Improving the Efficiency of Automated Protocol Implementation Using Estelle

R. Gotzhein[†], J. Bredereke[†], W. Effelsberg[*], S. Fischer[*], T. Held[‡], H. König[‡]

[†]: Dept. of Computer Science, University of Kaiserslautern, P.O. Box 3049, D-67653 Kaiserslautern, Germany

[*]: Praktische Informatik IV, University of Mannheim, D-68131 Mannheim, Germany

[‡]: Dept. of Computer Science, TU Cottbus, P.O. Box 10 13 44, D-03013 Cottbus, Germany

September 6, 1995

Abstract

Correctness and runtime efficiency are essential properties of software in general and of high-speed protocols in particular. Establishing correctness requires the use of FDTs during protocol design, and to prove the protocol code correct with respect to its formal specification. Another approach to boost confidence in the correctness of the implementation is to generate protocol code automatically from the specification. However, the runtime efficiency of this code is often insufficient. This has turned out to be a major obstacle to the use of FDTs in practice.

One of the FDTs currently applied to communication protocols is Estelle. We show how runtime efficiency can be significantly improved by several measures carried out during the design, implementation and runtime of a protocol. Recent results of improvements in the efficiency of Estelle-based protocol implementations are extended and interpreted.

1 Introduction

Formal Description Techniques (FDTs) have been successfully used to specify communication protocols precisely and concisely. Suitable tools now available for some FDTs generate large portions of a protocol implementation directly from its formal specification. However, FDT-based protocol implementations have often been criticized for their inefficiency at runtime in comparison with hand-coded implementations.

While runtime efficiency is a property desirable for protocol implementations in general, it becomes crucial in the context of modern high-speed communication systems. Here, the bottleneck is no longer the communication link, but the processing of incoming and outgoing messages in the nodes. This has led to the development of a new class of protocols called high-speed protocols [DDK+90] designed in particular to reduce processing time and thus to achieve higher throughput.

It is still an open question whether FDT-based implementations of high-speed protocols can compete in performance. Before a definite answer can be given, thorough experiments in order to improve the efficiency of FDT-based implementations have to be conducted. Though only very few results are available in this respect today, they clearly indicate that significant gains in runtime performance are possible by a systematic approach to performance optimization, taking advantage of all available information about the specified protocol, the FDT and its semantics, and the machine environment.

Let us first take a look at earlier, not FDT-based experience with performance optimization. It is interesting to learn from the results there. Protocol software tends to be very large and complex, and many efforts to optimize runtime performance have been reported in the literature.

One possibility is always to hack the code by loop unrolling, minimizing copy instructions etc, but that is of little interest to us here.

A first systematic approach is to design new protocols that are better suited for high–performance execution. An important example is XTP, the Xpress Transfer Protocol [SDW92]. Unlike the OSI or the TCP/IP protocols, XTP supports *function selection*: most of the protocol functions such as error correction, flow control, acknowledgments, and resequencing can be turned on or off on a per-connection basis. The same protocol module can thus be used for different application types, including connectionless (datagram) transfer, connection-oriented mass data transfer, and request-response transfers typical for client-server interaction in LANs (sometimes called "transactions").

But feature selection has disadvantages. First, the code for the module itself remains very big even if only a few protocol functions are active. This can lead to additional paging at runtime. Second, the selection leads to many IF– or CASE–statements in the data path, slowing it down.

Therefore, a second approach was developed: *configurable protocol stacks*. The idea is to link only those protocol modules together at connection setup time that are actually needed for the application [ZST93, PPVW92]. The advantages are much smaller code modules and reduced overhead for IF– or CASE–statements. But a major disadvantage is the added complexity of the configuration process, in particular by providing clean module interfaces for each protocol function so that it can be configured with every other function. Also, this approach creates increased connection–setup times and incompatibility with non–configured protocols (i.e. both sides must construct the same configuration in order to interoperate).

Neither function selection nor configurable stacks are directly related to parallel execution. Parallelisation of protocol modules "by hand" has also been tried in the past [Zit89, BrZi92, BjGu93]. Experience from these projects is often very disappointing: the speedup gained by parallel execution on multiprocessor hardware remains below expectation. The main reason is that *it is very difficult to balance additional overhead for synchronization and communication with the concurrency gain.* For example, if a specific protocol function requires only one hundred machine instructions, it does not make sense to off–load it to another processor. And many functions in the data phase of a transport protocol are on that order of magnitude [ClTe90]. A well–balanced distribution of functions on parallel processors is very difficult to do by hand.

In this paper, we concentrate on FDT-based performance optimizations. More specifically, we collect, extend and interpret a number of recent results related to efficiency improvements of Estelle-based protocol implementations. Estelle [DeBu89] is an FDT especially designed for the formal description of communication protocols, it is internationally standardized since 1989 [ISO89]. In Section 2, we examine the extent to which the specification style influences the efficiency of such implementations and how it may improve them. Section 3 shows how the code generated by an Estelle-compiler and the runtime system influence efficiency. Section 4 discusses experiments with the configuring of protocol modules at design time and at runtime. All experiments are quantified by measurement results.

2 The influence of specification styles

Estelle specifications are usually (and should be) written in a problem-oriented style. Thus, the specification is as close as possible to the conceptual model, ensuring a high level of abstraction and good readability. The conceptual model, however, is generally not a suitable basis for automated generation of runtime efficient code, despite the fact that some optimizations can still be made by the compiler (see Section 3), even during execution in a parallel environment using load balancing mechanisms (see Section 4). The reason is that a number of factors influencing runtime efficiency are not known or at least should not be considered when the problem-oriented Estelle specification is written, among them:

- the specification structure (number of module instances, depth of the module hierarchy, dynamic modifications of the structure),
- the internal module structure (number of major states, number of transitions, length of transition blocks),

- synchronization among module instances (synchronization imposed by the Estelle semantics, synchronization due to the specified problem),
- the target architecture (degree of possible parallelism, communication costs between processes),
- compiler and runtime system (policy of determining fireable transitions, costs for exchanging messages between module instances).

Given a problem-oriented Estelle specification, and determination of the factors listed, a number of improvements become feasible by transforming the original Estelle specification in a stepwise manner towards an Estelle specification with improved runtime efficiency. In this section, two main methods of enhancing runtime efficiency by means of suitable specification styles are elaborated. Firstly, *structure* available in the original specification is transformed and/or reduced to lower the overhead in the implementation. Secondly, the expressive power of the Estelle language is enhanced in order to further reduce the difference between the conceptual model and the Estelle specification. We outline a means to specify the full conceptual problem-inherent *concurrency* which allows for more efficient implementations on parallel as well as non-parallel architectures[1]. Both approaches can be applied together. A number of experiments with measurements on different hardware platforms demonstrate that the transformations significantly improve runtime efficiency.

2.1 Transforming specifications for more efficiency

The use of different specification styles which are oriented more towards the problem or more towards an implementation, and of transformations among them, has been investigated for several FDTs, e.g. for LOTOS [VSvS88]. For Estelle, too, there are different specification styles, and the runtime efficiency may be increased by a transformation from one style into another which makes less use of "expensive" language features. In current automated implementations, the synchronization of module instances and their communication by message passing is relatively time-consuming, as is the frequently repeated determination of the next transition to fire (compare Section 3). The implied overhead may be reduced by transformation into a specification which has a different and/or reduced structure, for instance prescribes fewer but larger transitions, fewer module instances, and less message passing and synchronization.

A common example of a situation with many module instances is a layered protocol architecture, where each layer is expressed by at least one Estelle module

[1] Other work to increase the expressive power [Ten90, ChAm91] should have potential for efficiency improvements, too, but this is not elaborated here.

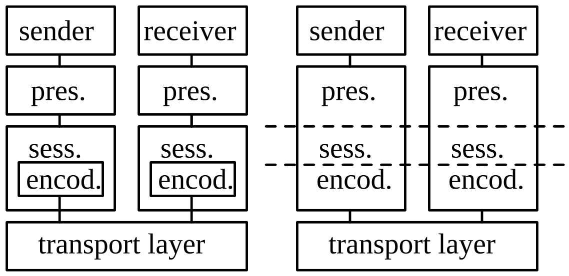


Figure 1: The system architectures.

instance. Hofmann [Hof93] proposes to integrate two (or more) such protocol automata by constructing a *product automaton*, thus eliminating the communication interface between two adjacent automata. Each transition which outputs a message to a common interaction point is merged with the corresponding receiving transition. A single new automaton is formed which exhibits the same behaviour as the previous pair of automata. This cuts the costs both for passing a message to the second automaton and for determining which transition can process it. Hofmann integrates specifications of the ISO presentation and session layers (Figure 1), achieving a speedup of up to a factor of three on a DECstation for the pass-through of a message in the data phase. A disadvantage is the lessened legibility of the integrated specification.

This is mostly avoided by the approach in [Bre95] which employs a *structured product automaton*. Two communicating module instances are still transformed into a single one. But the functionality contained in each remains syntactically separated from that of the other, and the communication interface remains visible. Furthermore, the transformation is reversible, so the original automata can be recovered. This is achieved by transforming each transition definition into one procedure definition, and by transforming each communication *between* the two automata into a procedure call. There remain some aspects which have to be taken care of, one of them being that a minor (straightforward and compatible) language extension is a prerequisite for this specification style. Estelle requires that sending a message to another module instance (*output* statement) be done literally in a transition definition and not be performed indirectly in a procedure. This restriction has to be dropped. The syntax and semantics of this extension have been defined formally [Bre95], and it has been implemented into the Pet/Dingo toolset introduced in [SiSt93].

Case studies with different layered protocol automata have investigated both the effects on the runtime efficiency and on the legibility of the specification. It turns out that the transformation of the examples is straightforward, and that the problem-specific structure is still clearly visible. So, the legibility can be preserved. Runtime measurements show that Hofmann's product automaton and the structured product automaton gain the same speedup on a SUN SPARCstation

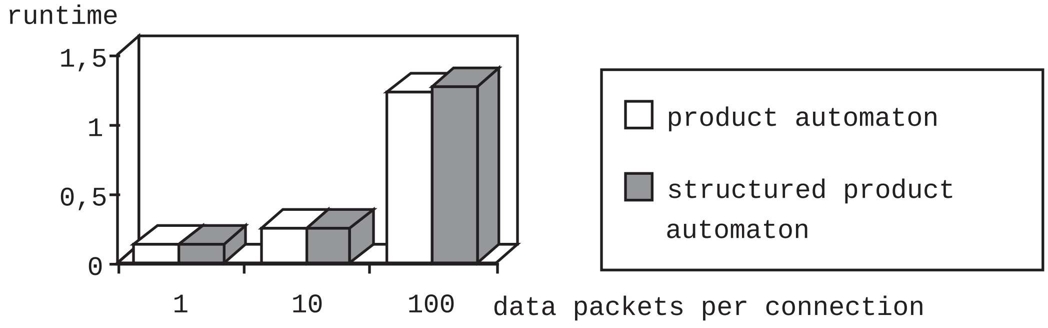


Figure 2: No additional costs by structuring the product automaton.

(Figure 2). The performance of an entire system, where three layered automata have been integrated (Figure 1), rises by about the factor of four on the SUN SPARCstation.

2.2 Efficiency enhancements through specified concurrency

An Estelle specification describes a hierarchically structured system of module instances communicating asynchronously by exchanging messages. In principle, the overall system behaviour may be perceived as the concurrent execution of module instances, where required synchronization between system components can be achieved explicitly by message passing. However, in Estelle, concurrency is further reduced by fixed priority regulations between module instances (called parent-child priority), and by additional synchronization constraints that can be specified by attributing modules[2]. An example is shown in Figure 3, where the specification module A is attributed by *systemprocess*, and A3 is attributed by *process*[3]. This means that all modules are synchronized, i.e. they proceed in rounds such that in each round, at most one transition can fire in each module. Still, transitions may fire concurrently, e.g. modules A1, A2, and A3 may proceed in parallel in an implementation if A has no fireable transition. Thus, it is still possible to specify a certain degree of concurrency in Estelle, which can later be exploited in the automatic generation of code for parallel execution. Several Estelle compilers already support parallel execution (see Table 1), which is a prerequisite for a more efficient implementation of Estelle specifications in a multiprocessor environment.

[2]Reasons for this synchronization are the simpler treatment of dynamic changes of module and communication structure, and the use of external variables visible to the parent module.

[3]Only the relevant attributes are shown.

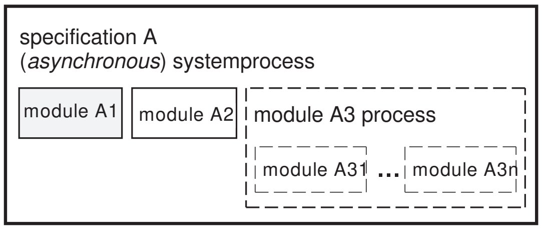


Figure 3: An Estelle module structure

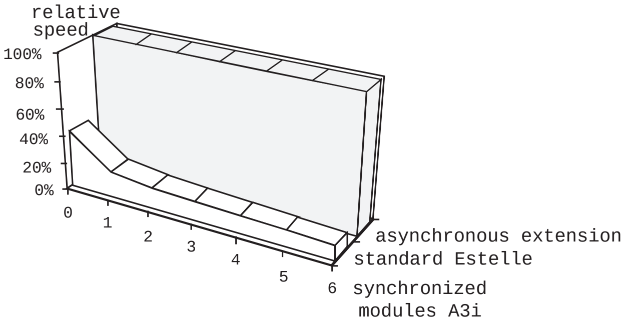


Figure 4: Experimental results

Practice shows that the degree of concurrency that can be specified in Estelle often fails to match the problem-inherent concurrency [BrGo94], since Estelle enforces some synchronization between module instances that may not be required by the conceptual model. As a consequence, the potential for efficiency enhancements is reduced due to restrictions of the language Estelle. For example, modules of different levels of hierarchy belonging to the same part of the module tree can never fire transitions concurrently. In Figure 3, this applies to modules {A,A1}, {A,A2}, {A,A3,A3i}. Concurrency lost in an early development stage cannot simply be reintroduced later. In this particular case, the reintroduction of concurrency by the Estelle compiler would violate the Estelle semantics. This limits the usefulness of Estelle as a specification language for parallel systems.

By removing the parent-child priority, and by reducing synchronization constraints between module instances, significantly more concurrency can be specified. In [BrGo94], a proposal for a compatible language extension has been made: the keyword *asynchronous* is used as an additional attribute of process modules in order to specify that the mentioned synchronization constraints no longer apply to the child modules of the next module hierarchy level. In Figure 3, *asynchronous* is attributed to A, therefore allowing concurrent execution of A, A1, A2, and A3. Synchronization between A3 and A3i is not changed and still applies. This allows to "switch off" synchronization where required, with the extreme case that the overall system behaviour equals the concurrent execution of its components. The semantical implications are non-trivial and are studied in [BrGo94], resulting in a proposal for a modified Estelle semantics. As pointed out earlier, as much concurrency as possible should be expressed in the problem-oriented specification, since it cannot be reintroduced later without violating the Estelle semantics.

In [Fis95], the described extension has been realized in an experimental implementation based on the Pet/Dingo toolset [SiSt93]. Several experiments have been performed, with the result that in cases where the traditional Estelle-enforced synchronization is not part of the conceptual model, substantial runtime improvements have been observed on a parallel machine. Figure 4 shows some results obtained on a KSR [FBR93] based on the Estelle specification outlined in Figure 3. In that experiment, A and A3 are inactive, A1 and A2 communicate, and A31 through A3n have spontaneous transitions only, thus producing some load. In all cases, each module has its own processor, i.e. up to 10 processors are involved. The number of synchronized modules A3i varies between 0 and 6. Figure 4 shows the relative speed of module A1, with the maximum speed of all experiments as the basis. This maximum execution speed is reached only with the "asynchronous" extension. In standard Estelle, only 40% of that execution speed or less is possible for A1, which is due to the synchronization constraints. Indeed, to implement these constraints, an expensive protocol is used in the Pet/Dingo toolset. Adding more synchronized modules reduces the execution speed of A1 in the case of standard Estelle, while no change is observed for the asynchronous extension. This shows that the language-inherent synchronization between modules results in substantial overhead at runtime and should be dropped in cases where it is not part of the specified protocol.

3 Design issues of compiler and runtime system

Tools that generate an implementation automatically can be aimed at different goals: a simulation, a prototype implementation, or an implementation for real-life use. Accordingly, the efforts to achieve runtime efficiency vary. In current implementations, we have identified three pieces of code consuming most of the execution time: synchronization between module instances for the selection of fireable transitions, selection of fireable transitions within a module instance, and implementation of the Estelle primitives for the communication between modules. In a study of the LLC-3 protocol [HeKö95], 37% of the total runtime in an already optimized setting was related to the first and second code category, and another 7% to the third (see Figure 5). In Section 3.1, we will address these three pieces in more depth, and we will evaluate the status quo in Section 3.2.

3.1 Approaches for efficiency improvements

Synchronization between module instances for the selection of fireable transitions. In Section 2.2, we already discussed the synchronization of Estelle module instances due to the different degrees of specified concurrency. The formal semantics of Estelle [ISO89] describe this synchronization by rounds of transition selection and firing. In a straightforward implementation, this typically results in two recursive passes over the module instance hierarchy per round, and a buffering of fireable transitions. This approach (referred to as "approach I") is used, e.g., in the early NBS prototype compiler [NBS87]. For the LLC-3 protocol, 93% of the total runtime is spent on synchronization and selection of transitions, i.e., on the execution of the first and second code categories [HeKö95]. It is more efficient to make only one recursive pass through the module instance hierarchy, and to execute fireable transitions upon completion of the search for one subtree. This can be done so that it is in accordance with the semantics of Estelle. All current compilers (e.g., [SiSt93, KrGo93, HeKö95]) use this approach (referred to as "approach II"). In the setting of [HeKö95], it reduces the runtime share for synchronization and selection of transitions to 55%[4].

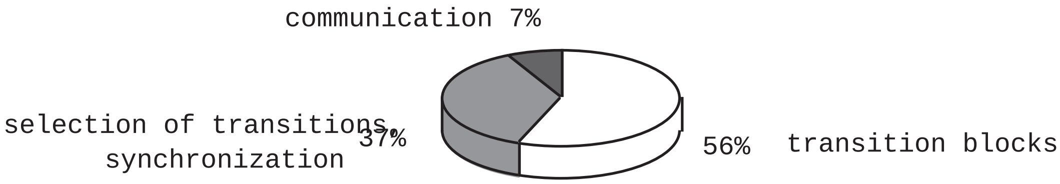


Figure 5: Relative shares of runtime system code categories on the execution time in a study of the LLC-3 protocol.

Selection of fireable transitions within a module instance. The selection depends among other things on the major state (*from clause*), on the messages received (*when clause*), and on boolean expressions over the extended state space (*provided clause*). One approach to determine the fireable transitions (called *program driven*) is to let the compiler generate appropriate *if* and/or *case* statements. Since the domain of the major state is usually small, one can also use a table which allows a direct (indexed) access to the set of transitions with an appropriate *from* clause (*table driven* approach). This may be extended to the *when* clause because the domains of the message channels are usually relatively small. This is not the case for the many boolean expressions of the *provided* clauses. Consequently a combination of a table driven and a program driven approach is necessary, using a two-dimensional table for the *from* and *when* clauses and a program driven selection for the *provided* clauses afterwards. Since the runtime of the table driven approach is independent of the size of the table, it outperforms the purely program driven approach for large specifications [Svo89]. There is evidence [HeKö95] that the break-even point can be as low as 4 decisions (see Figure 6).

[4] A major portion of the improvement is due to an optimized treatment of the *delay* clause (see below).

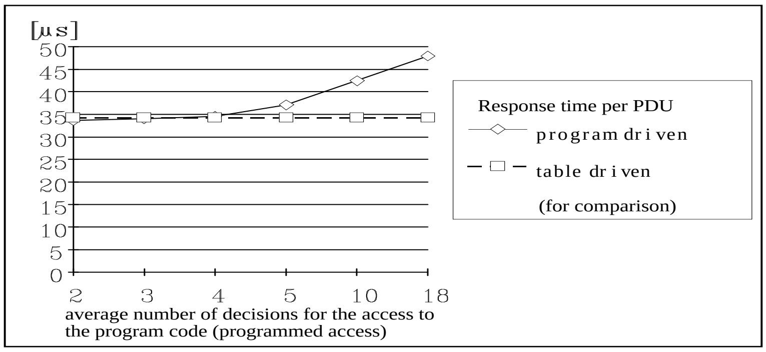


Figure 6: Comparison of the program driven and the table driven approach.

A specification may contain indeterminism in the selection of transitions. An implementation may "throw dice" here. But such fair choice is not required by the semantics of Estelle. Therefore, we are free to always fire the first transition found in a module, thereby immediately ending the determination phase for fireable transitions. This is advantageous especially for the program driven selection approach. If only one transition is fireable at a time, on the average only half of the transitions have to be searched instead of all of them. If several transitions are fireable simultaneously, the factor is even greater than two. For the table driven approach, the effect is smaller and depends on the number of *provided* clauses used.

Optimizing the implementation of the *delay* clause may further improve efficiency. Whether or not the specification uses the *delay* clause, approach I permanently performs the time control needed to realize it, resulting in major overhead. This largely explains the disappointing outcome of the measurements reported above. Approach II enables time control only when the *delay* clause is actually used in a specification. Figure 7 demonstrates the influence of the *delay* clause on execution times in both approaches.

Communication between modules. This is not an important topic in sequential implementations since it can be realized as fast as procedure parameter passing. In distributed implementations, however, it accounts for a large share of the total run time. Current tools (e.g., [SiSt93, KrGo93]) allow remote communication only through an entire protocol stack over a relatively slow network, with high costs. But there are experiments allowing not only the distribution of subsystems with acceptable speed but also the parallel execution of concurrent protocol instances. This requires tightly coupled hardware processors. Using shared memory, read and write access to the interaction point queues have to be synchronized by some kind of semaphore or monitor mechanism. In [FiHo94], monitors have been implemented using OSF/1's lock and condition variables. Inter-module communication was reduced to obtaining access to queues followed by inserting pointers, which proved to be very fast. Further experiments [FiEf95] show that using barrier synchronization, a mechanism specific to OSF/1 on KSR machines, instead of lock and condition variables results, under certain conditions, in further gains (in these experiments, up to 25%).

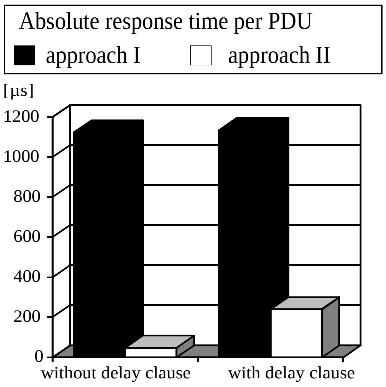


Figure 7: Impact of *delay* clauses on the absolute processing time per PDU in selection approaches I and II.

In addition, there have been some experiments [HeKö95] on the optimal implementation of Estelle message queues. Some overhead can be reduced by merging all queues of a module instance into one central (non-FIFO) queue and picking out messages as required by the semantics of Estelle. This reduces the absolute response time per PDU in the case of the LLC-3 protocol by about 8.3%. The share of the inter-module communication routines of the total response time decreases from about 9.5% to about 7.2%.

3.2 Evaluation

In Table 1, we have compiled available information for current Estelle compilers with respect to the discussed optimizations. The compiler from NIST [NBS87], as the earliest of these, generating only sequential implementations and performing no optimizations, employs the table driven transition selection approach quite efficiently. It has been extended to the UHH compiler [KrGo93] which additionally allows parallel and distributed implementations. The latter are also achieved by the Pet/Dingo toolset [SiSt93]. Designed to prove the feasibility of distributed implementation rather than for efficiency, it accounts for a not very efficient program driven transition selection scheme. Another performance drawback stems from the target language C++ rather than plain C. Furthermore, communication is implemented by string streams enhancing portability at the cost of expensive encoding and decoding. Pet/Dingo has been ported to parallel hardware [FiHo94, PHK95] and has been supplied with a modified runtime system in order to allow efficient parallel execution of module instances on tightly coupled parallel hardware. The commercial EDT toolset [Bud92] is the only one to provide a complete development environment. Apart from generating prototypes, it also allows implementations for real-life use. [CLV93] provides a more comprehensive list of Estelle tools.

Table 1: How much do current compilers employ the discussed optimizations?

compiler	NIST	Pet/Dingo	EDT	UHH
distributed implementation	no	yes	yes[a]	yes
parallel implementation	no	(yes)[b]	no	yes
target language	C	C++	C	C
synchronization passes	2	1	2/1?	1
transition selection scheme	table	program	table	table
quick but unfair choice	no?	prepared[c]	no?	yes
comm. link shortcut	—	partially[d]	—	partially[d]
compile-time optimization	no	no	no	no
runtime optimization	no	no	no	no

[a] see [LFV95]
[b] in modified versions [FiHo94, PHK95]
[c] prepared for different choice probabilities, including unfair choice
[d] only on the subsystem level

If we design a code generator for a specific hardware and software environment, more information is available than on the specification level. This is only exploited by the Pet/Dingo variants to a limited extent. Some information will even only be known at compile time. The *variable implementation model* approach in [HeKö95] allows a manual compile time decision for some of the issues discussed above. The configuration approach described in Section 4 is an attempt to automate certain implementation decisions.

To summarize, not even the statically applicable strategies for more efficiency are generally applied in current Estelle-tools. Also, there is clearly a need for optimizing compilers [Svo89] if the generation of efficient runtime code is an objective. They need to add and to exploit the implementation-specific information which should not be included in a specification. This information can be derived in part automatically at compile time, in part it has to be supplied manually. A prerequisite are tools which would allow sufficiently numerous implementation choices. No such tools exist to date, and there is not enough empirical data to judge if they can be made powerful enough to match the flexibility of hand-coded implementations. But we have already seen a number of possible improvements to the current tools.

4 Configuring protocol implementations for parallel execution

4.1 Exploitation of the specified concurrency

Standard Estelle offers several language constructs to express concurrency between modules. In an implementation, modules are mapped on the available processors of one or more given machines, which may lead to distribution and parallelism during execution. In this approach, we concentrate on the parallelism on one machine. We call the approach the configuring of protocol implementations for parallel execution.

It has to be decided up to which degree the specified concurrency is really exploited. Basically, there are three approaches: execute no, only some or all concurrent modules in parallel.

For the solutions implementing parallelism, there is also the choice of a static or dynamic mapping process. Dynamic strategies change the mapping of tasks onto processors depending on machine load and changes in the dynamic structure of the specification.

The first, sequential approach has been implemented in many existing Estelle compilers, e.g. [BGS87, VLC88, NBS87, SiBl90, SiSt93, Bud92, KrGo93]. It is good for single-processor implementations, but makes no use of possibly available multiprocessors[5]. In the third, straightforward approach for exploiting all possible parallelism, each Estelle module is mapped onto one parallel thread of the operating system [FiHo94, PHK95]. Dynamic mapping is realized by the operating system scheduler which usually tries to achieve a load-balanced system. Measurements show a good speedup, while much speed is lost by synchronization and communication overhead between the operating system threads.

Therefore, we present in this section a solution realizing the second approach which exploits only a certain degree of the specified concurrency, but does so "intelligently" by distributing *groups of modules* over processors/operating system threads in order to save resources. A dynamic strategy is realized by reconfiguring during runtime the assignment of modules to threads. Such "user scheduling" uses our knowledge of the specification structure and helps us to avoid idling of threads. Measurement results for all three approaches show significant performance gains using the (re-)configuring approach, which is described in more

[5]Some of them [SiSt93, KrGo93] allow the distribution of system module trees and even of single modules over a network. However, they aim more at distributed than at high-speed execution. Communication costs between modules are exceptionally high in these approaches. Recently, the tools presented in [KrGo93] have been extended to support also parallel implmentations on a transputer system.

detail in [FiEf95].

4.2 Module groups as parallel units

The straightforward solution does not take the specification structure into account. Three important issues limit the processor usage:

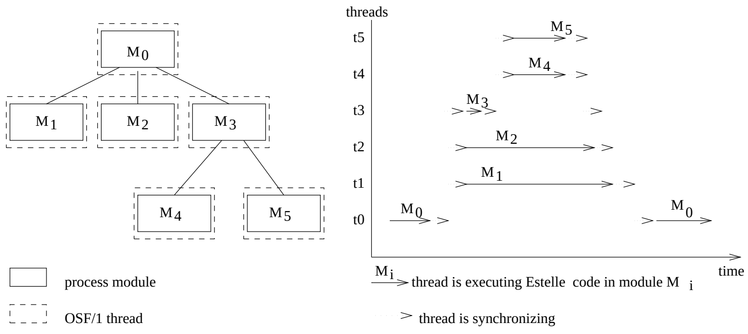
1. The computational complexity of modules running in parallel may be very different.
2. The computational complexity of some modules is so small that synchronization time cannot be neglected in comparison to protocol processing time.
3. Estelle's prohibition of parent–child concurrency makes at least one processor run idle. When the parent module passes the right to execute to its children, it waits for all responses, wasting one processor.

Please note that these three limitations exist only in the case of standard Estelle. If we employ the enhancement of asynchronous modules described in Section 2.2, issues 2 and 3 have less or no influence.

The solution for all three problems is the "intelligent" mapping of the specified concurrency onto operating system and machine parallelism. As a result, we get an increase in the overall system utilization, as there are more processors available for other modules.

From the three issues listed above, we derive the following **mapping rules**:

1. Threads running in parallel but synchronized due to Estelle semantics (i.e. children of the same parent) should have similar computational complexity. Threads which are ready will not have to wait a long time. For asynchronous modules (i.e. systems), we need no such rule, as threads running in such a way do not have to wait for each other.
2. Modules with greater synchronization than protocol processing time should not run exclusively in their own thread. Running them sequentially, together with another module, in one thread will save synchronization time.
3. The thread executing a parent module should always include one child module. This thread will then not be idle, but execute a child module in parallel to other threads running other child modules of the same level.



(a) Example thread mapping in the first appproach

(b) Thread synchronization in the first approach (6 processors)

Figure 8: Mapping of modules to threads in the first solution.

A comparison of the straightforward and the "intelligent" thread mapping schemes is illustrated in Figures 8 and 9. Let us assume that modules M_0, M_1, M_2, M_3, M_4 and M_5 have average execution times of 20, 80, 70, 10, 30 and 30 milliseconds, respectively[6]. Modules M_1 and M_2 have nearly equal protocol execution times, while module M_3 has much less. For that reason, in the new approach, it is grouped together with modules M_4 and M_5. Execution time in the traditional solution totals 100 ms — 20 ms for M_0 and 80 ms for M_1, which is the longest-running module of the three running in parallel — using six processors; in the new one, it also totals 100 ms — again, 20 ms for M_0 and 80 ms for M_1. The sequential execution of modules M_3, M_4 and M_5 only adds up to 70 ms, less than the execution time of concurrent module M_1. The clear advantage is that we only use three processors. The remaining three may be assigned to other modules.

To determine the runtime of protocol module implementations, we instrument modules with time measurement routines, allowing us to extract protocol execution times during runtime. We are thus able to detect modules which have very small execution times as well as those which are running in parallel, but have very different execution times[7].

The algorithm for dynamically (re-)configuring the parallel protocol software reads as follows. First, the generated protocol software is provided with necessary information about the machine (e.g. number of processors and characteristic timing figures). It is then run sequentially on one processor for a certain length of time with each module measuring its runtime. Based on these measure-

[6] We concentrate on average execution times. Experience shows that there is very little data dependency in protocol processing.

[7] Runtime measurements not discussed in Section 4.3 show that time measurements have only a negligible effect on runtime.

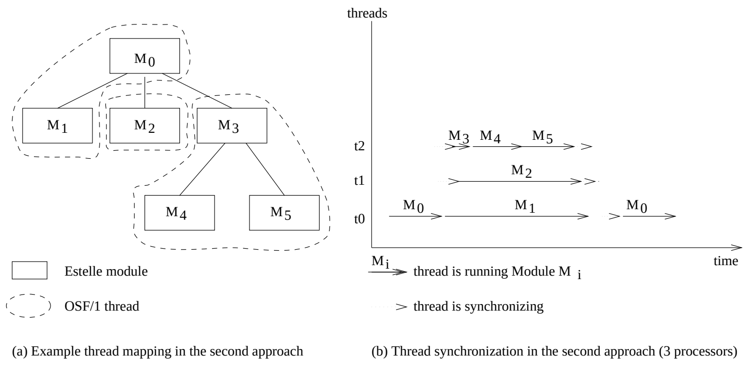


Figure 9: Mapping of modules to threads in the improved solution.

ments, the software is configured for parallel execution and run in parallel. Each module continues measuring its runtime, and from time to time, the software is reconfigured.

4.3 Experimental Results

To compare the new configuration methodology to the straightforward version, we performed measurements with a massively parallel specification. We implemented a dynamically configuring and a statically configured version as well as — for comparison purposes — a sequential version. The dynamic version was restricted to four processors. The sequential version used one processor, while, during one experiment, we had the static version run on four processors and, during another, on as many processors as there were module instances. For all experiments, we varied the specification between 3 and 17 modules (consisting of a root module and 2 to 16 parallel child modules). Thus, the statically parallel version used at most 17 processors. The results are shown in Figure 10.

The diagram shows that the new configuration methodology is superior to the statically parallelized implementation. It is not only better on the same number of processors, but also on much fewer processors[8]. We achieve a speedup between 3.3 and 3.5 compared to the sequential execution. This is very near to the optimum of four (the number of processors for the dynamic version). It is also obvious that it is unreasonable to use (many) more threads than there are processors available. The static version on four processors shows a dramatic runtime increase with an increasing number of threads.

These results have two important implications. First, parallel protocol imple-

[8]This last point is mainly due to the specification structure, which introduces a slight disadvantage for the parallel version because of synchronization variable accesses.

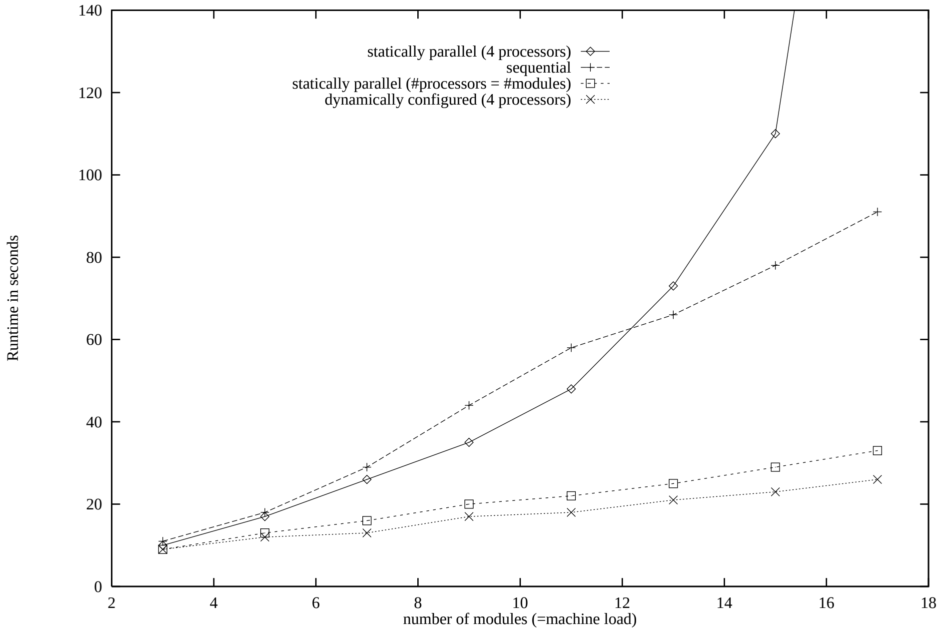


Figure 10: Runtime Comparison between different implementations.

mentations will not only be successful on massively parallel supercomputers such as the KSR or an Intel Paragon etc., but also on much lower priced multiprocessor workstations, equipped with two or four or even more processors, e.g. a SUN SPARCstation 20. Second, we could easily restrict the protocol execution on fewer processors and achieve the same performance. This may be used for Quality of Service issues. With our compiler, we can e.g. restrict a certain set of connections in a multimedia system (the non-time critical ones) to one or two processors and can then reserve the other processors for connections with very high performance requirements supporting the users' QoS specification.

5 Conclusions

Correctness and runtime efficiency are essential properties of software in general and of high-speed protocols in particular. Establishing correctness requires the use of FDTs during protocol design, and to prove the protocol code correct with respect to its formal specification. Another approach to boost confidence in the correctness of the implementation is to generate protocol code automatically from the specification. However, the runtime efficiency of this code is often insufficient. This has turned out to be a major obstacle to the use of FDTs in practice.

It is still an open question whether the use of FDT-based implementations of high-speed protocols is feasible. Comparisons with hand-coded implementations are difficult, mainly due to the fact that Estelle specifications describe closed worlds, which makes it a hard task to integrate Estelle code into existing environments or vice versa.

In this paper, we have collected, extended and interpreted a number of recent measures which improve the efficiency of protocol code generated from Estelle specifications. We have shown how the runtime efficiency of code generated automatically from a formal protocol specification can be improved significantly by a number of measures during design, implementation, and runtime. In particular, we have addressed the influence of specification styles, issues of compiler and runtime system optimization, and the dynamic configuring of protocol modules for parallel execution on multiprocessor platforms. For best results, these measures should be harmonized. However, it remains an open question which measures may be combined and to what extent each measure should be applied to the specification.

Apart from the measures listed, there is significant potential for efficiency improvements in the following two areas. Firstly, runtime efficiency largely depends on the protocol itself, which should be designed for high-performance execution. An example here is XTP, the Xpress Transfer Protocol, as discussed above. Secondly, the Estelle specification can be used as a basis for the development of protocol hardware, which is of particular interest for lower layer protocols. In [Wyt95], it is shown how Estelle can be translated to VHDL [Coe89], a standard hardware description language.

In summary, this work shows the large potential of Estelle and Estelle-tools for improving the efficiency of Estelle-based implementations. Up to now, this potential has been only partially discovered and exploited.

References

[BGS87] Bochmann, G., Gerber, W., and Serre, J.-M. *Semiautomatic implementation of communication protocols*. IEEE Trans. Softw. Eng. **SE–13**(9), 989–1000 (Sept. 1987).

[BjGu93] Björkmann, M. and Gunningberg, P. *Locking effects in multiprocessor implementations of protocols*. In "Proc. of ACM SIGCOMM 93". ACM (Sept. 1993).

[Bre95] Bredereke, J. *Specification style and run-time efficiency in Estelle*. Tech. Rep. 245/95, Univ. of Kaiserslautern, Dept. of Comp. Sci. (1995).

[BrGo94] Bredereke, J. and Gotzhein, R. *Increasing the concurrency in Estelle*. In Tenney et al. [TAU94].

[BrZi92] Braun, T. and Zitterbart, M. *Parallel Transport System Design*. In Danthine, A. and Spaniol, O., editors, "4th IFIP conference on high performance networking, Liège, Belgium", pp. H3:1–H3:16 (1992).

[Bud92] Budkowski, S. *Estelle development toolset*. Comp. Networks and ISDN Syst. **25**(1) (1992).

[ChAm91] Chamberlain, S. C. and Amer, P. D. *Broadcast channels in Estelle*. IEEE Trans. Comput. **40**(4), 423–436 (Apr. 1991).

[ClTe90] Clark, D. D. and Tennenhouse, D. L. *Architectural Considerations for a New Generation of Protocols*. In "SIGCOMM '90 Symposium Communication Architectures & Protocols", pp. 200–208, Philadelphia (Sept. 1990).

[CLV93] Chanson, S. T., Loureiro, A. A. F., and Vuong, S. T. *On tools supporting the use of formal description techniques in protocol development*. Comp. Networks and ISDN Syst. **25**, 723–739 (1993).

[Coe89] Coelho, D. *The VHDL Handbook*. Kluwer Academic Publishers (1989).

[DDK+90] Doeringer, W., Dykeman, D., Kaiserswerth, M., Meister, B., Rudin, H., and Williamson, R. *A survey of light-weight transport protocols for high-speed networks*. IEEE Trans. on Communic. **38**(11), 2025–2039 (Nov. 1990).

[DeBu89] Dembinski, P. and Budkowski, S. *Specification language Estelle*. In Diaz, M. et al., editors, "The Formal Description Technique Estelle", pp. 35–75. North-Holland (1989).

[DeŚr95] Dembiński, P. and Średniawa, M., editors. *PSTV '95, Proceedings*, Warsaw, Poland (13–16 June 1995).

[FBR93] Frank, S., Burkhard, H., and Rothnie, J. *The KSR1: High performance and ease of programming, no longer an oxymoron*. In Meuer, H.-W., editor, "Supercomputer '93", Informatik aktuell, pp. 53–70. Springer, Heidelberg (1993).

[FiEf95] Fischer, S. and Effelsberg, W. *Efficient configuration of protocol software for multiprocessors*. In Puigjaner, R., editor, "Proceedings of hpn'95", Palma de Mallorca, Spain (Sept. 1995). To appear.

[FiHo94] Fischer, S. and Hofmann, B. *An Estelle compiler for multiprocessor platforms*. In Tenney et al. [TAU94], pp. 171–186.

[Fis95] Fischer, S. *On the suitability of Estelle for multimedia systems*. In Dembiński and Średniawa [DeŚr95], pp. 349–364.

[HeKö95] Held, T. and König, H. *Increasing the efficiency of computer-aided protocol implementations*. In Vuong, S. T. and Chanson, S. T., editors, "Protocol Specification, Testing and Verification XIV", pp. 387–394. Chapman & Hall (1995).

[Hof93] Hofmann, B. *Integration von Darstellungs- und Kommunikationssteuerungsschicht in Estelle*. In Gerner, N., Hegering, H.-G., and Swoboda, J., editors, "Kommunikation in verteilten Systemen", pp. 560–573, Munich (1993). GI/ITG-Fachtagung, Springer.

[ISO89] ISO/TC 97/SC 21, ISO 9074. *Information Processing Systems — Open Systems Interconnection — Estelle: A Formal Description Technique Based on an Extended State Transition Model* (1989).

[KrGo93] Kreuz, D. and Gotzhein, R. *A compiler for the parallel execution of Estelle specifications*. In König, H., editor, "Formale Methoden für Verteilte Systeme", vol. 8 of "FOKUS Praxis, Information und Kommunikation", pp. 161–178. K. G. Saur, Munich (1993).

[LFV95] Lallet, E., Fischer, S., and Verdier, J.-F. *A New Approach for Distributing Estelle Specifications*. In V.Bochmann, G., Dssouli, R., and Rafiq, O., editors, "Proceedings of the 8th Int. Conf. on Formal Description Techniques, Montreal, Quebec, Canada" (Oct. 1995). To appear.

[NBS87] NBS. *User guide for the NBS prototype compiler for Estelle*. Final report ICST/SNA–87/3, Institute for Computer Science and Technology, National Institute for Standards and Technology (Oct. 1987).

[PHK95] Plato, R., Held, T., and König, H. *PARES– a portable parallel Estelle compiler*. In Dembiński and Średniawa [DeŚr95], pp. 383–399.

[PPVW92] Plagemann, T., Plattner, B., Vogt, M., and Walter, T. *A Model for Dynamic Configuration of Light–Weight Protocols*. In "IEEE Third Workshop on Future Trends of Distributed Computing Systems" (1992).

[SDW92] Strayer, W. T., Dempsey, B. J., and Weaver, A. C. *XTP The Xpress Transfer Protocol*. Addison–Wesley Publishing Company, Reading, Massachusetts (1992).

[SiBl90] Sidhu, D. P. and Blumer, T. P. *Semi-automatic implementation of OSI protocols*. Comp. Networks and ISDN Syst. **18**, 221–238 (1990).

[SiSt93] Sijelmassi, R. and Strausser, B. *The PET and DINGO tools for deriving distributed implementations from Estelle*. Comp. Networks and ISDN Syst. **25**(7), 841–851 (Feb. 1993).

[Svo89] Svobodova, L. *Implementing OSI systems*. IEEE Jour. on Sel. Areas in Commun. **7**(7), 1115–1130 (1989).

[TAU94] Tenney, R. L., Amer, P. D., and Uyar, M. Ü., editors. *Formal Description Techniques, VI*. North-Holland, Amsterdam (1994).

[Ten90] Tenney, R. L. *Adding interaction sets to Estelle*. In Quemada, J., Maños, J., and Vazquez, E., editors, "FORTE '90", pp. 477–483, Madrid, Spain (5–8 Nov. 1990). North-Holland.

[VLC88] Vuong, S. T., Lau, A. C., and Chan, R. I. *Semiautomatic implementation of protocols using an Estelle-C compiler*. IEEE Trans. Softw. Eng. **14**(3), 384–393 (Mar. 1988).

[VSvS88] Vissers, C. A., Scollo, G., and van Sinderen, M. *Architecture and specification style in formal descriptions of distributed systems*. In Aggarwal, S. and Sabnani, K., editors, "PSTV '88", pp. 189–204, Atlantic City, N.J., USA (1988).

[Wyt95] Wytrebowicz, J. *Hardware specification generated from Estelle*. In Dembiński and Średniawa [DeŚr95], pp. 417–432.

[Zit89] Zitterbart, M. *High–Speed Protocol Implementations Based on a Multiprocessor–Architecture*. In Rudin, H. and Williamson, R., editors, "Protocols for High–Speed Networks", pp. 151–163, Zürich (1989). IFIP WG 6.1/WG 6.4, North-Holland.

[ZST93] Zitterbart, M., Stiller, B., and Tantawy, A. *A model for flexible high–performance communication subsystems*. IEEE Journal on Selected Areas in Communications **11**(4), 507–518 (May 1993).