Specification Style and Efficiency in Estelle

Jan Bredereke

McMaster University, CAS
1280 Main Street West · Hamilton, Ontario L8S 4K1 · Canada
jan.bredereke@topmail.de · http://www.poboxes.com/jan.bredereke

Abstract.
There still seems to be a substantial potential for improvements of current compiler tools for the FDT Estelle with respect to the optimization of the run-time efficiency of the generated implementations; many compiler tools perform no optimizations at all. Until higher optimizing tools are available, a suitable specification style can improve the run-time efficiency, by making less use of expensive language constructs. We propose two approaches for this: *light-weight sub-modules* and *structured integrated protocol entities.* Case studies showed a speedup of about factor four for a structured integrated specification.

1 Introduction

In this paper, we investigate the relationship between the specification style used for specifying systems in the FDT Estelle, and the run-time efficiency of implementations generated automatically from these specifications. Correctness and run-time efficiency are essential properties of software in general and of high-speed protocols in particular. We can boost confidence in the correctness of a protocol implementation by generating code automatically from the specification. However, the run-time efficiency is often insufficient.

The run-time efficiency of FDT-based implementations can be improved at the *design* stage, during the *implementation* process, and at *run-time* [GBE+96, Bre97]. In this paper, we are interested in optimizations during design.

After we have captured the requirements of a new communication service, we must devise a design of the architecture of a protocol that provides this service. In order to realize high-speed communication services, this has led to the development of a new class of protocols called *light-weight protocols* [DDK+90] (for example, the Xpress Transport Protocol XTP [XTP95]), designed in particular to reduce processing time and thus to achieve higher throughput. Since communication systems are designed frequently to interwork with already existing systems, it is not always possible to introduce a new protocol. An alternative approach is to restructure the design specification in order to facilitate a run-time efficient implementation. *Integrated layer processing* [Atk88, Cla85, ClTe90, Svo89, AbPe93] reduces the structure of the specification in order to achieve the goal. Hofmann [Hof93a] applies this approach to Estelle.

Communication systems today are usually designed using a *layered architecture* (see, e.g., ISO-OSI-BRM [ISO81], DECnet [Mal89], IBM's SNA [Mar87], and the Internet [Com88]). Each layer solves a partial problem and hides the details of it from the other layers. In the following, we adhere to the terminology of the ISO-OSI basic reference model [ISO81]. This reference model defines the well-known seven layers. Nevertheless, this is a *conceptual* architecture, the reference model does not require an implementation to have the same architecture.

In Section 2, we investigate the main run-time cost factors for Estelle and fathom the potential for improvements. Currently, existing compilers are rather naive about optimization heuristics, therefore we investigate how a suitable specification style can improve the run-time efficiency, too.

In Sections 3, 4, and 5, we propose to reduce *structure* available in the original specification in order to lower the overhead in the implementation; this may be done either from the beginning or by a transformation. A number of experiments demonstrate that the approach can significantly improve the run-time efficiency. Section 6 summarizes the results and lists potential future work on optimizations by compilers.

2 Run-Time Costs in Estelle

We now investigate the main run-time cost factors of automatically generated implementations. We are interested, in particular, in those run-time costs that are due to the use of Estelle and that we would not have to pay in a manually hand-coded implementation. In a first step, we consider theoretically where the use of Estelle implies additional overhead, and in a second step, we support the results by measurements.

2.1 Qualitative Considerations

The execution model of Estelle distinguishes between two basic phases: the transition *selection* phase and the transition *execution* phase. In the transition execution phase, the result of executing a transition block is determined, which is specified mainly by Pascal-style imperative statements. There is usually a strong structural similarity of a transition block specification and the corresponding hand-implementation in an imperative language, for example Pascal or C. Therefore, we conclude that the run-time costs of the execution phase of an implemented specification are roughly similar to the corresponding run-time costs of a hand-implementation in a Pascal-like language. The only exception for which this conclusion does not necessarily hold are those statements that concern communication[1] since communication is handled differently in Estelle compared to the usual communication mechanisms provided by imperative programming languages.

There are differerent ways to implement the transition selection scheme. Most current Estelle compilers generate code that indeed generally evaluates the firing conditions of *all* transitions in the subsystem,[2] except for those transitions that are prevented from firing by the Estelle parent-child priority principle. In principle, it is possible to determine already at compile-time certain conditions under which it is impossible that a fireable transition exists in a module instance. For example, when a subsystem has completed a transition selection phase without finding any fireable transition and when this subsystem has no transition with a `DELAY` clause then it can be concluded that no transition needs to be fired until a message arrives from another subsystem. Therefore, the implementation of a subsystem could optimize the transition selection phase by just doing nothing until a message arrives from another subsystem. If a manually coded im-

[1]The creation and termination of protocol entities and communication links is special in Estelle, too. We disregard this here since such changes are usually seldom compared to plain communication events.

[2]A few compilers use tables in order to find those transitions faster that have the correct major state and the correct incoming Estelle message.

plementation is designed carefully then it will probably make a similar optimization.[3]

Unfortunately, current Estelle compiler tools are generally rather "naive" about optimizations such as the one described above, and most perform only very few optimizations or even none at all.[4]

There are two solutions to this problem:

- find *better optimization heuristics for Estelle compilers*, or
- employ a specification style that *reduces* the use of *expensive language constructs*.

The first solution is the one that should be chosen in the long run, but it also requires considerable effort. In [Bre97], we present an overview of some first steps into this direction. We still have to see how much can be gained by optimizing compilers. Until there is sufficient economic interest (and accompanying effort) in this approach, we therefore investigate the second approach. Nevertheless, both approaches can be applied together.

2.2 Quantitative Measurements

The Estelle compiler MET[5] [The97] can measure several parameters related to run-time resource consumption, for example, it allows to determine the run-time costs for transition selection and for transition execution. We have conducted measurements for two specifications.

The first specification describes the well-known alternating-bit-protocol (AB protocol). The specification comprises five active module instances in total (two for the protocol machines) which are all separate subsystems, and it has 389 lines of specification text. It creates 32 instances of transitions in total. In [Bre97], we present more details. The ratio of the total transition selection time to the total transition execution time turned out to be 1.9 : 1.

The second specification describes the eXpress Transport Protocol (XTP) [XTP95]. This Estelle specification is one of the largest existing, it has 8904 lines of specification text and creates 17 active module instances with 580 transition instances in total. The top-level module instance is attributed as `SYSTEMPROCESS` such that all its child module instances run synchronously parallel. The ratio of the total transition selection time to the total transition execution time turned out to be 2.2 : 1 for this particular specification and use case. In [Bre97], we present more details.

Communication is a critical aspect for another Estelle compiler, the Pet/Dingo toolset [SiSt93]. Hartmut Penner investigates the performance of automatically generated implementations in his Masters thesis [Pen96], using our AB protocol, too. He finds that the transition selection and execution phase together incur run-time costs of roughly the same order of magnitude as with MET. But he also finds that 99.7 % of the total run-time are spent for communication (including synchronization messages) which is measured separately, here. This effect is due to a very inefficient communication scheme of the generated code, and it leads to an implementation with a throughput of only about 2.4 messages per second[!].

---

[3] We generally assume that the manual implementation roughly reflects the structure of the specification.

[4] We have to stress that finding good optimization heuristics is a difficult and time-consuming task and requires a lot of experience with many different kinds of specifications.

[5] In the meantime, the compiler MET has developed into the compiler XEC [The98, ThGo98a, ThGo97, ThGo98b]. We performed our measurements still with MET.

The Pet/Dingo toolset was the first[6] publicly available Estelle compiler that allowed to generate code for distributed and parallel implementations. The above numbers were measured for a distributed implementation, in contrast to the results for the MET compiler which were measured for a single process implementation. Obviously, the Pet/Dingo toolset was in disadvantage compared to MET due to the additional inter-process communication. The very inefficient Pet/Dingo communication scheme could be improved, nevertheless this comparison points out that run-time costs for communication and synchronization become crucial when we deal with parallel (or even distributed) implementations.

To summarize, the selection of fireable transitions is expensive in implementations generated by current compilers. Furthermore, communication is expensive in parallel (or even distributed) implementations. The latter concerns user-specified communication as well as synchronization messages which result from the Estelle execution model for module instances that are not subsystems.

3 Structure vs. Efficiency

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 run-time efficient code, despite the fact that some optimizations can still be made by the compiler (compare Section 2 and [GBE+96, Bre97, The98, HeKö95]).

We derive two main specification style rules that improve run-time efficiency:

- reduce the transition selection costs by specifying
  - fewer module instances,
  - a less deeply nested module instance hierarchy, and
  - fewer transitions per module instance,

  and
- reduce the number of messages sent.

Both rules work hand in hand since reducing the number of module instance boundaries usually also reduces the number of messages that have to cross these boundaries. Inside a module instance, we can use other and more efficient communication mechanisms; in particular, we can use communication through shared memory which, in Estelle, is not possible for sibling module instances.

There are two basic approaches for reducing the structure of a specification:

- use a specification style *from the beginning* that renders little structure, and
- transform a specification *later* in order to achieve less structure.

[6] There was earlier work covering only a subset of Estelle [JaJe89]. The UHH compiler was published later [KrGo93].

Since specifications should be oriented towards the problem rather than towards an implementation, the latter should be preferred.

Only if the components of the problem are so closely interrelated that it is hard to structure them into different (Estelle) module instances, the first alternative should be considered.

A transformation of a specification into another specification with less structure may be viewed as a first step towards an implementation. More implementation details may be added, such as information about the run-time costs of transition selection and of communication. The difference to the usual implementation step is that the specification is not transformed into a description in a programming language but into a description still in the same (specification) language.

An important point with a transformation is that the semantics of the described system is not changed[7] by the transformation. It is much easier to prove this property of a transformation when it is performed inside a single description language and not between two different languages.

4 Specifying Light-Weight Sub-Modules in Estelle

A central means of structuring in Estelle is the concept of a module. But we can also specify more than one protocol entity by only one Estelle module.[8] This reduces the structure of the specification (in order to enhance the run-time efficiency) and may have other advantages as discussed in the examples below, at the cost of a maybe reduced readability of the specification. This alternative specification style proved useful in several particular situations, and additionally it may be regarded as a base for yet another style, presented later. We discuss the situations which are suitable for applying the style after we have introduced to it.

4.1 The Specification Style

The alternative specification style uses only one Estelle module for two (or more) protocol entities. We illustrate our discussion by a simplified example. In our example, there are two protocol entities called `pres`[9] and `sess`, they communicate through a channel called `sService` among each other, and they communicate through channels called `pService` and `tService` with other protocol entities. Figure 1 outlines the system structure in Estelle.

The language definition of Estelle allows only one set of major states for one module, but a module may contain an arbitrary structured extended state space. Therefore we encode the major state sets of the protocol entities into the extended state space of the module. We specify a variable of a suitably defined enumeration type in order to encode one of the state sets. Often, the specification of a protocol entity comprises an extended state space itself, too. For example, this can be a sequence number counter or a message buffer. When we specify more than one protocol entity into one Estelle

[7] A change of semantics may be defined either by some equivalence relation or by an implementation relation.

[8] In the formal description technique SDL [ITU94], there is the related concept of *services* which introduce a substructure to a process; and in the Statechart formalism [Har88], there are AND states.

[9] Since we extend our ideas in Section 5 in order to comprise the idea of layer integration, the names in the example are already taken from the presentation layer and the session layer which are discussed there.

```
CHANNEL pService(user,prov);
  BY user: pConReq,pDatReq(dt:CHAR); BY prov: pConInd,pDatInd(dt:CHAR);
CHANNEL sService(user,prov);
  BY user: sConReq,sDatReq(dt:CHAR); BY prov: sConInd,sDatInd(dt:CHAR);
CHANNEL tService(user,prov);
  BY user: tConReq,tDatReq(dt:CHAR); BY prov: tConInd,tDatInd(dt:CHAR);

MODULE pres_sess_header SYSTEMPROCESS;
  IP pres_up:   pService(prov) INDIVIDUAL QUEUE;
  IP sess_down: tService(user) INDIVIDUAL QUEUE;
END;
BODY pres_sess_body FOR pres_sess_header;
  IP pres_down: sService(user) INDIVIDUAL QUEUE;
  IP sess_up:   sService(prov) INDIVIDUAL QUEUE;
  ....
  INITIALIZE BEGIN CONNECT pres.down TO sess.up;  .... END;
END;
```

Figure 1: Module and communication structure for the light-weight sub-modules example.

```
TYPE pres_TYPE =
  RECORD
    majorState: (idle,con,w4ack);
    buf: CHAR
  END;
VAR pres: pres_TYPE;
....
INITIALIZE
  BEGIN WITH pres DO majorState:=idle END;
TRANS WHEN pres_down.sConInd PROVIDED pres.majorState=idle
  BEGIN WITH pres DO
    BEGIN majorState:=con; OUTPUT pres_up.pConInd END;
  END;
TRANS WHEN pres_up.pDatReq(dt) PROVIDED pres.majorState=con
  BEGIN WITH pres DO
    BEGIN majorState:=w4ack; OUTPUT pres_down.sDatReq(dt) END;
  END;
```

Figure 2: Keeping the state space of `pres` separate by the `RECORD` and the `WITH-DO` constructs.

module, it is important that the state space of these protocol entities be kept separate. We therefore use the data structuring concept of a `RECORD` in order to keep the state space components of one protocol entity together. In principle, we can access each component by prefixing it with the name of the protocol entity (variable) it belongs to. In order to avoid repeating this prefix all over, we instead frame the entire block of each transition by a Pascal-style `WITH-DO` statement that allows us to access the components of the extended state space exactly in the same way as if the protocol entities would have been specified into separate Estelle modules. Furthermore, this also marks each of the transitions as belonging to one of the protocol entities. We describe the major state of a protocol entity by a record component that is called `majorState` by convention. Figure 2 demonstrates how we keep the state space of protocol entity `pres` separate by the `RECORD` and the `WITH-DO` constructs.

The above discussions on the extended state space refer to Pascal-style variables only. Module variables and internal interaction points present some difficulties, since they cannot be enclosed into a record. When a protocol entity employs a module variable since it creates a child module, it has so much structure that we don't recommend to use a specification style for *light*-weight sub-modules. If this should be done nevertheless, one can use a naming convention to separate module variables and internal interaction points. A special case is an internal interaction point that is the result of specifying two light-weight sub-modules into one Estelle module. This internal interaction point is associated to more than one sub-module and should therefore not be enclosed into the data structure of only one sub-module anyway.

Specifying the state transitions of the protocol entities is straightforward. Since an Estelle module may have only one set of state transitions, we simply concatenate the state transitions of the protocol entities, and we require that they are separated by appropriate comment lines.

4.2 Discussion

The protocol entities may communicate by the usual Estelle means, i.e., (internal[10]) interaction points and channels. But the alternative specification style opens new possibilities for communication, too: a transition of such a module may have firing conditions that depend on the state of more than one protocol entity, and its firing effects may modify the state of more than one protocol entity. Effectively, this is a way to specify *synchronous communication* in Estelle! Otherwise, Estelle allows communication over asynchronous[11] FIFO queues, only.

And the discussed specification style allows another new kind of communication mechanism between sibling protocol entities. Up to now, *communication through shared memory* was possible only between parent and child modules. Now, a protocol entity may access (and modify) the extended state space of a sibling protocol entity by just stating the fully qualified variable name of the state component, for example: `pres.buf:='J'`.

In summary, this alternative specification style allows for much more flexibility. This flexibility is achieved *without any extension of the language Estelle* (as, for example, with the synchronous communication in Estelle* [Cou88, Cou87, CDA89, CoSa92]).

Of course, specifying more than one protocol entity in one Estelle module bears the danger that conceptually separated things are intermingled and that the logical structure of the specification is blurred. Therefore it is crucial that the above stylistic rules which separate concerns of different layers are strictly enforced. Generally, this specification style should be applied only

- when the additional expressive means (such as synchronous communication and shared memory) are needed, or
- when the reduction of structure in order to enhance the run-time efficiency of an implementation is required.

In fact, we describe a more structured way to achieve light-weight sub-modules in Section 5. Only when the transformation described there cannot be applied but nevertheless more run-time efficiency is needed, this specification style should be applied.

---

[10]Estelle allows to specify interaction points that are internal to a module.

[11]One may argue that the semantics of a message delivery into the destination queue of an Estelle channel describes synchronous communication. Nevertheless, the processing of enqueued messages must happen only later. Therefore, we consider this communication as asynchronous.

4.3 Examples

We now present two examples where such a specification style proved useful. The first example[12] is the ISO protocol for distributed transaction processing (OSI-TP) [ISO92] which we specified (partially) in Estelle [Bre93]. OSI-TP is a complex application layer protocol which is described in the international standard [ISO92] by state transition tables and additional English text. One of the components of OSI-TP is the *multiple association control function (MACF)* which coordinates the communication on the different *associations* of the TP protocol service provider. The international standard specifies one "dialogue" instance per association in the MACF. Each dialogue instance has its own set of states and extended variables, and there is also a set of extended variables that is common to all dialogue instances. There are state transitions that affect only one dialogue instance, but there are also special state transitions that depend on conditions from all dialogue instances and that affect the states of all dialogue instances. The (American) National Institute for Standards and Technology (NIST) has produced an Estelle description[13] of OSI-TP including the MACF component, and this Estelle description indeed models each of the dialogue instances by a separate Estelle module instance. This attempt demonstrated that synchronous communication, such as by the common variables of the MACF dialogue instances, can be approximated in Estelle only awkwardly.[14] Also, the shared variables of the MACF component forced the authors to move the transitions that used these variables to a conceptually wrong place, i.e., to the parent module of the association handling module. Our specification style, on the contrary, allowed to express the OSI-TP structure naturally [Bre93].

The second example is an Estelle specification of a telephone switching system which is discussed in [Bre97]. The system contains an *Originating/Terminating Basic Call Process (OBCPandTBCP)*. Its duty is to handle call setup and call termination messages, and it also forwards voice messages when a call is in progress. We modeled the OBCPandTBCP component by one Estelle module. Its major state is determined by the different call processing states, and most transitions concern call processing. But there are also a few transitions that do nothing but forwarding incoming voice messages, depending on an extended state variable that indicates whether a voice connection exists. These transitions do not depend on the major state of the module. Conceptually, the voice message forwarding is rather a distinct component (which communicates with the main call process by a shared variable). We therefore decided to group the transitions of the voice message forwarding component together and separated them by appropriate comment lines. Similarly, the state space was separated. Furthermore, we identified another sub-automaton which handles call setup requests while another call is already in progress. These transitions depend on the main automaton only insofar as the main automaton must not be in its initial major state. In this case, a second call setup request is rejected. The two automata are even more closely coupled than in the first case. These two examples from the telephone switching system show how one may sometimes find light-weight sub-automata in a specification by just grouping some transitions together. In this case, the grouping improved the readability of the specification.[15]

---

[12] In fact, we already applied the additional ideas of Section 5 on the structured integration of protocol entities in this example.

[13] This specification was intended as an appendix to the OSI-TP standard [ISO92].

[14] The conditions for the common transitions were flagged by special exported variables to the parent module which then notified its children (asynchronously) by message passing of the event.

[15] We abstained from specifying a separate Estelle module after we identified the structure. The

5 Structured Integration of Protocol Entities in Estelle

When we need to reduce the structure of an Estelle specification for run-time efficiency reasons, it may be possible to transform the specification into another specification with less structure. The applicability of this approach depends on a suitable structure of the specification.

A common example for a situation with many module instances is a layered protocol architecture, where each layer is expressed by at least one Estelle module instance. Hofmann [Hof93b] proposes to integrate two (or more) such protocol entities by *layer integration*, thus eliminating the communication interface between two adjacent protocol entities. The two module instances that describe the protocol entities are merged into a single one which exhibits the same behaviour. Each transition that outputs a message to a common interaction point is merged with the corresponding receiving transition. This cuts the costs both for passing a message to the second module instance and for determining which transition can process it. In a case study, Hofmann integrates specifications of the ISO presentation and session layers, 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 of Hofmann's approach is that the transformation is not reversible. Once the two layers are integrated, the original specification cannot be recovered. For example, the intermediate interface between the two layers is lost which is important for testing and validation. Therefore, it is mandatory to keep the specifications of the two separate layers as the original source. A special tool that performs the integration automatically must be available in order to be able to redo the layer integration quickly after a possible change of the specification.

These drawbacks lead us to combine the idea of layer integration with our idea of light-weight sub-modules from Section 4. Layer integration is indeed a special case of the integration of arbitrary protocol entities. The result of the combination is the idea of the *structured integration of protocol entities*. The structured integration of protocol entities is an integration of two (or more) separate protocol entities into a single Estelle module, where the individual protocol entities are still syntactically separated such that the integration transformation can be reverted, and such that the intermediate interface can still be identified.

Semantically, we integrate two automata into a single one in the same way as Hofmann; but we perform the merging step of the automata's transitions using a different syntax. This improved syntax, which renders the desired structuring, basically distinguishes both approaches. The blocks of the transitions are not just concatenated, but each block is transformed into a (Pascal-style) procedure first, and only two procedure calls are concatenated. The state space part of each protocol entity is now kept separately, as with the specification style for light-weight sub-modules. Similarly, the procedure definitions of each protocol entity are kept together. In Section 5.1, we describe the specification style for the transformation result in more detail. The structure of the resulting specification looks similar to the structure of a specification in the style for light-weight sub-modules. The advantages of the structured integration of protocol entities are that

- we gain the run-time improvements of layer integration, and

additional module would have become too small to justify the overhead it would have implied for the intended automatic validation for feature interactions.

- the integration result remains readable and maintainable.

The approach can be used in either of two ways:

1. The integration transformation is performed automatically by a tool.
2. The integration is performed manually and the integrated specification becomes the main source document.

The second approach is an extension of the approach of manually specified light-weight sub-modules from Section 4. Compared to light-weight sub-modules, this approach further improves the run-time efficiency due to a further reduction of the run-time cost factors identified in the end of Section 2: selection of transitions and communication. The (internal) interaction points, which are in general still used for the communication between light-weight sub-modules, are replaced by less expensive procedure calls; and the number of transitions is reduced due to their (structured) merging, thus exchanging transition selection costs for much lower procedure call costs.

5.1 Syntactic Transformation

The transformation is applicable when the specification has a *suitable structure.* The described system must contain two module instances that are connected by one channel; these two module instances will be integrated.[16] Figure 3 introduces to the structure of a running example which we develop step by step further in the following discussion. The modeled protocol entities are the still same as with the example in Section 4.

Each of the modules has

- a set of transitions[17],
- a set of major states, and
- a set of extended state variables.

Each of these sets may be empty. Furthermore, there are some external interaction points for each module, of which one is connected to the channel linked to the other module. Figure 4 illustrates the situation. Figure 6 presents the Estelle code for module `pres` from our example. We assume that the other parts of the extended state space are empty, i.e., there are *no internal interaction points* and *no child modules.*

The result of the transformation is a single module which has

- two sets of procedures,
- two record variables, and

[16] In general, Estelle allows to specify the dynamic creation and termination of module instances as well as of connections between them. The transformation requires that either both module instances and their connection are created at system initialization time and are *static* afterwards, or that at least the time of creation and termination is the same for both module instances and their connection. (The latter is not discussed explicitly by Hofmann.) In the following, we assume that there is exactly one static module instance for each module definition, and we do not further distinguish between these terms for simplicity.

[17] We assume that nested transitions have been expanded already, as described in the Estelle standard [ISO97].

```
CHANNEL pService(user,prov);
  BY user: pConReq,pDatReq(dt:CHAR),pDisReq;
  BY prov: pConInd,pDatInd(dt:CHAR),pDisInd;
CHANNEL sService(user,prov);
  BY user: sConReq,sDatReq(dt:CHAR),sDisReq;
  BY prov: sConInd,sDatInd(dt:CHAR),sDisInd;
CHANNEL tService(user,prov);
  BY user: tConReq,tDatReq(dt:CHAR),tDisReq;
  BY prov: tConInd,tDatInd(dt:CHAR),tDisInd;

MODULE pres_header SYSTEMPROCESS;
  IP up:   pService(prov) INDIVIDUAL QUEUE;
  IP down: sService(user) INDIVIDUAL QUEUE;
END;
MODULE sess_header SYSTEMPROCESS;
  IP up:   sService(prov) INDIVIDUAL QUEUE;
  IP down: tService(user) INDIVIDUAL QUEUE;
END;
....
MODVAR pres: pres_header; sess: sess_header;
INITIALIZE BEGIN INIT pres WITH pres_body;      INIT sess WITH sess_body;
                 CONNECT pres.down TO sess.up;  .... END;
```

Figure 3: Module and communication structure of two protocol entities to integrate.

- a set of transitions.

For each of the two modules,

- the transitions are transformed into procedures,
- the set of major states is transformed into a variable of enumeration type, and
- all extended state variables and the variable describing the major state are packaged into a record variable.

Figure 5 shows the situation after the transformation. Figure 7 presents the Estelle code resulting from module `pres` in Figure 6. At the top of Figure 7, we see the definition of the state space consisting of the major state space and one variable in the extended state space. The result of the transformation complies entirely to the style rules for light-weight sub-modules as discussed in Section 4. It is straightforward to perform the transformation, therefore we do not describe it in more detail.

The details of the transformation of the transitions to procedure definitions are described in [Bre97]. Basically, there is one procedure definition for all transitions with the same `WHEN` clause. The procedure name is derived from the name of the interaction point and the name of the message. Its parameters are those of the message, if existing. When there is only one transition with this `WHEN` clause, the procedure block is identical to the transition block[18] with three exceptions: (1) we frame it by a `WITH-DO` statement exactly as with the specification style for light-weight sub-modules in Section 4, (2) the transition block is prepended with an assignment to the variable `majorState` in order to reflect the `TO` clause, and (3) the `OUTPUT` statements that refer to the internal

[18] In order to convert a transition block into a procedure block, we have to relax a restriction imposed by the definition of the language Estelle, as discussed in Section 5.4 below.

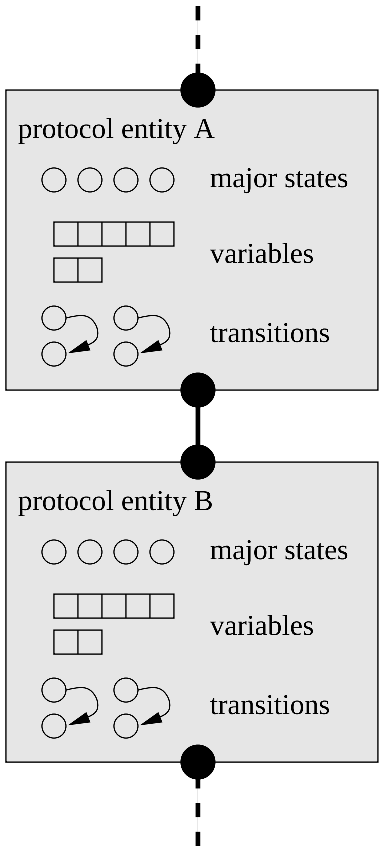

Figure 4: The modules before the transformation.

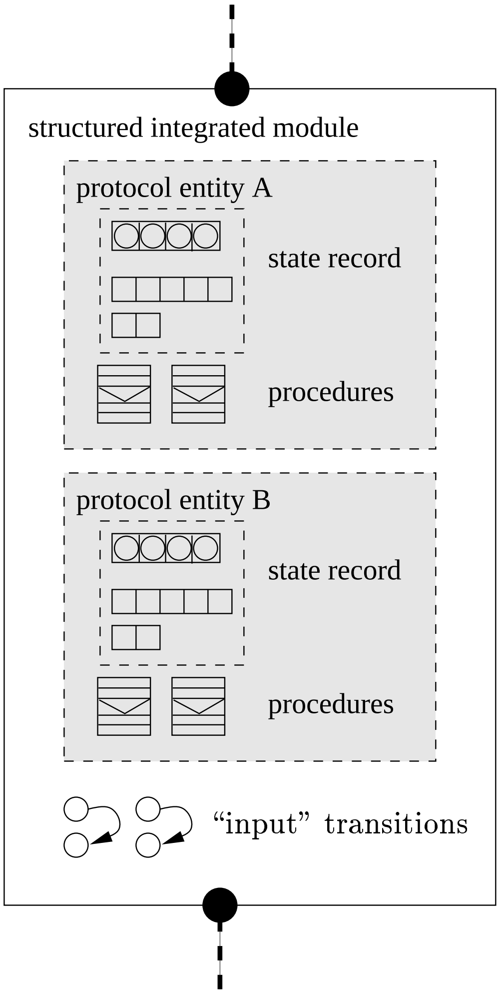

Figure 5: The resulting module after the transformation.

interaction point of the two modules to merge are replaced by a procedure call. Figures 6 and 7 present an example. When there exists more than one transition with this `WHEN` clause then we merge them into only one procedure. When the transitions have different `FROM` clauses, we insert an appropriate `CASE` construct directly inside the `WITH-DO` statement described above. When there are different `PROVIDED` clauses, we enclose the generated text for each transition block by appropriate `IF/THEN/ELSE` constructs.[19] Please compare also Figure 7.

Up to now, we have described how the data part is transformed, and we have described how transitions are turned into procedure definitions. One part of the integrated module is still missing, its *transition part*. This part depends on the messages that can be received by the integrated module. Figure 8 presents the transition definitions (and channel definitions) of the integrated module of our example. The purpose of the transitions of the integrated module is receiving messages from the external interaction points and calling the appropriate procedure. We define one transition for each message from each external interaction point. The transition block of the new transition holds noth-

[19]In case that the transitions exhibit non-determinism, we construct a corresponding non-deterministic statement by a "`FORONE x:....SUCHTHAT TRUE DO CASE x OF....`" construct. Please note that we do not recommend the use of our approach when there are more than a few cases of non-determinism.

```
BODY pres_body FOR pres_header;
  VAR buf: CHAR;

  STATE idle,con,w4ack;

  INITIALIZE TO idle
    BEGIN END;
  TRANS WHEN down.sConInd
    FROM idle       TO con   BEGIN OUTPUT up.pConInd END;
    FROM con,w4ack  TO idle  BEGIN OUTPUT down.sDisReq END;
  TRANS WHEN up.pDatReq(dt)
    FROM con        TO w4ack BEGIN buf:=dt; OUTPUT down.sDatReq(dt) END;
    FROM idle,w4ack TO idle  BEGIN OUTPUT up.pDisInd END;
  ....
END;
```

Figure 6: The body of module `pres` in our example.

```
TYPE pres_TYPE =
  RECORD
    majorState: (idle,con,w4ack);
    buf: CHAR
  END;
VAR pres: pres_TYPE;

PROCEDURE pres_INITIALIZE;
  BEGIN WITH pres DO majorState:=idle END;
PROCEDURE down_sConInd;
  BEGIN WITH pres DO CASE majorState OF
    idle:       BEGIN majorState:=con;   OUTPUT up.pConInd END;
    con,w4ack:  BEGIN majorState:=idle;  down_sDisReq END;
  END; END;
PROCDURE up_pDatReq(dt:CHAR);
  BEGIN WITH pres DO CASE majorState OF
    con:        BEGIN majorState:=w4ack; down_sDatReq(dt) END;
    idle,w4ack: BEGIN majorState:=idle;  OUTPUT up.pDisInd END;
  END; END;
....
```

Figure 7: The `pres` protocol entity turned into a procedural style.

ing but a call to the procedure that is the transformation result of the corresponding set of transitions. The only transition clause is the `WHEN` clause.

For a detailed description of the transformation the reader is referred to [Bre97]. It is straightforward to define a *reverse transformation* that recovers the transitions back from the procedures, and the major state sets from the enumeration types. Only a few details are lost, such as the names of the transitions, and whether there were two transitions with alternatively enabled `PROVIDED` clauses or an equivalent single transition with an `IF/THEN/ELSE` construct in the transition block. This information could be kept in comments, which could be generated and evaluated automatically.

We need to make an important restriction in our approach: *for each incoming message, there must always be at least one fireable transition to accept it*, regardless of

```
CHANNEL pService(user,prov);
  BY user: pConReq,pDatReq(dt:CHAR); BY prov: pConInd,pDatInd(dt:CHAR);
CHANNEL tService(user,prov);
  BY user: tConReq,tDatReq(dt:CHAR); BY prov: tConInd,tDatInd(dt:CHAR);

MODULE pres_sess_header SYSTEMPROCESS;
  IP up:   pService(prov) INDIVIDUAL QUEUE;
  IP down: tService(user) INDIVIDUAL QUEUE;
END;

BODY pres_sess_body FOR pres_sess_header;
  ....
  .... { pres protocol entity in procedural style } ....
  .... { sess protocol entity in procedural style } ....
  INITIALIZE BEGIN pres_INITIALIZE; sess_INITIALIZE END;
  TRANS WHEN down.sConInd   BEGIN down_sConInd END;
  TRANS WHEN up.pDatReq(dt) BEGIN up_pDatReq(dt) END;
  TRANS ....
END; { pres_sess_body }
```

Figure 8: The integrated module in our example.

the major or extended state.[20] With this restriction, it is always admissible to call a procedure in order to process an incoming message.

Hofmann's layer integration algorithm already requires some prerequisites itself that ensure the correctness of the integration. For example, each transition may perform at most one output operation so that there is at most one transition to concatenate. We demand additionally that there are *no infinite cycles* in which the two protocol entities alternately send each other a message.[21] These would be transformed into an infinite procedure call sequence. A interesting topic for further research is how to relax the requirement that there be at most one output operation; the minimum requirement for our transformation is that there are no infinite cycles.

A similarly interesting extension of the basic formalism would be to include spontaneous transitions and `DELAY` transitions which could be incorporated into our transformation straightforwardly. Unfortunately, the semantical foundation of this extension would necessitate a change in Hofmann's formalism.

Another requirement for the transformation is that module attributes must be such that both modules run *asynchronously* in *parallel*.[22] Furthermore, we require that there are *no transition priorities* defined in any of the two modules, and that their interaction points all have the `INDIVIDUAL QUEUE` message delivery discipline.

5.2 Specification Style Issues

As with light-weight sub-modules, it is crucial for the structured integration of protocol entities that a suitable specification style ensures the readability of the specification.

[20] We may have combinations of a message and and a state where no transition is fireable to accept the message, but it must be proven that these combinations are unreachable.

[21] Hofmann does not discuss this case, nevertheless, his algorithm silently breaks these cycles, ensuring the termination of the algorithm.

[22] They run asynchronously in parallel if they are attributed as `SYSTEMPROCESS`, or if we employ the Estelle extension that we propose in [Bre97, BrGo94, BrGo93].

The entire results from the transformation of one protocol entity have to be kept together as one part, and these parts have to be separated by appropriate comment lines.

The linear textual structure of the specification document enforces a sequential order of the above parts. Nevertheless, the internal communication among these parts by procedure calls may call any procedure from any part in the general case. Thus there are likely to be cases when a procedure is called at some point in the specification text where its definition point is still to come later. This is a syntactic error in Estelle, but it can be avoided easily by *forward declarations of the procedures.*

Pascal (and Estelle) allow to declare the calling interface of a procedure without already defining the procedure block. We employ this language construct and let the transformation generate additionally a forward declaration for each of the generated procedures. The forward declarations are arranged textually before any of the actual procedure definitions appears.

These forward declarations of procedures have a positive impact on the readability of the specification: the set of forward declarations of one protocol entity describes its input interface, since there is exactly one declaration for every message that this protocol entity can receive. Furthermore, the name of the message and the name of the interaction point can be seen from the name of the procedure. We require that the forward declarations of one protocol entity are grouped together, too, and that they are separated by appropriate comment lines, again.

The actual procedure definitions for each protocol entity follow later, each prepended by the corresponding type and variable definitions.[23] Figure 9 presents an example.

5.3 Correctness of the Transformation

The correctness of our transformation into a structured integrated module is based on Hofmann's proof of correctness for his transformation into an (unstructured) integrated automaton. This is the reason why we abstained from extending the transformation function by, e.g., allowing spontaneous transitions in the original modules; fundamental prerequisites for the proof of correctness would have been violated.

Hofmann's transformation consists of three steps: construct the product automaton, integrate the product automaton, and remove unreachable transitions. In our approach, the semantics of a transformation of two separate modules into one light-weight submodule (without removing the intermediate channel) is just constructing such a product automaton.[24]

Our fully structured integration transformation also performs the integration step. The syntactic transformation results in a structured integrated module. In order to determine the semantics of this module, we first have to expand any procedure call by replacing it by the procedure block (and perform an appropriate parameter substitution); by this we obtain an unstructured integrated module. The different `CASE` and `IF/THEN/ELSE` paths of an Estelle transition may contain different `OUTPUT` statements. The semantics of such an Estelle transition, in terms of Hofmann's mathematical model, is a set of transitions with the same input message but different output messages, plus (possibly) different starting states and end states. The integration step of Hofmann's algorithm leads to the same set of transitions since both approaches pair two transi-

[23] There is one minor, technical restriction on the order of forward declarations and variable definitions in the language definition of Estelle, we discuss it in Section 5.4.

[24] In Hofmann's formalism, the internal channel is not part of the product automaton, but it still exists externally and is removed only later by the integration step.

```
BODY pres_sess_body FOR pres_sess_header;
  { ******* calling interface for "pres" ******* }
  PROCEDURE pres_INITIALIZE; FORWARD;
  PROCEDURE down_sConInd; FORWARD;
  PROCDURE up_pDatReq(dt:CHAR); FORWARD;
  { ******* calling interface for "sess" ******* }
  ....
  { ******* type and var definitions for "pres" ******* }
  ....
  { ******* procedure definitions for "pres" ******* }
  PROCEDURE pres_INITIALIZE;
    BEGIN WITH pres DO majorState:=idle END;
  PROCEDURE down_sConInd;
    BEGIN WITH pres DO
      BEGIN majorState:=con; OUTPUT up.pConInd END;
    END;
  PROCDURE up_pDatReq; { formal parameters are: up_pDatReq(dt:CHAR) }
    BEGIN WITH pres DO
      BEGIN majorState:=w4ack; down_sDatReq(dt) END;
    END;
  { ******* type and var definitions for "sess" ******* }
  ....
  { ******* procedure definitions for "sess" ******* }
  ....
  { ******* input transitions ******* }
  ....
END; { pres_sess_body }
```

Figure 9: “Forward” declaration of procedures; and grouping of parts.

tions to integrate in the same way; they define pairs by the message on the intermediate channel.

Removing unreachable transitions is achieved in our approach by deleting the original modules altogether after the transformation (and by deleting all procedure definitions after all procedure expansions have taken place).

Please note that we demand in Section 5.1 that there are no infinite cycles in which the two protocol entities alternately send each other a message. This ensures the termination of the procedure expansion step.

Both Hofmann's and our transformation may result in some minor changes in the behaviour.

Firstly, a message that passes through both layers is processed atomically after the integration step. Without the integration, two separate processing steps are required. As long as no other events happen between the input event and the output event, the reduced delay is merely a technical difference which manifests itself in an implementation only by a (desired) increase in run-time efficiency.

Secondly (disregarding the extended state space for the moment), the system of two separate modules and the intermediate channel may store an arbitrary amount of information in the intermediate channel, while the integrated module has only a finite state space. This is because the two *original modules communicate asynchronously* while the *procedure calls resulting from the transformation describe synchronous communication.*

The sequences of events that are possible after the integration are also possible before the integration (in our example, when an implementation is sufficiently fast), but the integration step may remove some possible sequences of events. Because of this, the transformation is in general truly a step towards implementation, and *the appropriate correctness relation is an "implements" relation rather than an equivalence relation.*

5.4 A Required Minor Extension of Estelle

A constraint of Estelle is that ***OUTPUT** statements* may appear in transition blocks but *cannot be used in procedures.* Nevertheless, our transformation function turns a transition block into a procedure while keeping some of the `OUTPUT` statements. Therefore, we propose to remove the constraint that `OUTPUT` statements may not be used in procedures. In [Bre97], we present the details of how the text of the Estelle standard [ISO97] must be changed in order to accommodate for this. Basically, we just remove a few words.

The second constraint of Estelle which we propose to relax is the following: the declarations in a module body may comprise different parts, e.g., for types, variables, and procedures/functions. In contrast to Pascal, Estelle already allows more than one such part of each kind in the declarations. But Estelle still inherits from Pascal the requirement that the corresponding *procedure/function definition must follow the forward declaration until the end of the same procedure/function declaration part.* As a consequence, there may be no type and variable definitions between the forward declaration and the definition of a procedure. We propose to relax this restriction such that the procedure definition must follow only until the end of the entire declarations of the body definition. In [Bre97], we present the details.

Technically, both changes are a minor language extension of Estelle, but they are compatible in the sense that every specification that is syntactically correct according to the Estelle standard [ISO97] remains syntactically correct and keeps its semantics.

5.5 Case Studies and Measurements

The student assignment (Projektarbeit) of Dahl [Dah95] uses the specifications from Hofmann's case study: an Estelle specification of the presentation layer [ISO88c, ISO88d] and of the session layer [ISO87a, ISO87b] of the OSI basic reference model [ISO81]. The student assignment (Projektarbeit) of Barthel [Bar95] reuses the specification of Dahl and adds an Estelle specification of the association control service element (ACSE) [ISO88a, ISO88b].

5.5.1 Run-Time Efficiency

Dahl compares the Estelle specification of Hofmann's (unstructured) integrated module with the specification of our structured integrated module. It turns out that the automatically generated implementations of both speficiations run at about the same speed (see Figure 10); *structuring the integrated module does not incur a noticeable additional run-time overhead.*

All measurements of Dahl and Barthel are performed with implementations generated by the Pet/Dingo toolset [SiSt93]. The implementation consists of a single Unix process comprising the entire specification, allowing to measure the consumed process time, thus eliminating the influence of other load on the machine. The hardware platform is a SUN SPARCstation 10 running SunOS 4.1.3.

Dahl compares the specification comprising three separate modules to the structured integrated specification comprising only one Estelle module for the protocol entities. He

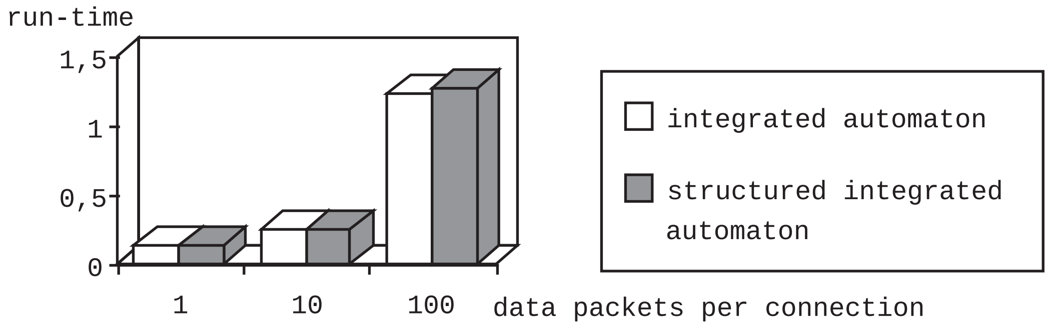

Figure 10: No additional run-time costs by structuring the integrated module.

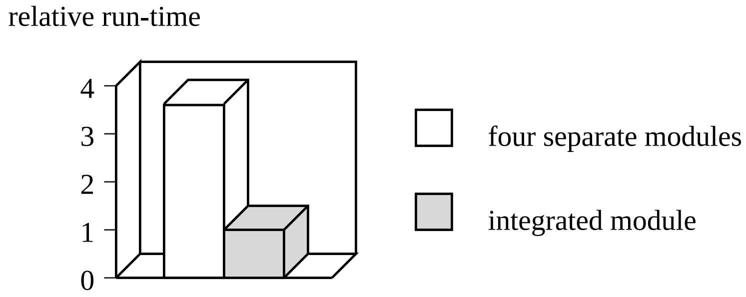

Figure 11: Relative speedup of about factor four for the structured integrated specification.

measures a relative speedup between 3.7 (when 1 data packet is sent) to 4.0 (when 100 data packets are sent). Barthel extends this measurement by including a fourth layer. He measures a relative speedup of 3.6 (no data packets sent, only association setup and termination). Therefore, we have *a speedup of about factor four for the structured integrated specification* (compare Figure 11). It differs slightly from Hofmann's speedup of up to factor three, which is probably due to different underlying operating systems and hardware.

Barthel investigates in more detail the additional run-time overhead for structuring a module into procedures. He transforms the ACSE module into a set of procedures *without* merging it with another module. Therefore, he gets a single set of procedures which is called by a set of transitions in the same way as with a structured integrated module. He compares this specification to the original ACSE specification by measuring the run-time share of only the ACSE module, and he finds that *structuring into procedures costs about 8 % additional run-time for ACSE*. Thus we can conclude (as Dahl's results already indicated) that the additional run-time overhead by structuring is negligible compared to the efficiency gains by reducing the costs for transition selection and communication.

5.5.2 Readability and Maintainability of the Specification

Dahl finds that specifying a system from scratch requires about the same time for specifications with one Estelle module per protocol entity as for specifications with light-weight sub-modules in the form of a structured integrated module. Furthermore in his experience, the manual *transformation from the conventional style into a structured integrated form is straightforward* and does not require much time.

Finally, he finds that *the transformation reduces the readability only marginally.* Both protocol entities remain clearly separated. He is able to modify one of the layers locally (for maintenance purposes) without difficulty. This is not the case with the integrated layers of Hofmann, especially after two such integration steps. Modifications can only be applied sensibly to the original modules, necessitating a new transformation.

Both Dahl and Barthel find that the readability of their specifications, even written in one of the conventional styles, can be enhanced by using our *proposed Estelle extension allowing `OUTPUT` statements in procedures.*

6 Summary and Discussion

There still seems to be a substantial potential for improvements of current compiler tools for Estelle with respect to the optimization of the run-time efficiency of the generated implementations; many compiler tools perform no optimizations at all. Until higher optimizing tools are available, a suitable specification style can improve the run-time efficiency, by making less use of expensive language constructs. We found two areas in which automatically generated implementations usually consume more run-time than hand-coded implementations: currently, the determination of the next action to take, i.e., the *selection of Estelle transitions* for firing, is expensive, as well as the *communication* between Estelle modules.

Both costs are reduced when *structure* available in the specification is reduced. We proposed two approaches for this: *light-weight sub-modules* and *structured integrated protocol entities.* The latter are an extension to Hoffman's unstructured layer integration approach [Hof93a].

In the general literature on protocol implementation [Svo89, Atk88, Cla85], two main implementation models are discussed: the *server model* and the *activity-thread model.* In the server model, every layer of a communication system is implemented by a separate operating system process. Inter-layer communication is implemented by inter-process communication. In the activity-thread model, a single process (or *thread*) performs all processing for one incoming event through all layers involved. There is one thread per incoming event, communication between layers is implemented by procedure calls. In general, the activity-thread model enables more efficient implementations than the server model.

For an Estelle specification, a compiler might produce code that determines the next transition to fire by first checking the module where the last message was sent to, and there first checking those transitions that have a `WHEN` clause corresponding to the message. Under certain circumstances, the compiler can determine the receiving transition as the next transition to fire even *statically* at compile-time, thus eliminating these run-time transition selection costs entirely! When there are long activity-threads, this heuristics drastically reduces the transition selection time [Bre97]. Henke et. al. [HKMT98, Hen97] investigate in more detail how a compiler can exploit this, both for SDL and Estelle; but there is still more research necessary before practical tools are available. In particular, Henke et. al. require a static module structure for Estelle specifications, where only the leaf modules may be active, and where only the "(system)process" module attribute is allowed. Unfortunately, the described heuristics cannot be applied successfully to most specifications in standard Estelle due to the synchronous parallelism of modules attributed as `PROCESS`, and due to the parent-child priority principle. Therefore, our "asynchronous" extension [Bre97, BrGo94, BrGo93] is necessary for specifing an arbitrary structured system with no such synchronization constraints, thus allowing to apply the described implementation heuristics on a large scale. Very recently, Thees et. al. improved the XEC tool [The98, ThGo98a, ThGo97, ThGo98b] to perform the optimizations discussed in the beginning of this paragraph, without the restrictions to Estelle that Henke et. al. make.[25]

Interestingly, this optimization heuristics for compilers is closely related to the structured integration of protocol entities proposed in this paper. Both try to find the next fireable transition in a very similar way. When there are additional compiler heuristics that optimize passing the parameters of the message in a similarly efficient way as a procedure call, the optimized generated code will probably look quite similar to code generated from a specification where our structured integration of protocol entities has been applied in the design phase.

References

[AbPe93] Abbott, M. B. and Peterson, L. L. *Increasing network throughput by integrating protocol layers*. IEEE/ACM Trans. on Networking **1**(5), 600–610 (Oct. 1993).

[Atk88] Atkins, M. S. *Experiments in SR with different upcall program structures*. ACM Trans. Comput. Syst. **6**(4), 365–392 (Nov. 1988).

[Bar95] Barthel, D. *Untersuchung und Beschreibung unterschiedlicher Spezifikationsstile in der formalen Beschreibungstechnik Estelle anhand des "Association Control Service Elements"*. Projektarbeit, Univ. of Kaiserslautern, Dept. of Comp. Sci. (Apr. 1995).

[Bre93] Bredereke, J. *Entwurf eines parallelisierten OSI-TP in Estelle*. Tech. Rep. FBI-HH-M-234/93, Univ. of Hamburg, Dept. of Comp. Sci. (Dec. 1993).

[Bre97] Bredereke, J. *Communication Systems Design With Estelle – On Style, Efficiency, and Analysis*. PhD thesis, Univ. of Kaiserslautern, Shaker Verlag, Aachen, Germany (Aug. 1997). ISBN 3-8265-2764-X.

[BrGo93] Bredereke, J. and Gotzhein, R. *Eine Estelle-Erweiterung zur Steigerung der Nebenläufigkeit*. Tech. Rep. FBI-HH-M-219/93, Univ. of Hamburg, Dept. of Comp. Sci. (Feb. 1993).

[BrGo94] Bredereke, J. and Gotzhein, R. *Increasing the concurrency in Estelle*. In Tenney, R. L., Amer, P. D., and Uyar, M. Ü., editors, "Formal Description Techniques, VI". North-Holland (1994).

[CDA89] Courtiat, J.-P., Diaz, M., and Ayadi, M. A. D. *Description formelle de protocoles OSI en Estelle*. In "Séminaire Franco-Brésilien sur les Systèmes Informatiques Répartis", pp. 47–55, Florianópolis-SC, Brazil (11–14 Sept. 1989).

[Cla85] Clark, D. D. *The structuring of systems using upcalls*. In "Proc. of the 10th Symposium on Operating Systems Principles (Orcas Island, Wash.)", pp. 171–180, New York (1985). ACM.

[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, USA (Sept. 1990).

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

[CoSa92] Courtiat, J.-P. and de Saqui-Sannes, P. *ESTIM: an integrated environment for the simulation and verification of OSI protocols specified in Estelle**. Comp. Networks and ISDN Syst. **25**, 83–98 (1992).

[25] XEC attempts to find a firable transition based on the effects of the previous transition, and it will succeed immediately in the cases where Henke et. al. can construct an activity thread.

[Cou87] Courtiat, J.-P. *How could Estelle become a better FDT?* In Rudin, H. and West, C. H., editors, "Protocol Specification, Testing, and Verification, VII", pp. 43–60. North-Holland (1987).

[Cou88] Courtiat, J.-P. *ESTELLE*: a powerful dialect of Estelle for OSI protocol description.* In Aggarwal, S. and Sabnani, K., editors, "Protocol Specification, Testing, and Verification, VIII", pp. 171–186. North-Holland (1988).

[Dah95] Dahl, S. *Spezifikation und Bewertung eines effizienten Kommunikationsprotokolls.* Projektarbeit, Univ. of Kaiserslautern, Dept. of Comp. Sci. (Feb. 1995).

[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).

[GBE+96] Gotzhein, R., Bredereke, J., Effelsberg, W., Fischer, S., Held, T., and König, H. *Improving the efficiency of automated protocol implementation using Estelle.* Comp. Commun. **19**(14), 1226–1235 (Dec. 1996).

[Har88] Harel, D. *On visual formalisms.* Commun. ACM **31**(5), 514–530 (May 1988).

[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).

[Hen97] Henke, R. *Generierung effizienter Implementationen aus formal beschriebenen Kommunikationsprotokollen.* PhD thesis, Univ. of Magdeburg, Germany (1997).

[HKMT98] Henke, R., König, H., and Mitschele-Thiel, A. *Automatische Protokollimplementierung unter Verwendung von Activity-Threads.* PIK, Saur Verlag **21**(1), 11–18 (Jan. 1998).

[Hof93a] Hofmann, B. *Generierung effizienter Protokollimplementierungen aus Estelle-Spezifikationen.* PhD thesis, Univ. of Mannheim, Germany, VDI-Verlag, Düsseldorf, Fortschr.-Ber. Reihe 10 Nr. 299 (1993).

[Hof93b] 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.

[ISO81] ISO/TC 97/SC 16, ISO 7498. *Data Processing — Open Systems Interconnection — Basic Reference Model* (1981).

[ISO87a] ISO/TC 97, ISO 8326. *Information processing systems — Open Systems Interconnection — Basic connection oriented session service definition* (1987).

[ISO87b] ISO/TC 97, ISO 8327. *Information processing systems — Open Systems Interconnection — Basic connection oriented session protocol specification* (1987).

[ISO88a] ISO/TC 97, ISO 8649. *Information processing systems — Open Systems Interconnection — Service definition for the Association Control Service Element* (1988).

[ISO88b] ISO/TC 97, ISO 8650. *Information processing systems — Open Systems Interconnection — Protocol specification for the Association Control Service Element* (1988).

[ISO88c] ISO/TC 97, ISO 8822. *Information processing systems — Open Systems Interconnection — Connection oriented presentation service definition* (1988).

[ISO88d] ISO/TC 97, ISO 8823. *Information processing systems — Open Systems Interconnection — Connection oriented presentation protocol specification* (1988).

[ISO92] ISO/IEC JTC 1/SC 21, IS 10026. *Information technology — Open Systems Interconnection — Distributed Transaction Processing* (1992).

[ISO97] ISO/IEC/JTC 1, ISO/IEC 9074:1997(E). *Information Technology — Open Systems Interconnection — Estelle: A Formal Description Technique Based on an Extended State Transition Model*, 2nd ed. (1997).

[ITU94] ITU-T, Recommendation Z.100 (03/93). *CCITT Specification and Description Language (SDL)* (1994).

[JaJe89] Jard, C. and Jezequel, J.-M. *A multi-processor Estelle-to-C compiler to protoype distributed algorithms on parallel machines.* In Brinksma, E., Scollo, G., and Vissers, C. A., editors, "Protocol Specification, Testing, and Verification, IX". North-Holland (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).

[Mal89] Malamud, C. *DEC Networks and Architectures*. J. Ranade DEC Series. McGraw-Hill (1989).

[Mar87] Martin, J. *SNA: IBM's Networking Solution*. Prentice-Hall, Englewood Cliffs, NJ (1987).

[Pen96] Penner, H. *Erstellung eines Software-Monitors zur Analyse automatisch generierter Protokollimplementierungen*. Masters thesis, Univ. of Kaiserslautern, Dept. of Comp. Sci. (Jan. 1996).

[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).

[The97] Thees, J. *Modularer Estelle Compiler (MET) – Benutzerhandbuch*. Univ. of Kaiserslautern, Dept. of Comp. Sci. (1997).

[The98] Thees, J. *Protocol implementation with Estelle – from prototypes to efficient implementations*. In Budkowski, S., Fischer, S., and Gotzhein, R., editors, “Proc. of the Int'l. Workshop on the Formal Description Technique Estelle – Estelle'98”, Evry, France (2 Nov. 1998). Institut National des Télécommunications.

[ThGo97] Thees, J. and Gotzhein, R. *Generation of efficient protocol implementations – an experimental code generator for Estelle and its application to XTP*. In “6th Open Workshop on High Speed Networks”, Stuttgart, Germany (8–9 Oct. 1997).

[ThGo98a] Thees, J. and Gotzhein, R. *The eXperimental Estelle Compiler – automatic generation of implementations from formal specifications*. In “Second Workshop on Formal Methods in Software Practice”, Clearwater Beach, Florida, USA (4–5 Mar. 1998).

[ThGo98b] Thees, J. and Gotzhein, R. *Open Estelle – an FDT for open distributed systems*. In “Formal Description Techniques XI”. Chapman & Hall (Nov. 1998).

[XTP95] XTP Forum, Santa Barbara, USA. *Xpress Transport Protocol Specification, XTP Rev. 4.0* (1 Mar. 1995). URL ftp://dancer.ca.sandia.gov/pub/xtp4.0/xtp4.0-specification-2s.ps.