REIHE INFORMATIK
5/95

On the Suitability of Estelle for Multimedia Systems

Stefan Fischer
Universität Mannheim
L 15,16/IV.Stock
D-68131 Mannheim

On the Suitability of Estelle for Multimedia Systems

Stefan Fischer
University of Mannheim, Praktische Informatik IV
D-68131 Mannheim, Germany
Phone: +49 621 292 5053 Fax: +49 621 292 5745
`stefis@pi4.informatik.uni-mannheim.de`

Abstract

Formal Description Techniques have been widely used for the specification of traditional networked applications. They have not been applied to the specification of new applications such as multimedia systems yet. In this paper, we examine the FDT Estelle with respect to its suitability for multimedia system specification and automatic derivation of efficient implementations. We show that it is possible to specify certain aspects of multimedia systems, but that Estelle is not sufficient for others. The derived implementations often perform badly. We show the reasons and propose to use a slightly modified Estelle syntax and semantics to solve the problems. The implemented solution was tested successfully.
Keywords: Estelle, Multimedia Systems, Quality of Service, Implementation

1 Introduction

Formal Description Techniques (FDTs) have been successfully used to specify protocols for data communication. Numerous tools have been developed to analyze formally specified protocols and to derive implementations from such specifications (semi-)automatically, e.g. [BGS87, BT82, VLC88, SB90]. For most applications performance was a minor issue in contrast to reliability, which was perfectly offered by FDT implementations. A good example of this class of applications is electronic file transfer. It is a major issue that all data are transferred correctly without strict constraints on a certain throughput or delay. Therefore, FDTs were mainly designed to allow specification of functional behavior.

In recent years, however, the situation has changed dramatically. With the advent of high-speed networks a new class of applications was found to be implementable that has very special needs with respect to *quality of service* (QoS). One of the best-known representatives of this class, distributed *multimedia systems*, requires the handling of very different kinds of media which are all transferred over the same network. For some of these media, connections are required which offer the same quality as in earlier networks. Examples include text and data files or graphics. However, there are also *continuous media* with totally different requirements.

These requirements are not expressed as functional properties but are time-related, so-called *non-functional* properties.

Existing formal languages are often unable to express those non-functional properties of a system, mainly because they have no notion of *time*. Several ideas were proposed to introduce time into formal languages. Especially enhancements for LOTOS [ISO87] and different kinds of temporal logic (for an overview see [Got92]) have been discussed intensively. The spectrum ranged from adding time to the language itself, e.g. [QF87, LL94], to hybrid approaches, i.e. writing specifications in two languages (e.g. LOTOS and temporal logic [BBBC94]).

Logic and algebra–based languages are very well suited for specification purposes. Requirements may be expressed elegantly. However, it is often difficult or impossible to derive efficient implementations automatically from these specifications. For these purposes, automata-based languages are often better suited [BBBC93]. The derivation of efficient implementations is an important issue in protocol engineering. Thus, we concentrate in this paper on the automata-based language Estelle [ISO89].

We will examine the suitability of Estelle for the formal specification and automatic implementation of multimedia systems. Estelle already has a concept of time: it is possible to *delay* transitions of Estelle modules for a certain amount of time. In traditional applications, this feature was used to specify timeouts. We will investigate the usefulness of this time concept for the above-mentioned purposes.

The paper is organized as follows: Section 2 gives a brief overview of characteristics and requirements for distributed multimedia system. In Section 3, we show an approach of how to specify these typical characteristics in Estelle using the language's delay operator. We show that it is possible to model some of the quality of service aspects, but that many may not be sufficiently (if at all) specified. In Section 4, we derive implementations from the developed specifications automatically for different architectures to check if at least the set of specifiable characteristics may be found again in the implementation. Performance measurements show that the intended behavior is not achieved. We identify several reasons for these results, and give, in Section 5, a solution for the problem introduced by the language, which only needs relatively small changes to the language. Measurement results show the usefulness and suitability of the new approach. Section 6 concludes the paper and discusses some ideas to improve the expressiveness of the language with respect to the whole set of QoS parameters.

2 Characteristics of Multimedia Systems

Multimedia systems are characterized by the use of very different kinds of media. Apart from *time-independent* media like text or graphics, *continuous* media play an important role, e.g. video and audio, where parts of the data stream (e.g. pictures of a movie) are related to each other in the time dimension. Human senses, namely visual and audial perception, impose strict constraints on their transfer and presentation. A video has to be transferred and presented to the user at a rate of at least 16 to 25 pictures per second to create the impression of a movie. The intervals between one picture and the next should be nearly equal for all pictures to create the impression of uniform progress.

To express these new requirements in a machine-readable form, new *quality of service parameters* have been developed, e.g. [HSS90]. A user may express his requirements on media transfer or presentation by specifying values for the QoS parameters. The most important of these parameters are:

- **Throughput**
  Movies consist of a sequence of pictures. In distributed systems, these pictures have to be transferred subsequently from the source to the sink computer. Even a single colored picture occupies a large amount of memory. In addition, 25 pictures have to be transferred every second. Considering a picture size of 640×480 pixel with 24 bit/pixel representing the color of the pixel, a throughput of about 175 MBit/s will be necessary. Even when this high value is reduced by data compression, the system needs a *guaranteed* throughput of several MBit/s.

- **Transfer Delay**
  In some applications, the time between the production of data at the source and its presentation at the sink is subject to very stringent bounds. These bounds are expressed by the transfer delay. An important example is a telephone call. Delays longer than several tenths of a second are unacceptable as they make conversation impossible.

- **Jitter**
  The jitter expresses the variance of the transfer delay. For movie presentation, jitter should be very slight to create the impression of uniform progress of the movie. Jitter can be reduced in the end systems by the use of buffers. However, these buffers have to be quite large, thereby requiring large memory resources. Thus, it is better to have the jitter already controlled by the network itself [Fer92].

- **Error Rates**
  The transfer reliability for continuous media may be lower than 100%, as pixel or block errors or even whole missing pictures will not be noticed by the user while watching a movie. Missing blocks in an audio transfer result in a short noise. Retransmission of missing blocks is often not useful. By the time of the arrival of the retransmitted data, its presentation time has already passed.

Apart from these parameters, new service semantics have been found to be necessary [DBLL92, Fer90, Kur93]. For data communication, only *best-effort* services have been offered, where the service provider tries to do its best to maintain the requested service but cannot give any guarantees. In the *guaranteed* service, the service user can rely on the initially negotiated service characteristics. Apart from that, a *statistical service* has also been discussed that gives guarantees like “Not more than 10% of all packets will have a delay longer than 10 milliseconds.”

In the next section, we will try to express the provision of a certain quality of service and the related service semantics in Estelle. We examine the parameters and semantics introduced above.

3 Specification of Multimedia Systems in Estelle

To allow the specification of non-functional behavior, formal techniques need a notion of time. In Estelle, the only way to express time relations is the delay operator. Informally, the semantics of *delay(E1,E2)* in a transition t are described roughly as follows in [ISO89]:

1. Once newly enabled, t cannot be executed until it remains enabled for at least $E1$ time units.
2. If t remains enabled but is not fired for E time units, $E1 \leq E \leq E2$, then even if t is the only enabled transition within a module instance at the moment, t still may or may not be executed.
3. If t has been enabled for E time units, $E \geq E2$, then if t is the only enabled transition, t will fire. Of course, any other enabled transition may fire, too.

It is possible to omit the second parameter: *delay(E1)* is equivalent to *delay(E1,E1)*.

To assess the suitability of the delay operator, we consider a sample specification of a multimedia system. It consists of a sender and a receiver connected through a channel. The sender is required to send a data stream with a throughput of $3.2MBit/s$ isochronously, corresponding to one message of 4k size every 10 milliseconds. For this purpose, we use the leaky-bucket algorithm known, for example, from the XTP protocol. The receiver is required to receive the data stream with a jitter of at most $2ms$. We leave out the channel specification in this example. The sender and receiver specification may be found in Figure 1. Within the context of the Tempo project at the University of Lancaster [BBBC93], Estelle has already been briefly examined for its suitability for specifying multimedia systems. Parts of the example are based on these ideas.

The new QoS parameters introduced in Section 2 have been modeled in the following way:

- **Throughput**
  The provision of a certain throughput is expressed by a periodically selected and executed transition. The transition is prioritized, ensuring its selection when other transitions are enabled. In the example, the send transition always has priority over the transition receiving input from a higher layer. It should be selected every $10ms$. Thus, a possibly bursty user input is smoothed by the algorithm. By following this approach, however, we can, due to the semantics of the delay operator, only introduce an upper bound on the throughput. The transitions will not fire more frequently than every $10ms$. As a consequence of item (3) in the description of the delay operator, the transition may also fire after a longer delay without violating the specification. Thus, Estelle does not allow to express an exact intended timing behavior for the throughput. The specifier cannot express his needs exactly. In addition, there is no way to find out how long the delay really was as no time variables exist.
- **Transfer Delay**
  For controlling the provision of a certain transfer delay, one has to relate two events,

i.e. the sending of the data from the source, and its arrival at the sink. In an Estelle specification, these two events take place in two different modules. They may not be related to each other as the delay operator can only refer to one transition in one single module. We note that, in general, it is impossible to describe timing relations between events in different Estelle modules. This is an important disadvantage of the language.

```
module sender_type process;
ip   up: up_channel(provider);
     down: down_channel(user);
end;

body sender_body for sender_type;
var
      buffer : array[1..maxContent] of Message;

trans
    priority LOW
    when up.msg
    provided buffer_not_full begin
        insert_into_buffer(msg);
    end;

    priority HIGH
    delay(10)
        output down.msg(buffer[first]);
        remove_msg_buffer;
    end;
end;
```

```
module receiver_type
ip down: down_channel(user);
end;

body receiver_body for receiver_type;
state WAITING, RECEIVE, ERROR;

trans
    from WAITING TO RECEIVE
    delay (10)
    begin end;

    from RECEIVE TO WAITING
    when down.msg begin
        use_msg;
    end;

    from RECEIVE TO ERROR
    delay (2) begin
        (* message arrived too late *)
    end;
```

Figure 1: Specification of sender and receiver in the example

- **Jitter**
  The jitter in the receiver module is controlled in the following way: $10ms$ after receipt of the last message, the module enters the receive state again, waiting for the next message. If no message arrives after another $2ms$, the module enters en error state as the message has not arrived during the specified period.

- **Error Rates**
  Certain specified error rates, e.g. “No more than 10% of all packets should be lost.”, may be expressed by two counters, one for the packets arrived and one for those lost. Every time a packet arrives in order or does not arrive, the respective counter is incremented.

The relation of both counters models the error rate and may be checked in a `provided` clause of the transitions.

Speaking in terms of service semantics, Estelle can only provide best-effort service. Leaving out any delay clauses in the transition descriptions enables fast transition execution but it is not possible to give any guarantees on lower bounds for throughput or upper bounds for jitter. However, in some cases it is possible to check if specified values have been maintained and to signal violations to the user (see the item on error rates). Thus, it is possible to specify enhanced best-effort services in Estelle, but no guaranteed services.

How useful are the modeling possibilities for important aspects of specification language usage? We consider the following aspects: clearness and correctness, validation and verification and automatic implementation.

- **Clearness and Correctness**
  The delay operator provides the only way to specify throughput and jitter aspects. For an experienced Estelle reader, the intention of the specifier may be derived from the specification. However, the relation between the delay operator and the intended behavior is not as obvious as it is with other language constructs. In addition, the specifier is not able to express more than this intended behaviour. There is no way to enforce the provision of a guaranteed transition firing time after the delay timer has expired. Implementations may thus conform to the specification, but not, on the other hand, really implement the intended behavior. This is due to the semantics of the delay operator.

- **Validation and Verification**
  The semantics and underlying model of Estelle make verification difficult, if not impossible. This is also true if time aspects are examined. Validation, however, is easy with Estelle. The validation of timing aspects adds a further degree of complexity. It is often necessary to include the protocol's runtime environment characteristics in the experiments. The tester of a multimedia system is often more interested in the protocol's performance than in its correctness. Most validation tools do not support performance examinations; often, their code structure prevents rapid execution. Thus, it is often the tool itself that prevents testing of performance aspects.

- **Automatic Implementation**
  The task of an implementation is to conform to the specification and to be efficient. Speaking in terms of the delay operator, it is important that transitions not fire before the delay time has passed (conformity), but that they fire immediately thereafter (efficiency). In the next section, we will assess implementations with respect to these criteria, i.e. we find out how good the efforts made by the implementation to achieve the specified parameter values really are.

To summarize, the use of the delay operator is very limited with respect to multimedia system specification. In Section 6, we briefly present some ideas of how to improve the expressiveness of Estelle specifications concerning timing requirements. In the remainder of this paper, however, we concentrate on the given Estelle features.

4 Implementation Problems with the Delay operator

In this section, we answer the question whether automatic implementations of multimedia systems specified with Estelle's delay operator do indeed implement the intended behavior. We do this by deriving code from the specification outlined in Figure 1 and by measuring its performance. We concentrate on throughput as our main evaluation focus. To get a characteristic impression, we perform the measurements on a variety of hardware architectures and use, where possible, different Estelle code generators, resp. runtime environments.

Our specification consists of a root module and a number of children modules. One half of the children modules is sending messages of a certain size with a certain delay while the other half is receiving the messages. A single sender looks like the one in Figure 1, while the receiver is simplified by not controlling the jitter. We call a pair comprising a sender and a receiver a connection. The sender module mainly executes one transition in which messages of $maxData$ size are output sequentially:

```
MessageType = array[1..maxData] of char;
...
from sending to same
delay (x)
begin
    output port.msg(message);
end;
```

This means that a message of a size of $maxData$ bytes is sent every x milliseconds to achieve a throughput of $maxData \times 8 \times \frac{1000}{x} Bit/s$. From the Estelle semantics of the delay operator, it is clear that this is an upper bound for the throughput, because the transition becomes firable *after* x milliseconds have elapsed. However, it is intended that the transition should fire as quickly as possible after becoming firable. The goal of our measurements was to find out how long it took to select and execute the "right" transition, i.e. the period of time between x and the actual firing time.

We performed several measurements with one, two and three connections on different architectures. Our machines were a DECstation 5000/133 running Ultrix 4.3, an Intel PC 486DX33 running Linux 1.0.9, an IBM RS/6000 running AIX 3.2, a SUN SPARC 10 running Solaris 2.3 and a KSR1 [FBR93] equipped with 32 processors running OSF/1 1.3. Our software was the Pet/Dingo System from NIST [SS93], the EDT [Bud92][1], and, for the parallel machine, our modified Pet/Dingo [FH94]. For the `delay` parameter, we used the values 0, 10 and 20 milliseconds. The results of our measurements are given in Table 1.

For two of the machines, the SUN Sparc and the KSR, we performed some additional measurements to show the effects more clearly and in graphical form. We varied the delay from 0 to 40 milliseconds and the message load from 1024 to 9216 Bytes. The results may be seen in Figures 2, 3 and 4. The dotted lines show the optimal results, i.e. what was intended by the specification; the normal lines show the measured results.

[1]Those measurements were not possible when using delays, as in the current version of EDT, it is not possible to specify delays less than 1 second. So, EDT was only used for measurements without delays.

specified delay	connections	Pet/Dingo					EDT
		DEC	486 DX33	IBM	SPARC with PD	KSR1-32	SPARC
–	1	7.2	5.6	25.0	2.9	10.2	0.5
	2	14.1	10.8	42.3	5.8	11.8	0.9
	3	21.5	16.3	60.1	8.6	13.8	0.12
10	1	17.3	20,2	32.1	13.0	20.0	–
	2	21.6	30.4	50.0	15.5	20.1	–
	3	24.1	40.3	68	17.6	29.6	–
20	1	25.7	30.0	47.1	24.5	31.7	–
	2	32.2	40.3	51.5	27.5	31.9	–
	3	33.1	50.4	68.4	30.4	42.5	–

Table 1: Measured delay compared to specified delay (in ms/transition)

Obviously, the intended behavior is not at all achieved by all the implementations. Even with the fastest machine, the SUN Sparc, a further delay of at least 3 milliseconds per transition is introduced by the implementation. As a result, we get a throughput which is at least 23% less than intended in the case of a specified throughput of 100 messages per second (x=10). Comparing the lines in the tables for an increasing number of connections and Figures 2 and 3, we see that the situation becomes even worse.

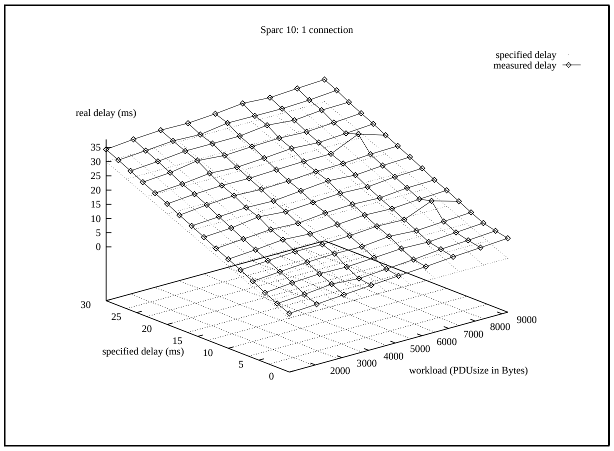

Figure 2: SPARC10 - 1 connection

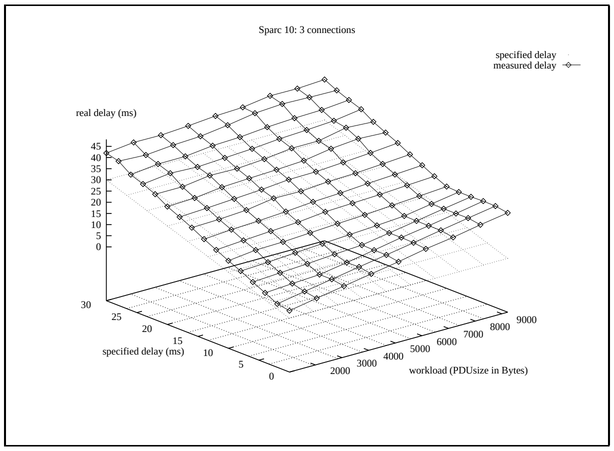

Figure 3: SPARC10 - 3 connections

Other problems are introduced by drawbacks of the operating system. This can be seen in Figure 4. To implement delays, one has to measure the elapsed time. In Unix systems, this is usually done by calls of the routine `gettimeofday()` (or similar). Unfortunately, the implementation on some systems (here in OSF/1 on the KSR) has a very coarse granularity of about 20 ms. This results in the typical shape of the curve in Figure 4.

What are the reasons for the performance problems? We identified two main reasons: the implementation environment and Estelle itself.

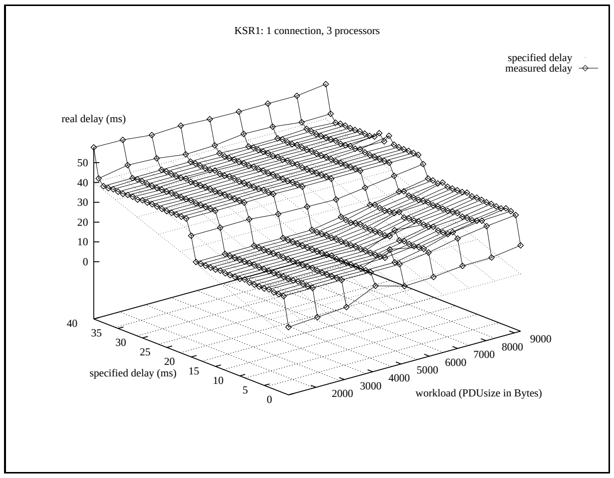

Figure 4: KSR1 - 1 connection, 3 processors

We already pointed out the problems introduced by some operating systems. Sometimes they can be solved by using other routines for time measurement. The KSR, for example, offers a better call which directly accesses the hardware. Other problems are often insoluble. When interprocess communication is used, one has to use the Unix `select()` call. On some systems, this call performs poorly, too.

A typical problem for currently available FDT compilers and runtime systems is their orientation towards simulation. In addition, many features are programmed inefficiently to allow for a closer mapping of Estelle to the target language. A typical problem is the implementation of the transition selection algorithm. This may be done by programmed access using hard-coded *if*-statements or by table-controlled access. The latter is very difficult to read but is very efficient [HK94]. Minor problems are introduced by the use of object-oriented languages like C++, which often need more processing power.

The second, more important, point is the problem introduced by the language Estelle itself. In principle, we identified two main problems.

The first problem (which we already encountered in the specification phase) is introduced by the semantics of the delay operator. Let us recall: when a transition t is guarded by the expression *delay(a,b)*, then it may not fire between the time e when t became enabled and $e + a$. It may fire between $e + a$ and $e + b$ and it has to fire after $e + b$, at least, if no other transition is enabled. However, the semantics give no statement about the exact point in time when the transition has to fire. This is considered to be implementation-dependent.

To understand the consequences for the implementation, we look again at our sample specification from the beginning of this section. We consider a specification consisting of a root module and two connections. The delay value in the sender module is $10ms$. We assume the following times needed for module execution: the root module r needs $3ms$ to select a transition (it never executes a transition, as all the work is done in the children). The sender modules s_1 and s_2 need $3ms$ for transition selection and $8ms$ for transition execution. The receiver modules r_1 and r_2 need $4ms$ for selection and $6ms$ for execution. We assume the whole specification is running on a single processor machine. A possible execution trace is depicted in Figure 5. The trace tells us when transitions may be fired in Estelle terms and when they are actually fired in "real time" terms. We are only considering the sender transitions. In the first Estelle cycle beginning at $0ms$, the two sender transitions become enabled (depicted by the two dots). They cannot be fired as they are guarded by a $10ms$ delay. This first cycle lasts $17ms$ until all modules have executed their part of the cycle. After $10ms$ during this time, the two enabled transitions should be executed speaking in terms of "real time" (dots at $10ms$). However, they may not be fired now, as (1) the sender modules do not have control, and (2) following Estelle semantics, the modules do not "know" that some time has already passed. Due to the notion of a **system snapshot**, they are all working with the same time for one cycle, which is, in this case, still $0ms$. One of the consequences of this time concept is that delayed transitions may only be fired every other cycle, independent of the specification structure.

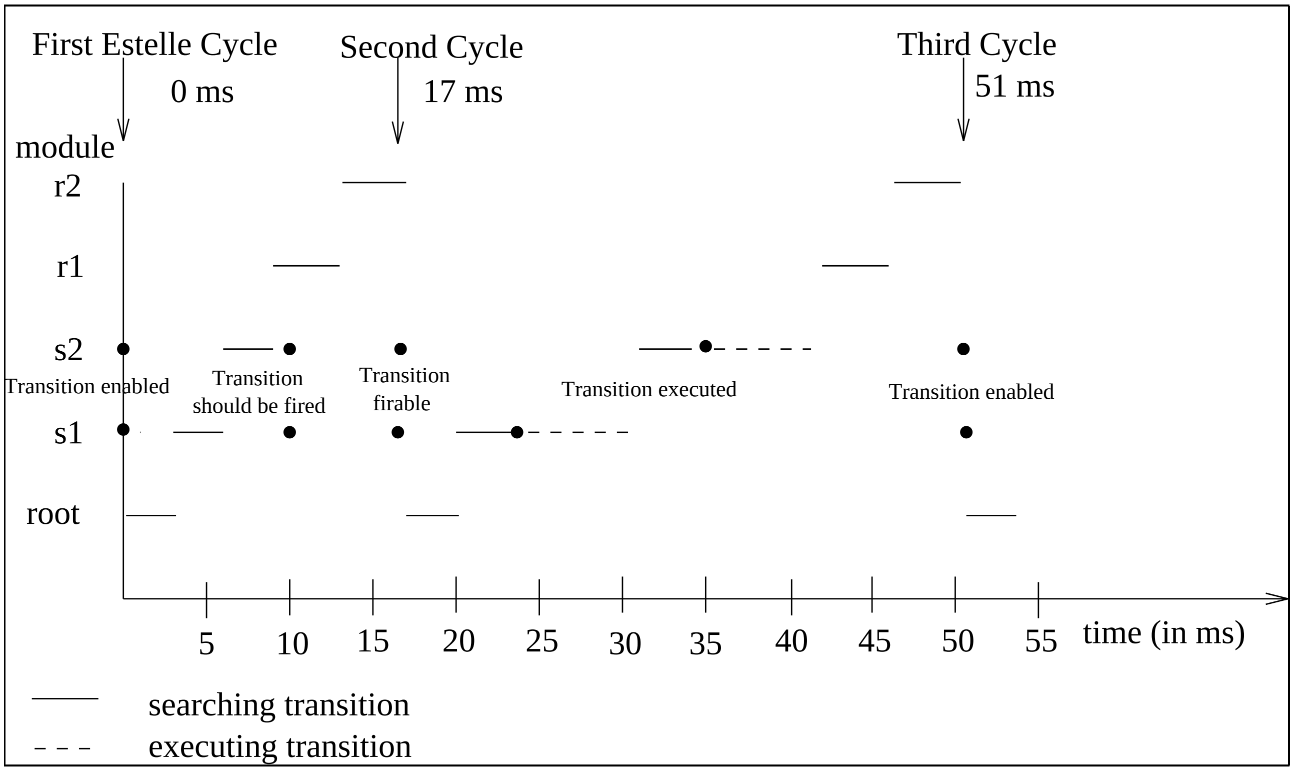

Figure 5: Time diagram for a sample specification

At $17ms$, the second Estelle cycle starts. All modules know that $17ms$ have passed since the last cycle. That means that the two sender transitions may now be fired (dots at $17ms$). However, the two sender modules do not have control. They have to wait another $7ms$ resp. $18ms$ until they may fire their transition. Afterwards, this Estelle cycle lasts another $16ms$ before a new cycle starts and the transitions become newly enabled.

Obviously, the semantics of the delay operator are correctly implemented, but this does not result in the intended behavior. We conclude that the contrast between the concept of a constant time during one system snapshot and the passage of real time during this cycle prevents the implementation of real-time characteristics.

The second problem is related to the first, and is due to Estelle's concept of parallelism. Asynchronous parallelism, i.e. modules which may run independently of each other, is only possible between system modules. However, the number of system module instances is static after system initialization. Synchronously parallel modules may be initiated during specification runtime, but they depend on their parent module. When a parent module is active, its children may not be active, and when the parent module passes the right to execute to its children, it has to wait until all children have done their work. Thus, the performance of a module is influenced by its ancestors, i.e. by its position in the module hierarchy, and by the performance of its siblings. These characteristics make it difficult to describe typical requirements of high-speed applications: the performance of one connection should not influence the performance (and thereby the provision of a certain quality of service) of another connection[2]. This is also true for the whole system, which means that any other parts somewhere in the hierarchy should not influence a connection. This is, however, the case when we implement the Estelle semantics.

In the next section, we show how the second problem may be solved and that, with this solution, the first problem will, to some extent, disappear.

5 A Solution

A solution to the language problem described in the last section has to take into account the following points:

1. The position of a module in the hierarchy should have no impact on the module performance. The position should be understood as a possibility for the specifier to structure his work. The synchronous parallelism stands against that.
2. Parallel modules should not influence each other.

Obviously, Estelle in its original form cannot fulfill both requirements. Thus, one has to think of either using another language or adapting the language itself. We chose the second option because of the already mentioned advantages of Estelle. However, any adaptation should be as minimal as possible and should fit into the current Estelle semantics. Otherwise, the advantage of using a standardized language is lost. In addition, it is desirable to use an enhancement which is already well known and part of the current standardization discussion.

Thus, we propose in this paper to use as a solution to the language problem described in Section 4 the concept of `asynchronous processes` described by Bredereke and Gotzhein [BG94]. In their work, the authors propose adding the new keyword `asynchronous` to the language. Estelle `process` and `systemprocess` modules may be attributed additionally with this keyword. The semantics for `asynchronous systemprocess` modules are that they have a child module which is of type `asynchronous process`. The semantics of `asynchronous process` modules is that they are no longer synchronized with their parent module. The effect is that these modules

[2]The problem cannot be solved by specifying each connection as an `asynchronous` module. In this case, the number of connections would be static, and no new connections could be opened during runtime.

actually attain the status of `systemprocess` modules, with the difference that they still may be created dynamically. They are running their own Estelle cycles.

With these new syntax and semantics, the second problem in the last section is solved. An `asynchronous` module is no longer dependent on its ancestors. Parallelism can be used without the constraints of the synchronous semantics of standard Estelle. Thus, the specifier is free to use module hierarchies as a structuring means without having to watch performance aspects.

In addition, we get a partial solution to the first problem. All the asynchronous modules are running their own Estelle cycles. That means that the main Estelle cycle (of the system module tree) becomes shorter, as the asynchronous modules are excluded. Typically, time-critical modules will no longer be included in this cycle, so the fact that the latter execute their own cycle is much more important. When only one module is executed during a cycle, the cycle will be very short. This has two main benefits: first, a module runs more often, having the possibility of firing more transitions, and second, the checking of delay values can be done at a much finer granularity. Modules will not have to wait for other modules to complete. Instead, they assume much better control of their timers and will be able to react very much faster on the expiration of a timer.

To make use of the additional performance offered by the new concept, it is not useful to simply add, in the implementation, the asynchronous modules to the list of system modules and execute this list sequentially. There will be no performance gain at all. Instead, there are generally two ways to improve performance:

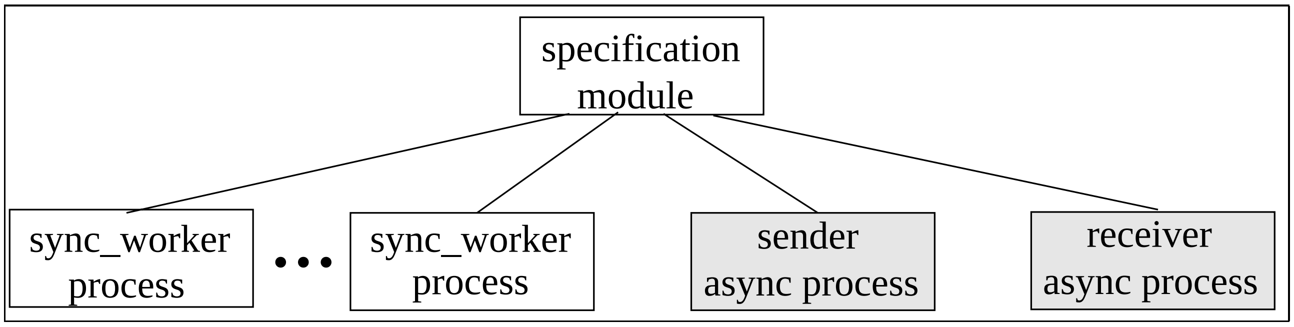

Figure 6: Module structure of the example

1. Execute some modules more often than others.

2. Use parallelism in the implementation, too. Single modules or groups of modules will be mapped onto an operating system (lightweight) process. To influence the performance of the time-critical modules, the implementor has several possibilities. On a multiprocessor system, processes or threads may be assigned their own processor. A single thread would then be able to execute its module(s) as quickly as necessary. If there are constraints on the number of processors, especially, if there is only one processor, one could work with priorities. Threads executing time-critical modules will have higher priority than others. A multiprocessor system, however, is the ideal machine for the new parallelism concept.

We implemented the asynchronous variant on the KSR multiprocessor and performed several measurements to prove that it is a better solution for the derivation of efficient multimedia systems implementations. We made sure of enough processing power and thus avoided priority issues. For our measurements, we used a slightly modified specification with respect to Sections 3 and 4. We still have a root module and one of the above connections with real time requirements, but in addition, we have several modules performing some other work that do not require such strict timing constraints. These modules may, for example, model connections for sending or receiving of text or directory services. The real time connection modules were attributed `asynchronous process` while the other modules had the normal `process` attribute. The example's module structure is visualized in Figure 6. Measurements with this specification have been done on a SUN Sparc (ignoring the `asynchronous` keyword) and on the KSR, once with the standard Estelle and once with new variant. The number of synchronous modules has been varied between 0 and 6. The results may be found in Figures 7, 8 and 9.

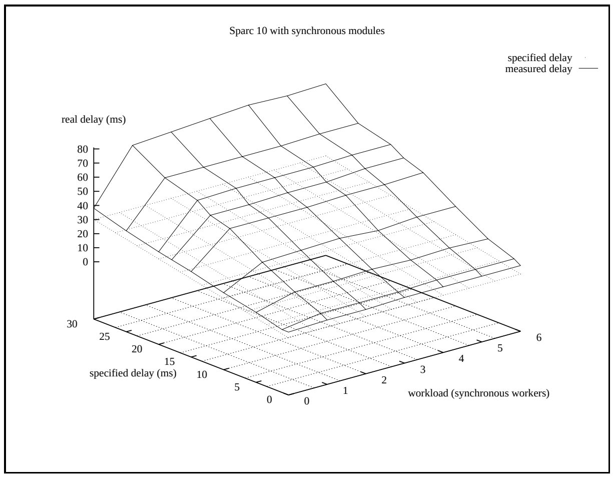

Figure 7: SPARC10: performance of the real time sender with varied other load

Figures 7 and 8 show that without other modules, the performance of the real time modules is quite good. As soon as we add synchronous worker modules, the synchronization overhead and the additional work introduce a strong performance decrease for the real time sender. This supports the measurements made in Section 4. With the new approach (Figure 9), the addition of new modules has no influence at all on the performance of the real time sender[3]. The performance of the time-critical module is thus independent of its position in the hierarchy which was one of our major goals. It is running in its own thread, executing an Estelle cycle only for itself. The Estelle cycle is thus much shorter than for the whole specification module tree, resulting in much more timer checking and allowing a finer granularity for delay value checking.

[3]The influence of the `select()` call could not be eliminated from the measurements. Thus, we still have the typical staircase shape.

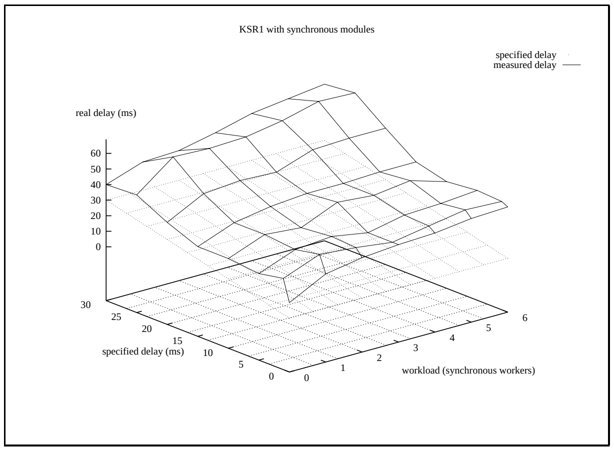

Figure 8: KSR with synchronous modules: real time sender performance

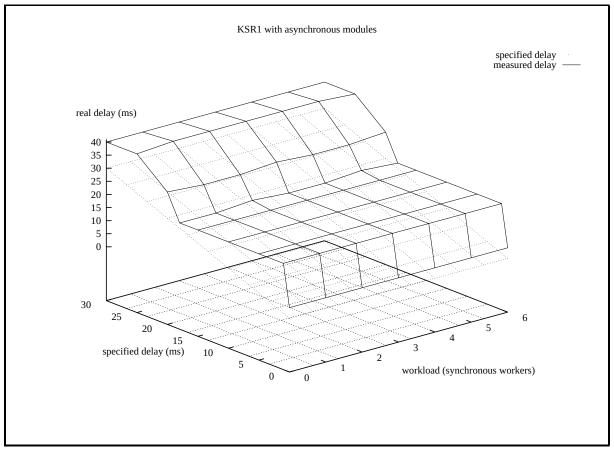

Figure 9: KSR with asynchronous modules: real time sender performance

6 Conclusion and Outlook

In this paper, we showed that it is generally possible to specify some aspects of multimedia systems in Estelle, while others cannot expressed. We derived implementations from a sample specification automatically. Their runtime results did not match the intended behavior of the specification. As reasons for this, we identified the language Estelle itself, namely the delay operator semantics, the synchronous parallelism, and the implementation environment. As a solution to the parallelism problem, we adopted and implemented the extended version of Estelle proposed in [BG94] which allows specifying asynchronous `process` modules and thus increases the degree of parallelism and independence between Estelle modules. Measurements on a parallel machine show the suitability of this approach.

In the current version of the implementation environment for asynchronous modules, we assume the existence of enough system resources to assign a whole processor to a thread running an asynchronous module. However, that cannot be done in the case of too many modules or too few processors. To a certain extent, this problem can be solved by the use of operating system priorities. Threads running modules with real-time requirements should have higher priority than other threads and thus be assigned the processor more often. A better solution, however, is the use of a **real-time operating system**. Using these systems, threads may be assigned processors when they really need it. The problem of when to execute which thread is solved by the scheduling strategy. This adds a further requirement to the mapping of Estelle modules onto the operating system.

To specify all aspects of multimedia systems, the language Estelle does not provide sufficient means. In this area, much work has yet to be done. One way could be to add to transitions some sort of pre- and postconditions expressing timing relations. There exists an approach where it is possible to annotate transitions with minimum and maximum execution times [DB87]. Its goal, howvever, is simulation and performance evaluation. Another possibility would be to add constructs for the relation of module states, i.e. to describe timing constraints on state-switching sequences. Expressing relations between two events in different modules is more difficult: the information-hiding principle of Estelle modules would be violated, resulting in a deep change of the language.

References

[BBBC93] G. Blair, L. Blair, H. Bowman, and A. Chetwynd. Formal Support for the Specification and Construction of Distributed Multimedia Systems (The Tempo Project). Technical Report MPG-93-23, Lancaster University, December 1993.

[BBBC94] H. Bowman, G.S. Blair, L. Blair, and A.G. Chetwynd. Time versus abstraction in formal descriptions. In Tenney et al. [TAU94], pages 467–482.

[BG94] J. Bredereke and R. Gotzhein. Increasing the Concurrency in Estelle. In Tenney et al. [TAU94], pages 127–141.

[BGS87] 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.

[BT82] T. P. Blumer and R. L. Tenney. A Formal Specification and Implementation Method for Protocols. *Computer Networks*, 6:201–217, 1982.

[Bud92] S. Budkowski. Estelle Development Toolset. *Computer Networks and ISDN Systems, Special Issue on FDT Concepts and Tools*, 25(1), 1992.

[DB87] P. Dembinski and S. Budkowski. Simulating Estelle specifications with time parameters. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing, and Verification VII*, pages 265–279, Amsterdam, 1987. Elsevier Science Publishers B.V. (North–Holland), Amsterdam.

[DBLL92] A. Danthine, Y. Baguette, G. Leduc, and L. Leonard. The OSI95 Connection–Mode Transport Service – The Enhanced QoS. In A. Danthine and O. Spaniol, editors, *hpn'92 – 4th IFIP conference on high performance networking, 14.–18.Dec. 1992, Liège*, pages E1–E18. North Holland, 1992.

[FBR93] S. Frank, H. Burkhard, and J. Rothnie. The KSR1: High Performance and Ease of Programming, no longer an Oxymoron. In H.-W. Meuer, editor, *Supercomputer '93*, Informatik aktuell, pages 53–70. Springer Verlag, Heidelberg, 1993.

[Fer90] Domenico Ferrari. Client Requirements for Real-Time Communication. *IEEE Communications Magazine*, 28(11):65–72, November 1990.

[Fer92] Domenico Ferrari. Real–Time Communication in an Internetwork. *Journal of High Speed Networks*, 1(1):79–103, 1992.

[FH94] S. Fischer and B. Hofmann. An Estelle Compiler for Multiprocessor Platforms. In Tenney et al. [TAU94], pages 171–186.

[Got92] Reinhard Gotzhein. Temporal logic and applications – a tutorial. *Computer Networks and ISDN Systems*, 24:203–218, 1992.

[HK94] Thomas Held and Hartmut König. Increasing the Efficiency of Computer-aided Protocol Implementations. In *Proceedings PSTV'94, Vancouver*, 1994.

[HSS90] Dietmar Hehmann, Michael Salmony, and Heinrich J. Stüttgen. Transport services for multi–media applications in broadband networks. *Computer Communications*, 13(4), 1990.

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

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

[Kur93] Jim Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees. *Computer Communication Review*, 23(1), January 1993.

[LL94] L. Léonard and G. Leduc. An enhanced version of timed LOTOS and its application to a case study. In Tenney et al. [TAU94], pages 483–498.

[QF87] J. Quemada and A. Fernandez. Introduction of Quantitative Relative Time into LOTOS. In *Protocol Specification, Testing and Verification VII*, pages 105–121. Elsevier Science Publishers B.V. (North–Holland), Amsterdam, 1987.

[SB90] D. P. Sidhu and T P. Blumer. Semi–automatic Implementation of OSI Protocols. *Computer Networks and ISDN Systems*, 18:221–238, 1990.

[SS93] Rachid Sijelmassi and Brett Strausser. The PET and DINGO tools for deriving distributed implementations from Estelle. *Computer Networks and ISDN Systems*, 25(7):841–851, 1993.

[TAU94] R.L. Tenney, P.D. Amer, and M.Ü. Uyar, editors. *Formal Description Techniques, VI.* Elsevier Science Publishers B.V. (North–Holland), Amsterdam, 1994.

[VLC88] 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.