REIHE INFORMATIK
5/93

An Estelle Compiler for Multiprocessor Platforms

S. Fischer und B. Hofmann
Universität Mannheim
Seminargebäude A5
68131 Mannheim

An Estelle Compiler for Multiprocessor Platforms

Stefan Fischer and Bernd Hofmann

University of Mannheim, Praktische Informatik IV, P.O. Box 10 34 62,
D-68131 Mannheim, Germany

[fischer, hofmann]@pi4.informatik.uni-mannheim.de

Efficient implementation of communication software is of critical importance for high-speed networks. Parallelism can improve the runtime performance of implementations gained by code generation. Therefore, we have modified an existing Estelle compiler to run under OSF/1. It exploits parallelism not only in the actions of the FSMs, but also in the runtime system of the protocol stack.

Keyword Codes: C.2.2; D.1.3; D.2.2
Keywords: Computer–Communication Networks, Network Protocols, Protocol Implementation; Programming Techniques, Concurrent Programming; Software Engineering, Tools and Techniques

1. Introduction

Existing protocol suites such as the ISO/OSI [19] or the INTERNET protocols [7] 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.

Given the fast transmission media based on fiber optics that are now available, current end systems are too slow for high performance communication. Communication software has become the major bottleneck in high speed networks [6, 36]. 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 [5, 22, 25]. They improve the correctness of specifications by avoiding ambiguities and by enabling formal verification. In addition, they allow semiautomatic code generation.

This technique 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 were made to easily get rapid prototypes for simulation purposes [2, 3, 10, 33, 34, 38]. Executable specifications lead to a better understanding of protocol behaviour. Since performance aspects are not essential for such a simulation, existing Estelle tools are designed for validation rather than for the generation of efficient code for high performance implementations.

A considerable amount of the runtime of automatically generated implementations is

spent in the runtime system. This gives rise to hopes that much more efficient implementations can be created by code generators if the runtime system can be improved, especially by the use of parallelism.

At the moment, there are different approaches to improve the performance of protocol execution by parallelism [1, 4, 14–16, 31]. But they either need special hardware or do not use formal description techniques. There do exist code generators for deriving parallel implementations but they mainly were intended for simulation and validation purposes [30, 35] or an intermediate technique between simulation and prototype implementation called *experimentation* [26]. In this paper, we describe a more efficient runtime system for one of them, the PET/DINGO, which avoids some of the weak points of the original runtime system [12]. It makes use of parallelism and is intended to run under the operating system OSF/1[1] [29].

This paper is organized as follows: Section 2 describes the different forms of parallelism resulting from Estelle specifications. Sections 3 and 4 show how this parallelism can be mapped to OSF/1 structures, focussing on parallel decomposition and synchronization aspects. Section 5 presents some results concerning the speedup we obtained with the compiler. Section 6 concludes the paper and gives an outlook on further work on the compiler.

2. Parallel Execution of Estelle Specifications

The use of a code generator for implementing Estelle specifications allows two sorts of parallelism:

1. Parallel execution of the protocol machines, and
2. Parallel execution of the runtime system.

Ad 1: Estelle itself provides syntactical constructs for expressing parallelism: It is possible to specify parallelism explicitly by decomposing the parallel tasks into separate Estelle modules.

Estelle precisely defines 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 sequentially. Modules in an ancestor/descendant relationship must not execute in parallel; a parent module takes precedence over its child modules.

However, there are at least two points on the way from specification to implementation where parallelism can get lost:

1. It is the specifier who decides which module attributes are used. Thus, it is possible that parallel activities are specified sequentially because the parallelism was not recognized by him or her. This hidden parallelism cannot be detected in a fully automatic way due to the need of background knowledge [17]. To avoid this problem, careful training of the specifiers is necessary.

[1] OSF/1 is an operating system supporting multiprocessor machines based on Mach [37]

2. Even if all inherent parallelism of a specification is described by syntactical elements, all or part of it can be ignored by a manual implementation. This might be caused by the lack of experience of the implementor as well as by the fact that the target machine has fewer processors than needed.

   This can be avoided by a code generator. Of course, it cannot solve the problem of too few processors, but since it maps parallel Estelle elements to parallel processors according to general rules, the described parallelism is preserved in the implementation.

Another benefit of code generators is the possibility of being able to describe parallelism in a problem–oriented, high level language independently of the hardware.

Ad 2: An implementation gained from pure code generation cannot be executed as it is. Finite State Machines (FSMs) are passive machines which only react on stimulations received from their environments. Since code generation can only map FSMs to corresponding pieces of code and data structures, code generation requires a *runtime system* as well. Besides routines for queue and buffer management and timers it comprises in particular a *scheduler*. This routine determines the set of transitions to be executed next. Therefore, all queues and state tables have to be examined *for each module* of the specification. In general, the following tasks have to be done on each round:

1. Determination of a set of transitions matching two conditions: They must start from the current state, and their input event must be at the head of a queue of this module.

2. Further reduction of this set by the evaluation of `provided` clauses (if present).

3. Consideration of `priority` clauses.

4. Remaining transitions in the set indicate an indeterminism. In this case, one of them has to be selected at random.

This procedure can be quite time–consuming. For example, if several transitions with the same input event are possible in the actual state and — in the worst case — all transitions have the same priority, all `provided` clauses have to be evaluated. These might be complex boolean functions causing lengthy computations.

In addition, the evaluation has to be done for each active module of the specification. When using only one sequential scheduler the evaluation times add up for all modules. This can be improved by parallelism: Since the *determination* of fireable transitions of a module is completely independent of that for other modules, it can be done in parallel for all. Furthermore, no synchronization conditions have to be considered since the *determination* of fireable transitions is independent of their actual *execution*.

In this way the runtime behaviour can be improved by replacing the central scheduler with a parallelized version. Each module now has to determine its fireable transitions on its own and is thus able to run concurrently to other modules. It is no longer a passive data structure but has become an active unit. Only the actual firing of transitions requires synchronization with other modules which can be achieved by semaphores or message passing.

3. Mapping of Estelle Modules onto OSF/1 Tasks and Threads

Before we can describe how an Estelle specification can be implemented on a parallel computer, we must briefly introduce parallel hardware.

There is a wide range of parallel computer architectures for which different classifications have been proposed [13, 18, 32]. 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 parallel computer with optimal results for any given problem.

For our purposes it is clear that SIMD architectures (Single Instruction Multiple Data) are inapplicable. They are well suited to problems which perform the same operation simultaneously on all data objects, 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 is less obvious. 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 multithreading operating system such as OSF/1. It allows creation of 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[2]. For consistency, OSF/1 offers lock mechanisms to synchronize the access of shared data.

The main task in implementing an Estelle code generator for this architecture is the mapping of Estelle language elements onto the programming level of the operating system. In this paper we will not deal with all mapping decisions but will concentrate on the parallelization aspects.

First, one has to deal with the granularity of parallelization. To be independent from an actual distribution on tasks and threads, we define a **PUnit** as the unit of execution that can be processed in parallel with other PUnits. Earlier work suggests that the minimum unit for parallelization should be an Estelle module [30]. In addition, we have identified three reasonable **logical distributions** at the module level (cf. Figure 1):

1. **Parallelization of `system` modules.** A module with the attribute `systemprocess` or `systemactivity` builds a PUnit together with its successors. All subtrees rooted at system modules may then be executed in parallel. We gain asynchronous parallelism, as Estelle systems are asynchronous, and thus do not need any synchronization mechanism. Inside these subtrees, modules execute sequentially, i. e. some of the possible parallelism remains unused.

2. **Parallelization of `process` modules.** All modules whose parents have one of the attributes `process` or `systemprocess` run in parallel. The Estelle semantics of parent/child synchronization has to be obeyed, leading to a synchronization overhead.

[2] Actually, tasks are passive units while threads are active. I. e. not tasks but threads in different tasks communicate via message passing.

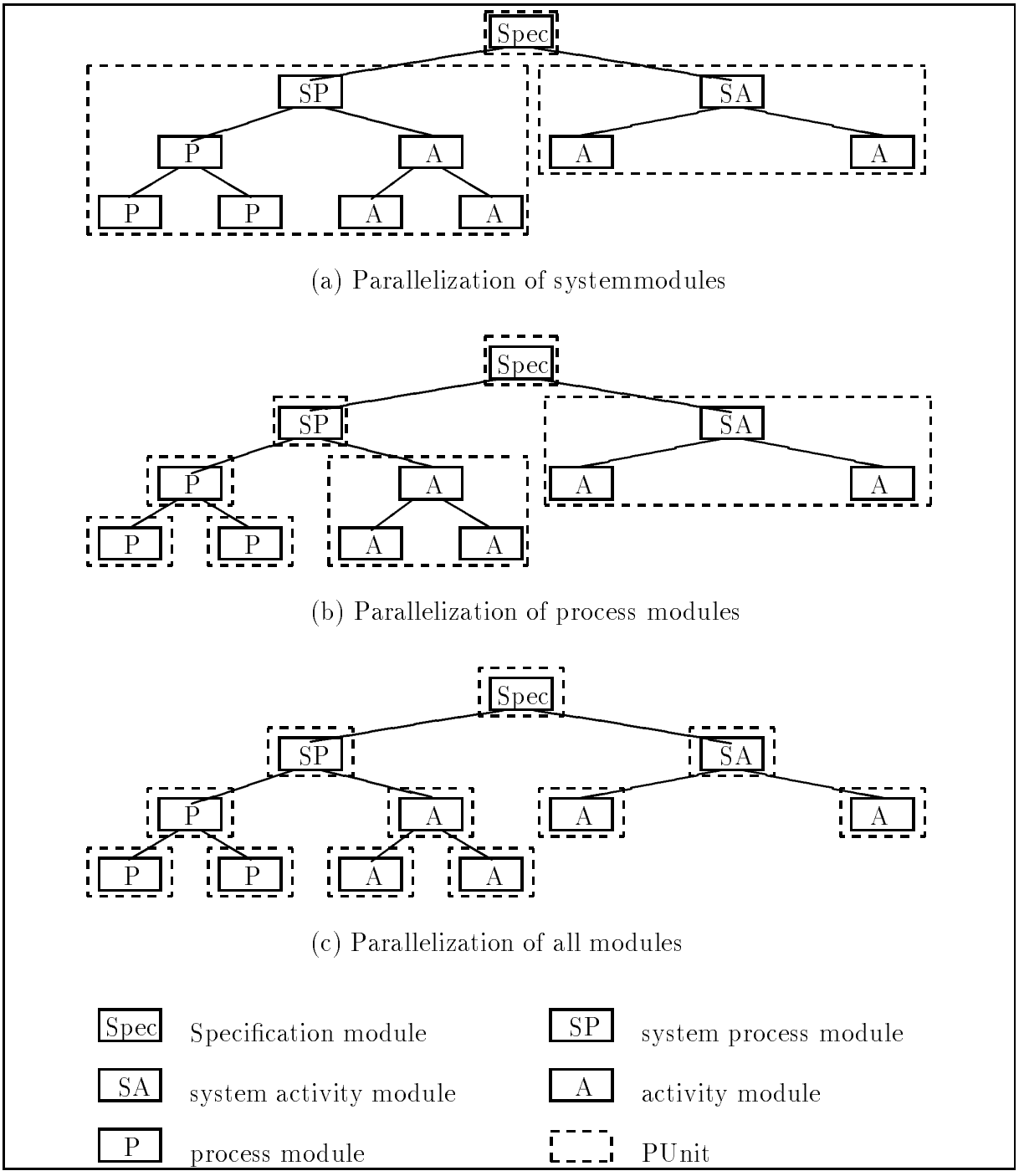

Figure 1. Logical distribution of Estelle modules

All module subtrees rooted at a module with `activity` or `systemactivity` attribute form a PUnit, i. e. all modules inside the subtree are sequentialized. This seems appropriate, as children of `activity` modules never execute simultaneously due to Estelle semantics. However, there is still some sort of parallelism which is not exploited. As shown in the previous section, transitions offered by children of `activity` modules may be selected in parallel by each child module without violating Estelle semantics. This feature cannot be implemented using this second architecture.

3. **Parallelization of all modules.** All modules have the ability to execute in parallel. Not only the parallelism but also the synchronization overhead is at its maximum because the semantics of Estelle now have to be implemented by explicit synchronization.

We decided to implement the third variant because it allows for maximum parallelism. It constitutes an extreme situation for the operating system. However, it would not be difficult to change our code generator to create code for the other variants.

Having achieved a distribution of Estelle modules to PUnits, we have to decide how to map these units onto the parallel programming features of OSF/1, i. e. tasks and threads. From the above, it is quite clear that each module runs in its own thread to allow for parallel execution. To be independent from the actual architecture, we leave it to OSF/1 to distribute the threads (the number of which depends on the specification) to processors (the number of which depends on the machine). But how many tasks should be used, and how should the threads be distributed among the tasks? Again, we identified three reasonable mappings (**physical distribution**):

1. **One task per module:** This mapping — which allows each thread to execute in its own task — has already been implemented several times [30, 35]. A task with one thread is nearly the same as a process in a "normal" UNIX system. It is quite useful for the validation of protocols because it allows for distributed testing. It is not suitable for the protocol implementation on multiprocessor systems because of the large message passing overhead. In addition, this solution causes many time–consuming context switches between tasks.

2. **One task for the whole specification:** In this variant, where all threads execute in the same task, there are no context switches nor is there any message passing between modules; all communication takes place via shared memory and semaphores.

3. **One task for each `system` module:** Subtrees of modules rooted at a `system` module communicate via message passing while modules inside the subtree communicate via shared memory.

The first variant is certainly not the one to choose for efficient protocol implementation. From the conceptional point of view, variant 3 is best because Estelle `system` modules are executing asynchronously. They have no shared data and may only communicate via their interaction points. This can be mapped exactly to OSF/1 tasks with message passing between them. Modules within a system's subtree often have to synchronize with

others (e.g. parent/child) and have shared data (Estelle's exported variables). These may be mapped to threads inside a task synchronized by and communicating via shared memory.

We decided to implement the third variant[3]. The second variant may easily be implemented by a simple change to the code generator.

4. Communication and Synchronization

Communication and synchronization of Estelle modules are closely related. Especially in a parallel environment, the mapping onto the operating system has to be designed carefully. We first take a look at communication problems without touching synchronization aspects. The latter will be covered in detail afterwards.

Estelle modules have three interfaces available for their communication with other modules. A description of our implementation for each interface follows:

1. **Initialization parameters** may be passed from a parent to a child module during creation time. By this feature, modules can be parameterized. The creation of a new module must be mapped to the creation of a new thread because each module runs in its own thread. To start a thread, a generic start routine has to be called which takes the name of a function as an argument. This name identifies the routine in which the thread starts its work. As a second argument, the generic start routine accepts a pointer to arbitrary data. This pointer will be passed to the special start routine and identifies the parameters of this routine. Thus, to implement initialization parameters, the code generator has to put them into a data structure and pass a pointer to this structure to the start routine.

2. **Exported variables** are variables of a child module accessible by its parent module. Inside a task, they are simply mapped to shared data and thus can be accessed by all threads in that task (the synchronization aspects are discussed later). Exported variables of modules in different tasks need not be implemented, their occurrence having been precluded by our mapping of modules onto tasks and threads (since variables can only be exported to parent modules). Both parent and child modules will always be in the same task as long as both are active, and therefore shared memory is the appropriate implementation.

3. **Interaction points** build an asynchronous message interface for modules. Their message queues are conceptually unlimited in size and can be simply implemented by a data structure representing a queue. A module wanting to send a message to another module could simply insert the message at the end of the interaction point's queue of the receiving module. The receiving module will eventually remove the message from the queue and process it.

   However, there has to be some kind of timing in accessing the queues. Consider the following situation: a parent module sends messages to each of its child modules. No predictions or assumptions about the time delay for passing the messages between

[3]The decision does not influence any load balancing issues. OSF/1 schedules threads and not tasks. The number of threads is equal in any of the three possible mappings.

the modules can be made. When child module 1 receives its message and, in reaction to it, sends a new message to child module 2, this message may arrive earlier at module 2 than the message from the parent module to this child. Thus, Estelle semantics would be violated. The solution for this problem is the use of temporary queues. All messages to be sent will first be put into these queues and will not be sent until all modules have finished, i. e. processed their part of the current Estelle cycle. This mechanism was already implemented in the PET/DINGO system and was adapted to our multithread environment.

Let us now come back to synchronization. The synchronization problem has to be solved for two different layers. The first layer includes all Estelle synchronization rules, essentially those for parent/child synchronization. The second layer includes synchronization of shared memory access, e. g. for interaction point queues.

To implement the Estelle synchronization rules in a concurrent environment, it is necessary to develop a **protocol** which defines the synchronization messages exchanged between threads (i. e. modules) and their possible orderings [30]. In general, there are three situations in which synchronization has to take place: *initialization* of a new module instance, *synchronization* during runtime and *termination* of an instance. We first present the general protocol in terms of messages and then describe the implementation of synchronization between threads of the same task. Intertask synchronization has already been described in earlier approaches [30, 35] and is therefore omitted.

When a parent module starts a new child module, Estelle rules require that the whole initialization of the child has to be completed before the parent module can continue its work. The initialization of the child may include further initializations of its own children. The Estelle `init` command therefore results in two messages: the parent module starts its child (which can be seen as an *init* message) and blocks until it receives an *end-of-init* message from its child.

During runtime of the Estelle specification the attributes of the modules determine the synchronization rules. System modules need not be synchronized as they are asynchronous. However, we have to distinguish `process` and `activity` modules. First, consider a `process` module that gets a *start-exec* message and thus has the right to execute a transition. At the beginning, it checks its own transitions. If it has one, this transition will be executed. Afterwards, it sends an *end-of-exec* message to the parent module and waits for the next synchronization message. If it has no fireable transition, it hands over the right to its own children. If it has at least one child, it passes a *start-exec* message to all of them and then waits for *end-of-exec* messages from all. Afterwards, it sends the *end-of-exec* message to its parent module.

`Activity` modules are treated in a slightly different way. If they are not able to execute a transition, they pass the right to exactly one of their children. If the child is not able to execute, it returns the right immediately. The parent module will then select another child module. This proceeds until one module has executed or all modules have been checked.

Implementing this procedure in exactly this manner would not make use of all possible parallelism. It is not against Estelle semantics to let all modules choose a transition and then select one of the modules with an enabled transition (= offering module) for

execution. On a sequential machine, that would result in longer execution times, as all modules will have to work, one after the other. On a parallel machine, however, the selection of transitions may be done fully in parallel (if there are enough processors), and afterwards, only one module has to execute its selected transition. The protocol used with `activity` modules thus works as follows: an `activity` that is unable to execute sends an *offer-request* message to all its children. Then, it waits for all *offer-response* messages (telling whether the corresponding child has a fireable transition or not). It now has to select at random one of the offering children and send a *start-exec* message to it. Again, the parent module blocks until it receives an *end-of-exec* message from this child.

Estelle modules are terminated by their parent module using the **terminate** or **release** statement. In the protocol, the parent module sends a *terminate-request* to the corresponding child module and blocks until it receives a *terminated* message from that child. The *terminated* message is the last message from the child before it ends.

In general, the implementation of the synchronization messages is done via shared memory. We use one (in case of start and termination of modules) or two (in case of runtime synchronization) shared integer variables for synchronization between a parent module and some child module threads. These shared variables are protected by a *lock variable*[4]. A *condition variable*[5] is used to implement the non–active waiting of threads for the change of values in the shared variables. To start a module, the *init* message is simply implemented by the call to the `pthread_start()` routine which allows one thread (in this case the parent thread) to start another one (the child). The parent thread sets the shared variable *response* to 1 and then waits until it is reset to zero by the child (this is the *end-of-init* message).

For synchronization during runtime, we use the two variables *command* and *response*. Via the *command* variable, the parent thread specifies the kind of message, e. g. *start-exec* or *offer-request* (for activities). Before it passes the right to execute to its child(ren), it assigns the number of executable children to the variable *response*. Then it wakes up the children using the condition variable and waits until the *response* variable is reset to zero. This will be the case when all modules have completed execution. When they are ready, they will decrement the value of *response* by 1 (after having locked the variable for their exclusive use). The corresponding *end-of-...* messages are thus implemented by this decrement operation.

To terminate a module, the parent thread specifies the value for *terminate* in the *command* variable. In this case, it does not check the response variable to wait for completion but waits for the end of the child's thread by calling the function `pthread_join()`. When the child gets the *terminate* command, it terminates all of its children and then calls `pthread_exit()`, which can be seen as a kind of suicide.

An example of the implementation is shown in Figure 2. We have a parent module M_0 of type `process` that has three child `processes` M_1, M_2 and M_3. Child module M_3 itself has two further children: M_4 and M_5 (Figure 2(a)). Figure 2(b) shows the parallelization of the corresponding threads t_i when M_0 issues a *start-exec* message.

The second layer of synchronization is shared data access. Shared data are mainly the

[4] *Lock variables* are provided by OSF/1 and can be used to implement mutual exclusion.

[5] *Condition variables*, also provided by OSF/1, allow for restarting threads depending on a change of variable values.

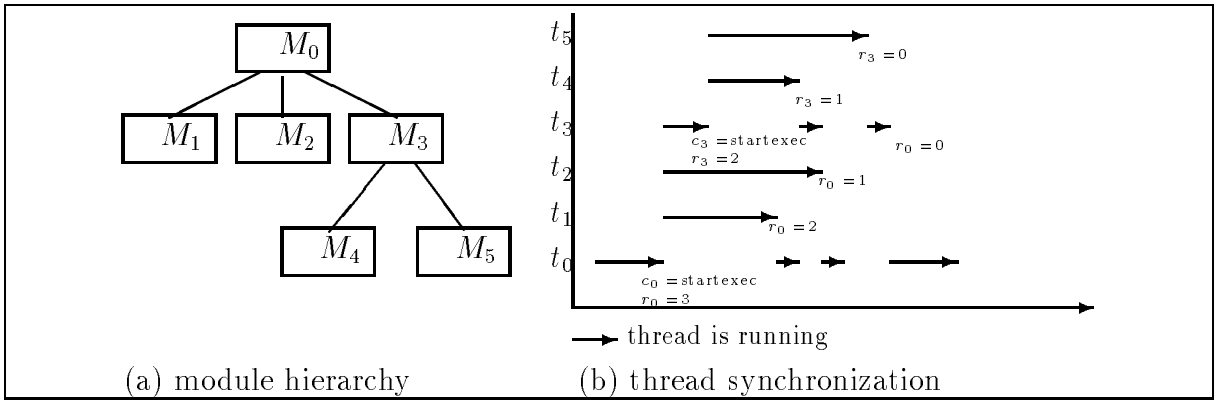

Figure 2. Thread synchronization example

implementation of exported variables and interaction point queues. Access to exported variables can be handled easily. It is always only the module itself and its parent module that have access to exported variables. As we saw above, parent and child threads will never run in parallel and therefore will not try to access the data simultaneously. The interaction point queues are a bit more difficult. We already implemented temporary queues to adhere to Estelle semantics. These queues, together with a 2–phase distribution algorithm for interactions (which was already used in the PET/DINGO Toolkit [35]), avoid the simultaneous access of an interaction point queue by two threads[6]. The most important result of our work on this layer was that an explicit synchronization by lock variables is unnecessary for shared data structures. The synchronization is already achieved by the implementation of Estelle semantics and by means of the existing interaction distribution algorithm.

5. Results

The code generator described in this paper was implemented on a parallel computer manufactured by Kendall Square Research, KSR1, running OSF/1 [27]. Our machine has 32 processors with a performance of 1.28 GFlops and 1 GByte virtual global memory. The processors have a wordlength of 64 bits, operate at a clock rate of 20 MHz and can execute two instructions per cycle. They are connected to a ring–shaped bus with 1 GB/s bandwith and have 32 MB local memory besides 256 KB cache memory each for data and instructions. All local memories together form the above–mentioned global memory of 1 GB, the consistency of which is guaranteed by an appropriate protocol on the ring.

As an example we used Estelle specifications of ISO presentation and session layer

[6]In the first phase, the algorithm collects all interactions from bottom-up. In the second phase, the collected interactions are redistributed over the modules until they reach their destination interaction point.

[20, 21, 23, 24] described in [11, 28, 39]. The session layer comprises the *basic combined subset*, whereas the functionality of the presentation layer is restricted to the functional unit *kernel*.

These specifications were enhanced by modules for the transport service and the simulation of presentation service users *initiator* and *responder* (cf. Figure 3).

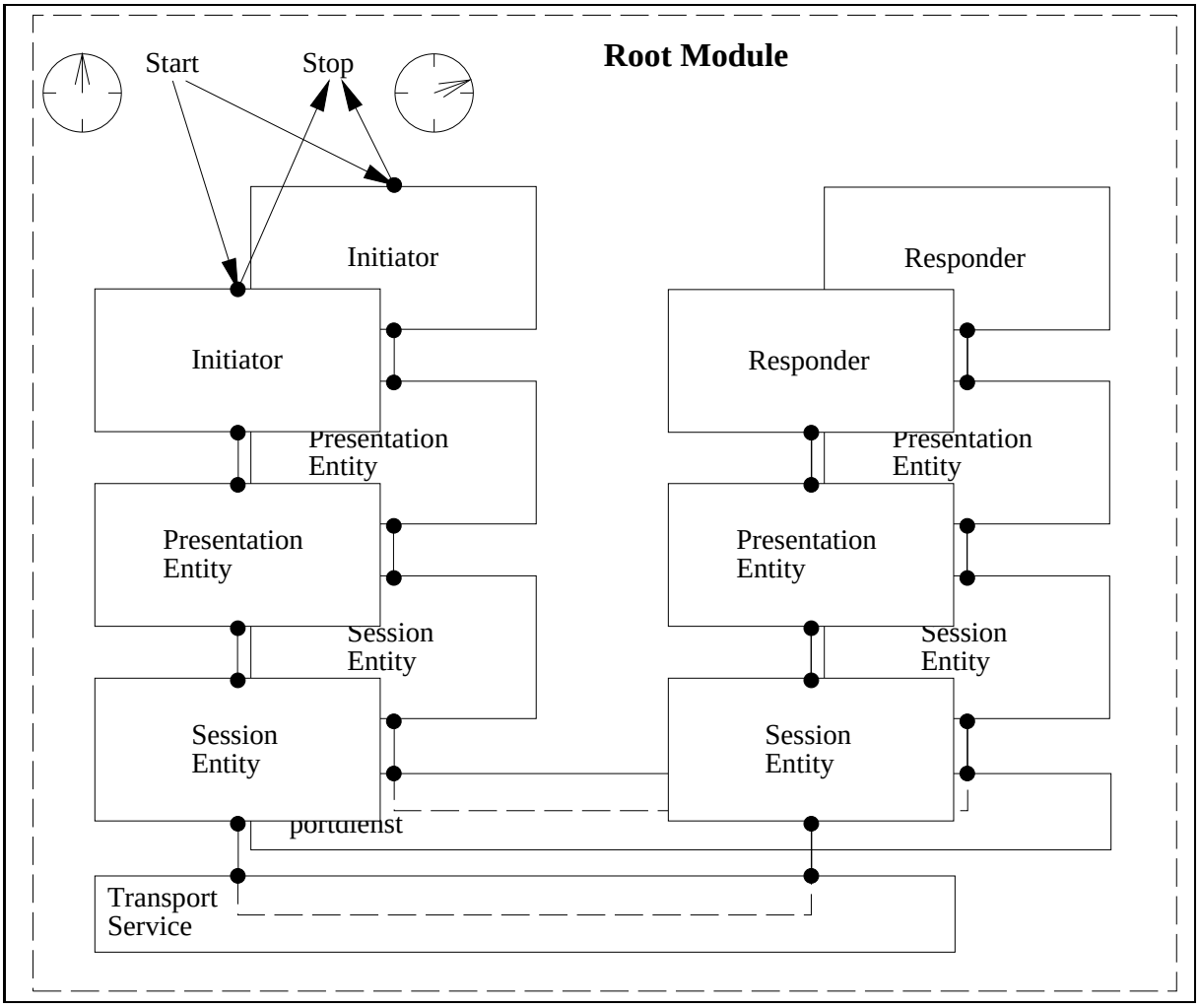

Figure 3. Specification used for the measurement

Several instances of these stacks can be created at runtime, thus allowing for experience with parallelism introduced by different connections. The root module triggers the measurement by sending a Start message to all *initiators* and starting a timer. Those in turn establish a connection to their peer *responders*, hand over a number of P–DATA.requests to their presentation entity (depending on a runtime parameter) and release the connec-

tion. After connection establishment, the data packets are sent in only one direction without waiting for any acknowledgements. Eventually the *initiators* send a Stop message to the root module which stops the timer after having received this message from all of them.

From this specification four variants were generated:

A: Sequential with one connection,

B: Sequential with two connections,

C: Parallel with one connection,

D: Parallel with two connections.

The sequential variants A and B were generated with the original PET/DINGO system, the parallel variants C and D with the modified version. In the parallel case each module entity is mapped to a thread and, since there is only one system module — the root module —, all threads run within the same task. Thus the version with one connection consists of eight threads, whereas the one with two connections comprises fifteen; of course, both sequential variants are represented by only one thread within one task.

All measurements took place on a closed set of sixteen processors with exclusive access only for the measured threads. Thus each thread was executed on its own processor. The average values of five series are depicted in columns A–D of Table 1.

Table 1
Runtimes and resulting speedup of the four variants

DATA-requests	sequential [s]		parallel [s]		speedup	
	1 conn.	2 conn.	1 conn.	2 conn.		
	A	B	C	D	A/C	B/D
10	1.9	3.2	1.7	2.7	1.12	1.19
50	4.7	8.6	3.4	5.1	1.38	1.69
75	6.7	11.6	4.7	6.2	1.43	1.87
100	8.2	14.8	5.4	8.4	1.52	1.76
250	18.5	35.7	12.6	16.9	1.47	2.11
500	35.3	66.5	22.8	32.2	1.55	2.07
750	53.5	101.8	33.7	47.0	1.59	2.17
1000	69.8	137.4	44.9	60.0	1.55	2.29

Two conclusions can be drawn from these results:

- A comparison of variants A and C yields the speedup based on a **layer–oriented parallelism**. Since each layer entity is specified by exactly one module and our code generator maps each module to exactly one thread, a pipelining effect is realized by variant C, with each layer being processed by its own processor. The resulting speedup is shown in column A/C.

- The speedup introduced by the combination of this parallelism and the one enabled by the **concurrent execution of two connections** can be evaluated with variants B and D and is shown in the last column of Table 1.

Both speedups are shown in Figure 4. As can be seen, from 100 DATA.requests on the values stabilize at about 1.5 for the layer–oriented speedup and at ca. 2.2 in the combined case.

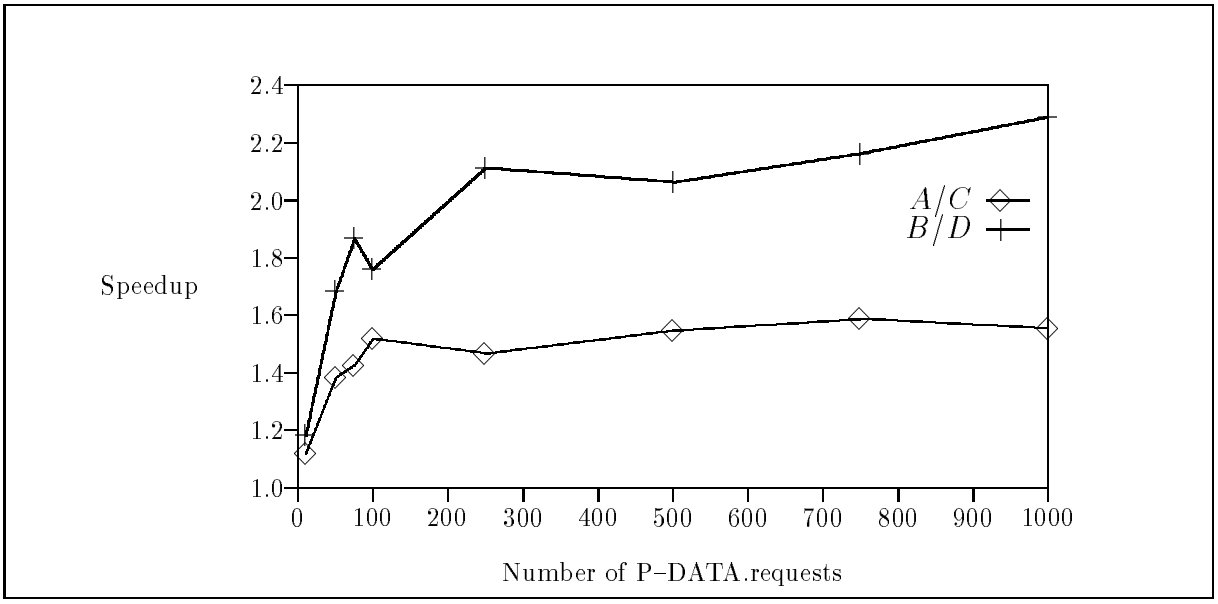

Figure 4. Speedup

At first sight the speedup is disappointing. However, the values shown here cannot be generalized. They are specific for the underlying protocols (presentation and session layer), which were not designed particularly for parallel execution. Protocols especially developed with parallelism in mind can be expected to yield better values.

6. Conclusion and Outlook

We have described how code generators can be used not only for simulation but also for actual implementation of communication protocols. Efficiency at runtime can be achieved by the use of parallelism. For this purpose, the runtime system of an existing Estelle code generator, PET/DINGO, was modified to run under the parallel operating system OSF/1. We emphasize that parallelization of the runtime system is as important as parallelization of the FSM code.

A main point was to separate the *determination* of fireable transitions — which can be done completely in parallel — from their actual *execution*. In addition, all Estelle modules

are processed concurrently, while a synchronization protocol guarantees the adherence to the semantic rules of Estelle.

The results obtained by the generated software show a considerable speedup of protocol execution. However, they are not too near to the theoretical upper limit. Currently, we are working to improve the obtained speedup. We already identified implementation issues that stand against the optimal performance:

- The **synchronization mechanism** of our implementation with semaphores is quite complex. KSR-OS offers another mechanism, called *barrier synchronization*. Its model of barrier master and slave is very similar to Estelle's father and child synchronization.
- Often, the **synchronization times** exceed the time spent on protocol execution. In this case, a parallelization is not advisable and should be left out.
- The KSR architecture leads to an increase in synchronization and communication time when more processors than belong to a ring are used. In that case, Estelle modules belonging to one system module tree should run on processors in the same ring.

Based on these experiences, we are developing a new strategy for the logical distribution of modules. It takes into account the runtime of each module, thus leading to a balanced distribution of modules to processors.

7. Acknowledgements

We would like to thank Prof. Dr. W. Effelsberg and Prof. Dr. R. Gotzhein for their valuable contributions to this paper as well as B. Weyerer for improving the readability of the paper.

REFERENCES

1. M. Bilgic and B. Sarikaya. An ASN.1 encoder/decoder and its performance. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verifikation, X*, pages 141–154. Elsevier Science Publishers B.V. (North–Holland), 1990.
2. T. P. Blumer and R. L. Tenney. A formal specification and implementation method for protocols. *Computer Networks*, 6:201–217, 1982.
3. G. v. Bochmann, W. Gerber, and J.-M. Serre. Semiautomatic implementation of communication protocols. *IEEE Transactions on Software Engineering*, SE–13(9):989–1000, September 1987.
4. T. Braun and M. Zitterbart. Parallel transport system design. In Danthine and Spaniol [8].
5. CCITT SG X: Recommendation Z.100: Specification and description language SDL. Contribution Com X–R15–E, 1987.
6. 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.

7. D. Comer. *Internetworking with TCP/IP*. Prentice–Hall, Englewood Cliffs, 1988.
8. A. Danthine and O. Spaniol, editors. *4th IFIP conference on high performance networking*, Liège, 1992.
9. M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azema, and V. Chari, editors. *The Formal Description Technique Estelle*. Elsevier Science Publishers B.V. (North–Holland), Amsterdam, 1989.
10. 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, February 1989.
11. 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.
12. S. Fischer. Generierung paralleler Systeme aus Estelle–Spezifikationen. Master's thesis, Lehrstuhl für Praktische Informatik IV, Universität Mannheim, 1992 (in German).
13. M. J. Flynn. Very high–speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.
14. R. Födisch, T. Held, and H. König. A protocol development environment based on Estelle. In *Proceedings of Information Networks and Data Communication*, Espoo, Finland, 1992.
15. D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. C. Williamson. High–speed parallel protocol implementation. In H. Rudin and R. Williamson, editors, *Protocols for High–Speed Networks*, pages 165–180, Zürich, 1989. IFIP WG 6.1/WG 6.4, Elsevier Science Publishers B.V. (North–Holland).
16. B. Heinrichs. XTP specification and parallel implementation. In *International Workshop on Advanced Communications and Applications for High Speed Networks*, pages 77–84, München, 1992.
17. 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.
18. K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw–Hill, New York, 1984.
19. Information processing systems — Open Systems Interconnection — Basic Reference Model. International Standard ISO 7498, 1984.
20. Information processing systems — Open Systems Interconnection — basic connection oriented session service definition. International Standard ISO 8326, 1987.
21. Information processing systems — Open Systems Interconnection — basic connection oriented session protocol specification. International Standard ISO 8327, 1987.
22. Information processing systems — Open Systems Interconnection — LOTOS: Language for the temporal ordering specification of observational behaviour. International Standard ISO 8807, 1987.
23. Information processing systems — Open Systems Interconnection — connection oriented presentation service definition. International Standard ISO 8822, 1988.
24. Information processing systems — Open Systems Interconnection — connection oriented presentation protocol specification. International Standard ISO 8823, 1988.

25. Information processing systems — Open Systems Interconnection — Estelle: A formal description technique based on an extended state transition model. International Standard ISO 9074, 1989.
26. C. Jard and J. M. Jezequel. A multi–processor Estelle to C–compiler to prototype distributed algorithms on parallel machines. In Ed Brinksma, Giuseppe Scollo, and Chris A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 161–174. IFIP WG 6.1, Elsevier Science Publishers B.V. (North–Holland), 1989.
27. KSR manual set. Kendall Square Research Corp., 1991.
28. P. Mondain-Monval. Estelle description of the ISO session protocoll. In Diaz et al. [9], pages 229–269.
29. A guide to OSF/1: A technical synopsis. O'Reilly & Associates, Inc., 1991.
30. D. Peter. Entwurf, Realisierung und Integration eines Protokolls zur verteilten Ausführung von Estelle–Spezifikationen. Master's thesis, Universität Hamburg, Februar 1991 (in German).
31. T.F. La Porta and M. Schwartz. A high–speed protocol parallel implementation: Design and analysis. In Danthine and Spaniol [8].
32. M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw–Hill, 1988.
33. J.-L. Richard and T. Claes. A generator of C–code for Estelle. In Diaz et al. [9], pages 397–420.
34. D. P. Sidhu and T P. Blumer. Semi–automatic implementation of OSI protocols. *Computer Networks and ISDN Systems*, 18:221–238, 1990.
35. R. Sijelmassi and B. Strausser. The PET and DINGO tools for deriving distributed implementations from Estelle. *Computer Networks and ISDN Systems*, 25(7):841–851, 1993.
36. L. Svobodova. Measured performance of transport service in LANs. *Computer Networks and ISDN Systems*, 18(1):31–45, 1989.
37. A. S. Tanenbaum. *Modern Operating Systems.* Prentice–Hall, Englewood Cliffs, 1992.
38. S. T. Vuong, A. C. Lau, and R. I. Chan. Semiautomatic implementation of protocols using an Estelle–C compiler. *IEEE Transactions on Software Engineering*, 14(3):384–393, March 1988.
39. S. Weber. Spezifikation und Implementation eines Datenkommunikationssystems mit Estelle. Master's thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, April 1991.