shared memory can avoid unnecessary copying of data. On the other hand, if all the data is stored in shared memory, and all communication and synchronization is done via shared buffers and semaphores, memory access becomes the bottleneck. As a consequence, a combination of both local and shared memory is desirable for the parallel implementation of Estelle specifications.

Multiprocessor systems are best supported by a multithread operating system such as MACH [Tan92, Loe92]. It allows to create light-weight processes (so-called *threads*) executing in the same address space defined by *tasks*. Thus threads can share variables whereas tasks don't have memory in common and have to communicate via message passing.

In addition, MACH is part of the kernel of the OSF operating system *OSF/1* [OSF91] and therefore a general platform for multiprocessor machines from different vendors. Applications developed under MACH are easy to port and not bound to specific hardware.

An Estelle specification can now be implemented under MACH as follows:

- All active modules are implemented as threads.
- Each *system* module is mapped to a task under which all descendant modules are running as threads. *System* modules describe autonomous, closed subsystems of a specification which communicate over message passing. Since they don't share variables with other modules, they can be implemented as tasks.

Compared with the original NIST approach the runtime system now has completely changed: There is no central scheduler any more; each module has its own scheduler. Thus each module executes in a thread in parallel with other modules. The concurrency restrictions as pointed out in the last section (ancestor/descendant conflict, activities) are guaranteed by semaphor variables.

We are currently adapting the Estelle runtime system to the MACH multithread architecture. This leads to a library of functions which can be linked to the generated code. In the moment, we are also modifying the NIST code generator itself to map Estelle modules to threads and tasks according to the rules stated above. This will allow the automatic generation of an implementation from an Estelle specification under MACH.

4 Integration

In the previous section we have proposed a parallel implementation of Estelle modules for performance improvement. Another approach to more efficient implementation is the integration of adjacent layers. Layered implementation is considered to be an obstacle to higher performance [Svo89a, Haa90].

Traditionally, there are two methods of implementing a layered communication subsystem: the *server model* and the *activity thread* model [Svo89a]. In both cases, the interfaces between the layers are kept, and inter-layer communication occurs according to the standards, either by interprocess communication or by procedure calls.

While this may be acceptable for lower layers where natural boundaries between a user process and the operating system or the host computer and an adapter card have to be observed, strict layering introduces unnecessary communication overhead in the application-oriented upper layers. The transport service offers a one-to-one-connection regardless of the underlying network. In all layers above, this communication structure remains unchanged. They only add more functionality (i. e. dialogue management, synchronization points, encoding/decoding of different data representations etc.) to the transport service.

Some of these functions (e. g. normal data transfer) are available in layer four already. Because of the layered structure, they must be handed over layer by layer, up to the application. This passthrough mechanism produces both additional transitions as well as communication and synchronization overhead which can be avoided by the integration of upper layers, i. e. the integration of the FSMs of these layers into one common FSM.

In principle, an integrated FSM is the product automaton of the FSMs to be integrated. However the product automaton can be simplified as we will show below.

Definition 1 (Product automaton) *Let $A_j = (S_j, I_j, O_j, \delta_j, \lambda_j, r_j)$, $j \in \{1,2\}$ be two finite state automata with the state sets S_j, the input alphabets I_j with $I_1 \cap I_2 = \emptyset$, the output alphabets O_j, the transition functions $\delta_j : S_j \times I_j \to S_j$, the output functions $\lambda_j : S_j \times I_j \to O_j$ and the initial states r_j. Then we define the product automaton as*

$$A_1 \times A_2 := (S_1 \times S_2, I_1 \cup I_2, O_1 \cup O_2, \delta, \lambda, (r_1, r_2))$$

with

$$\delta : (S_1 \times S_2) \times (I_1 \cup I_2) \to (S_1 \times S_2),$$

$$\delta((s_1, s_2), i) := \begin{cases} (\delta_1(s_1, i), s_2) & : \quad i \in I_1 \\ (s_1, \delta_2(s_2, i)) & : \quad i \in I_2 \end{cases}$$

and

$$\lambda : (S_1 \times S_2) \times (I_1 \cup I_2) \to (O_1 \cup O_2),$$

$$\lambda((s_1, s_2), i) := \begin{cases} \lambda_1(s_1, i) & : \quad i \in I_1 \\ \lambda_2(s_2, i) & : \quad i \in I_2 \end{cases} .$$

If the two automata are protocol machines of adjacent layers, a transition of one automaton can directly cause a transition of the other one. This occurs if the output element of the first transition and the input element of the second transition are identical. These two transitions – which are connected by a state in the product automaton – can be replaced by one transition:

Lemma 1 *Let A_1, A_2 be two finite state automata with $O_1 \cap I_2 \neq \emptyset$. Let further be*

$$s_1 \overset{i_1/\lambda_1(s_1,i_1)}{\longrightarrow} \delta_1(s_1, i_1)$$

a transition of A_1 and

$$s_2 \overset{i_2/\lambda_2(s_2,i_2)}{\longrightarrow} \delta_2(s_2, i_2)$$

a transition of A_2, then

$$(s_1, s_2) \overset{i_1/\lambda_1(s_1,i_1)}{\longrightarrow} (\delta_1(s_1, i_1), s_2) \overset{i_2/\lambda_2(s_2,i_2)}{\longrightarrow} (\delta_1(s_1, i_1), \delta_2(s_2, i_2))$$

is a sequence of transitions of $A_1 \times A_2$.

If $i_2 = \lambda_1(s_1, i_1) \in O_1 \cap I_2$, this sequence can be replaced by the transition

$$(s_1, s_2) \overset{i_1/\lambda_2(s_2,\lambda_1(s_1,i_1))}{\longrightarrow} (\delta_1(s_1, i_1), \delta_2(s_2, i_2)).$$

In addition there are some transitions in the product automaton which can never be executed: the transitions triggered by events occuring at the layer interface between the two original FSMs. Since these events are obsolete in the product automaton, the corresponding transitions can be dropped.

We will use the ISO Presentation and Session layers to clarify this point. At connection establishment, the Presentation user sends a P-CONNECT.request to the Presentation protocol machine. That in turn sends an S-CONNECT.request to the Session protocol machine which creates a T-CONNECT.request for the transport layer (see Fig. 1).

In the product automaton these transitions correspond to the part shown in Fig. 2.

Since S-CONNECT.request is both an element of the input alphabet of the Session protocol machine and an element of the output alphabet of the Presentation protocol machine, transitions 1 and 2 can be replaced according to lemma 1 (see Fig. 3).

Furthermore, S-CONNECT.request is an event of the “inner” interface and thus obsolete. Since this event cannot be received by the integrated automaton, transition 3 in Fig. 2 will never be executed and can therefore be dropped. Because of that, state *idle/await T-CONcnf* is unreachable and can be removed as well as transition 4. So, Figure 3 shows the final situation. The FSM integration has reduced two transitions to one and saved one synchronization/communication event.

Of course, this is only possible, if no erroneous behaviour is introduced by the integration, i. e. the input/output behavior of the integrated layers remains unchanged. Because of the

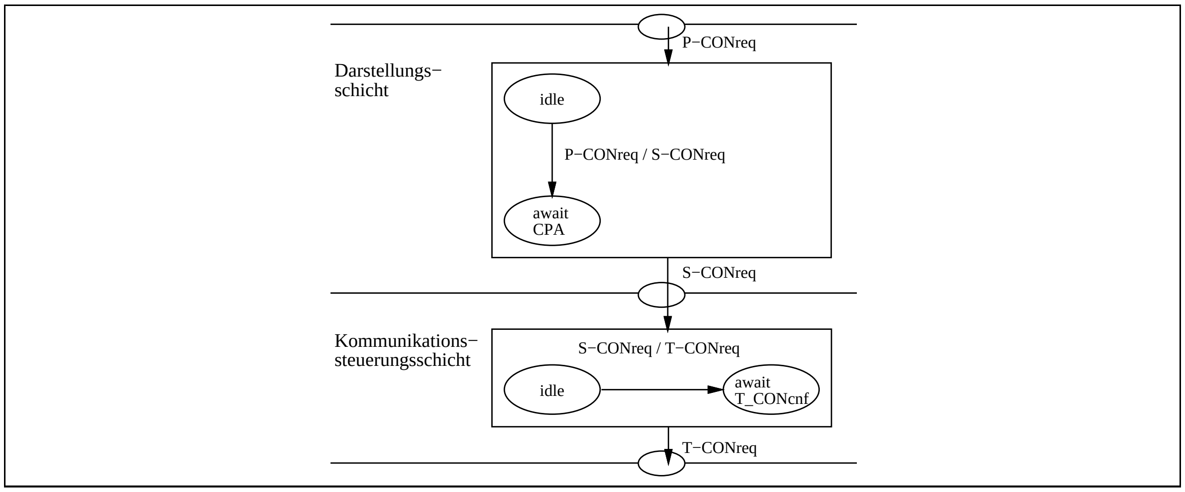

Figure 1: Connection establishment with separate Presentation and Session layers

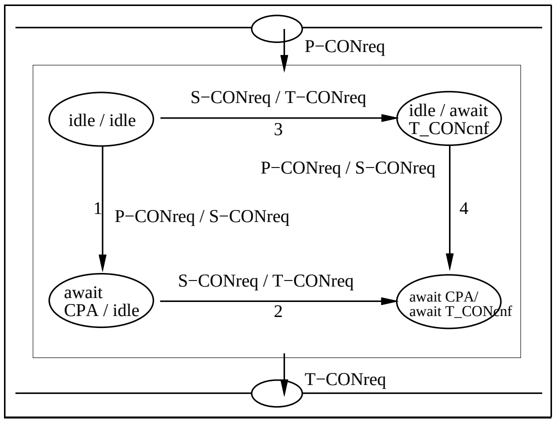

Figure 2: Connection establishment in the product automaton of Presentation and Session layers

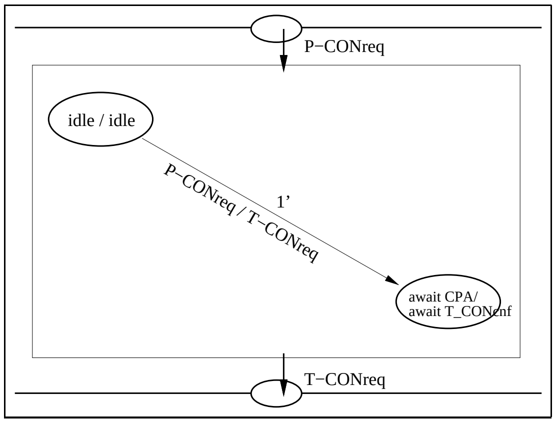

Figure 3: Connection establishment in the integrated automaton

construction method of the integrated automaton it should be clear that the integrated automaton and the combination of the original automata are equivalent in the sense of automata theory, i. e. always produce identical output under the same sequence of input events [Sal69].

At the moment we are working on the integration of Presentation and Session layers. It turns out that there are many passthrough services in these layers, similar to the CONNECT.request service. Therefore layer integration will lead to a considerable simplification of internal structure of upper layer communication software. Since our specification is not quite complete yet, exact measurements are not possible. But first measurements show that a speedup of at least a factor of 2 can be expected. After the encouraging experience we intend to also include ACSE [ISO88] into the integrated upper layer FSM.

5 Conclusion and Outlook

We have analyzed the typical bottlenecks in implementation code derived automatically from an Estelle specification. It is our goal to generate code running (almost) as efficiently as hand-written code.

Measurements have shown that up to 80% of the execution time was spent in the scheduling component of the FSM implementations. Based on these results we propose to exploit parallelism not only for the predicates and actions of an FSM but also in the runtime system. This is done by mapping the Estelle modules to operating system threads which can be executed on different processors. Each modules contains its own scheduler.

At the moment, we are working on the parallel runtime system as well as on the necessary modifications to the NIST compiler.

Inter-layer interfaces are another major source of inefficiency in an implementation. Whereas the layered approach is well suited for design and specification purposes it is inappropriate to maintain all layer interfaces in an upper layer implementation. We have presented a method for layer integration. We first compute a product automaton of two adjacent layer entities and then remove all events, transitions and states related to the former inter-layer interface. This leads to a much smaller integrated automaton. Based on first experience with ISO Session and Presentation layers we are currently working on a tool automating the layer integration process.

We believe that the combination of these two methods will enable the automatic generation of efficient code directly from Estelle specifications.

References

[CCI87] CCITT SG X: Recommendation Z.100: Specification and description language SDL. Contribution Com X-R15-E, 1987.

[Com88] D. Comer. *Internetworking with TCP/IP*. Prentice-Hall, Englewood Cliffs, 1988.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90 Symposium Communication Architectures & Protocols*, pages 200–208, Philadelphia, September 1990.

[FHSW89] J. Favreau, M. Hobbs, B. Strausser, and A. Weinstein. User guide for the NIST prototype compiler for Estelle. Technical Report No. ICST/SNA - 87/3, Institute for Computer Science and Technology, National Institute of Standards and Technology, September 1989.

[FLGL89] J.-P. Favreau, R. J. Linn, J. Gargulio, and J. Lindley. A test system for implementations of FTAM/FTP gateways. National Institute for Standards and Technology, USA, 1989.

[Fly66] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[Haa90] Z. Haas. A communication architecture for high-speed networking. In *Infocom '90*, pages 433–441, 1990.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[HE92] B. Hofmann and W. Effelsberg. Generating parallel code from Estelle specifications. In D. Hogrefe, editor, *Formale Beschreibungstechniken für verteilte Systeme.* Springer Verlag, 1992.

[HEHK92] B. Hofmann, W. Effelsberg, T. Held, and H. König. On the parallel implementation of OSI protocols. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, Arizona, February 1992.

[ISO84] Information processing systems – Open Systems Interconnection – Basic Reference Model. International Standard ISO 7498, 1984.

[ISO87] Information processing systems – Open Systems Interconnection – LOTOS: Language for the temporal ordering specification of observational behaviour. International Standard ISO 8807, 1987.

[ISO88] Information processing systems – Open Systems Interconnection – protocol specification for the Association Control Service Element. International Standard ISO 8650, 1988.

[ISO89] Information processing systems – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model. International Standard ISO 9074, 1989.

[Lie92] R. Lienhart. Erweiterung des NIST-Estelle-Compilers zur Analyse des Laufzeitverhaltens übersetzter Module. Internal report, Lehrstuhl für Praktische Informatik IV, Universität Mannheim, 1992 (in German).

[Loe92] K. Loepere. Mach 3 kernel principles. Technical report, Open Software Foundation and Carnegie Mellon University, 1992.

[MM89] P. Mondain-Monval. Estelle description of the ISO session protocoll. In M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azema, and Vijaya Chari, editors, *The Formal Description Technique Estelle*, pages 229–269. Elsevier Science Publishers B. V., Amsterdam, 1989.

[NIS89] Internals guide for the NIST prototype compiler for Estelle. Report No. ICST/SNA - 87/4, U.S. Department of Commerce, National Institute of Standards and Technology, February 1989.

[OSF91] A guide to OSF/1: A technical synopsis. O'Reilly & Associates, Inc., 1991.

[Qui88] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1988.

[RDF89] M. Rupprecht, M. Dresen, and F. Fehlau. A high throughput LAN-controller for high bit rate systems. In *Proceedings EFOC/LAN 89*, Amsterdam, Juli 1989.

[Sal69] A. Salomaa. *Theory of Automata.* Pergamon Press, Oxford, 1969.

[Svo89a] L. Svobodova. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, September 1989.

[Svo89b] L. Svobodova. Measured performance of transport service in LANs. *Computer Networks and ISDN Systems*, 18(1):31–45, 1989.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, 1992.

[UD90] R. Ulrich and H. Dietsch. A Transputer-based communication controller for FDDI stations. In *EFOC/LAN 90*, München, Juni 1990.

[Web91] S. Weber. Spezifikation und Implementation eines Datenkommunikationssystems mit Estelle. Master's thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, April 1991 (in German).

[Zit92] M. Zitterbart. Parallel protocol implementations on Transputers – experiences with OSI TP4, OSI CLNP, and XTP. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, Arizona, February 1992.