Increasing the Concurrency in Estelle

J. Bredereke[a] and R. Gotzhein[b]

[a]University of Hamburg, Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany

[b]University of Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern, Germany

Abstract

In general, the potential parallelism expressed in a specification should be as high as possible in order to allow for most efficient implementations. We argue that there are currently important limitations of expressing concurrency in Estelle. Therefore, we propose a minor, fully compatible extension of Estelle, which allows a significantly higher degree of concurrency. The syntactical extension consists of additional module attributes, which allow to specify that child modules of active modules may behave asynchronously. We study the semantical implications of this extension, and present the required changes to the Estelle semantics.

Keyword Codes: C.2.2; D.2.1; D.3.3
Keywords: Network Protocols; Requirements/Specifications; Language Constructs and Features

1. INTRODUCTION

The software development process is usually modeled as a sequence of activities leading from the problem to a software solution. In the area of communication protocols and distributed systems, it is important to consider concurrency (or parallelism) issues throughout these activities. Here, concurrency is a phenomenon that arises naturally. Also, concurrency is of interest for the software development in general, since it can lead to more efficient solutions. In the following, we distinguish the following types of concurrency:

- By *explicit concurrency*, we refer to apparent parallelism. Such parallelism exists, for instance, in systems with interacting components that may be executed in parallel. Explicit concurrency can typically be found in constructive descriptions.
- By *implicit concurrency*, we refer to potential for parallelism that is not immediately apparent, but has to be discovered. This is typically the case in non-constructive descriptions. But implicit concurrency can also be found in constructive descriptions, and can occur together with explicit concurrency.

Further types of concurrency can be distinguished along the stages of the software development process:

- *Problem-inherent concurrency* denotes the potential parallelism of the problem.
- *Specified concurrency* denotes the potential parallelism resulting from the design phase.
- *Implemented concurrency* denotes the actual parallelism realized in the final product.

In each of these stages, we can distinguish between explicit and implicit concurrency. Problem-inherent explicit concurrency, for instance, exists in distributed applications. A communication service is an example of problem-inherent implicit concurrency. Specified explicit concurrency is given by the description of a set of protocol entities. Specified implicit concurrency results, for instance, from input/output assertions allowing for parallel implementations. Implemented explicit concurrency results from the parallel implementation of a set of protocol entities. An example of implemented implicit concurrency is a sequential program for which parallel code executed on a specialized hardware is generated (pipelining). Thus, the above classifications of concurrency are orthogonal.

Problem-inherent concurrency refers to the maximal parallelism of a problem. This parallelism can be exploited to obtain an efficient solution and/or may be reduced in later development stages. For instance, it may be reduced by executing several system components on a single processor. Once it is reduced in an intermediate stage, the resulting specified concurrency becomes the upper bound of parallelism for the following stages including the implementation. Thus, design decisions such as a specific system architecture or even the choice of a particular specification language crucially influence the potential for an efficient implementation. As mentioned before, we distinguish between explicit and implicit concurrency in each design stage. By exploiting implicit concurrency, the explicit concurrency of subsequent design stages may be increased. This means that implicit concurrency is partially made explicit; however, the total concurrency is not increased.

In this paper, we consider and increase the potential of Estelle to express specified explicit concurrency in communication protocols and distributed systems. In Section 2, we describe the current means of Estelle to express this type of concurrency, and we point out why these means are not sufficient for more complex protocols. In Section 3, we propose a minor, fully compatible extension of Estelle which increases significantly the expressible specified explicit concurrency. Also, we list the required changes to the formal syntax and semantics of Estelle, and we address some implementation issues.

2. CONCURRENCY IN ESTELLE

2.1. A sketch of the Estelle semantics

Estelle ([ISO89]) is an FDT designed for the specification of distributed, concurrent information processing systems, in particular communication services and protocols of the OSI Basic Reference Model ([ISO81]). Since 1989, it possesses the status of an international standard. Introductions to Estelle can be found in [Di+89] and [Tur92].

An Estelle specification describes a system consisting of a number of subsystems. A subsystem is (like the entire system) hierarchically structured, it consists of a module instance attributed with `SYSTEMPROCESS` or `SYSTEMACTIVITY`. In the following, we will

also refer to such module instances by "system module instance". The attribute rules determine that the descendants of a system module cannot be system modules themselves. According to the definition of "subsystem", a subsystem cannot contain further subsystems. This has an important effect on the explicit concurrency expressible in Estelle, as we will see.

For the following considerations, two properties of subsystems are relevant. Firstly, the Estelle execution model does not define any synchronization constraints nor priority regulations between module instances of different subsystems. This means that the subsystems can proceed at different speeds, unless the specification itself provides for synchronization by message exchange between different subsystems. The second property follows indirectly from the attribute rules. Because the ancestors of a system module instance are passive (i.e, they only have an `INITIALIZE`-transition), the set of subsystems and the connections between them cannot be modified after the initialization phase. This means that the coarse structure of an Estelle system is static.

Inside a subsystem, fixed priority regulations and additional synchronization constraints that can be specified by module attributes are in effect. These measures enforce that the firing of transitions in a subsystem occurs in "rounds" (or "computation steps", [DeBu89]). In each round, executable transitions are selected and fired according to certain rules. One such rule is that each module instance has priority over all its descendants, independent of its attribute. This means that if in a given round, a module instance has an executable transition, then all its descendants must not fire any transitions in this round. Whether this module instance may fire its transition, depends on further constraints.

The attribute of a module instance determines how the right to fire a transition is passed to child module instances (direct descendants), if it cannot execute a transition of its own. A process module instance (carrying the attribute `PROCESS` or `SYSTEMPROCESS`) passes the right to fire a transition to *all* its child module instances. The child module instances may then fire a transition or pass the right further down the hierarchy. It follows that this type of attribute enables explicit concurrency. The rules for activity module instances (carrying the attribute `ACTIVITY` or `SYSTEMACTIVITY`) are such that no concurrency is possible (apart from implicit concurrency, see Section 1).

From the priority rules and synchronisation constraints, it follows that each module instance of a subsystem can fire at most one transition in a single round. To fire further transitions, it has to wait until the next round. This means that although the child module instances of a process module instance may fire their transitions concurrently, they cannot run at different speeds, as it can, for instance, be the case between subsystems. This strongly limits the usefulness of a subsystem consisting of several module instances for expressing explicit concurrency, and thus for parallel implementations.

There are two important reasons for these very restrictive regulations inside a subsystem. Firstly, a module instance may have exported variables, which can be accessed by itself and by its parent module instance (immediate ancestor). The priority rule ensures that transitions of these module instances will never be executed concurrently, which means that the parent module instance can check the value of an exported variable of a child module instance (using a `PROVIDED`-clause) to determine an executable transition and be sure that the child module instance has not modified that value until the transition is fired. The second reason concerns the dynamic modification of communication links

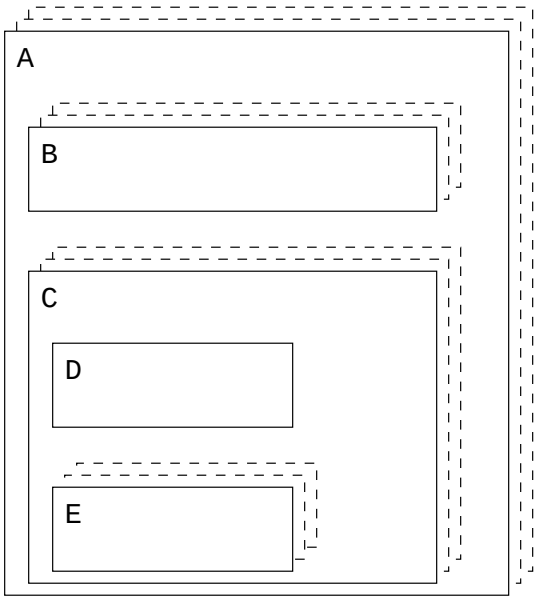

Figure 1. An abstract protocol architecture

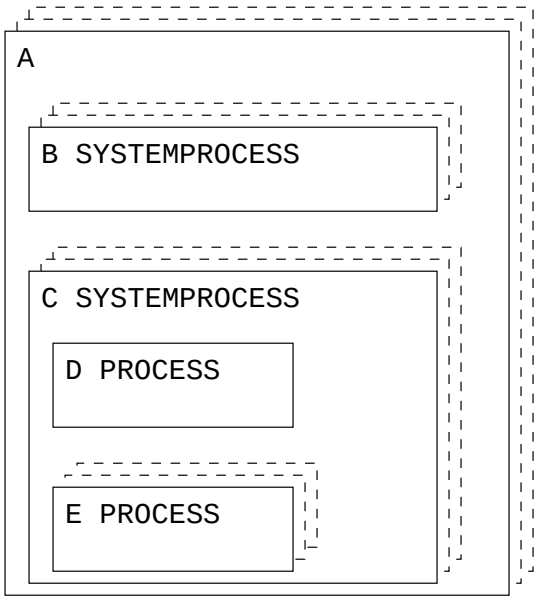

Figure 2. Module A specified as inactive

and the release of module instances, which is possible inside a subsystem.

2.2. Limitations of expressing concurrency in Estelle

Consider an abstract protocol architecture as in Figure 1. It shows an arbitrary, but fixed number of protocol components of type A. Each protocol component of type A contains a variable number of protocol components of types B and C. We suppose that the number of these components may change dynamically. Furthermore, each protocol component of type C contains one protocol component of type D and a variable number of protocol components of type E. Such an abstract protocol architecture and its side conditions are in every respect realistic, they may be found, e.g., in the ISO protocol for distributed transaction processing ([ISO92], see Figure 3). For demonstration purposes, we will use the simpler Figure 1, but everything said may be related directly to Figure 3 by setting `A` = open system, `B` = TPSU, `C` = TPPM, `D` = MACF, `E` = SAO.

The hierarchical protocol architecture of Figure 1 can be expressed directly by a hierarchy of Estelle module instances. But we still have to investigate whether the mentioned side conditions can be met by an appropriate module attributing, and to what degree this constrains the expressible explicit concurrency. One possibility is to specify the protocol components of type A as inactive, not attributed Estelle module instances. In every A-module, (active) B- and C-modules could be declared, and then the necessary number of module instances could be created. To achieve the maximal explicit concurrency expressible within Estelle, these modules could be attributed `SYSTEMPROCESS` (see Figure 2). The disadvantage of this solution is that the resulting Estelle architecture would be static, while the protocol architecture demands the dynamic creation and release of B- and C-module instances.

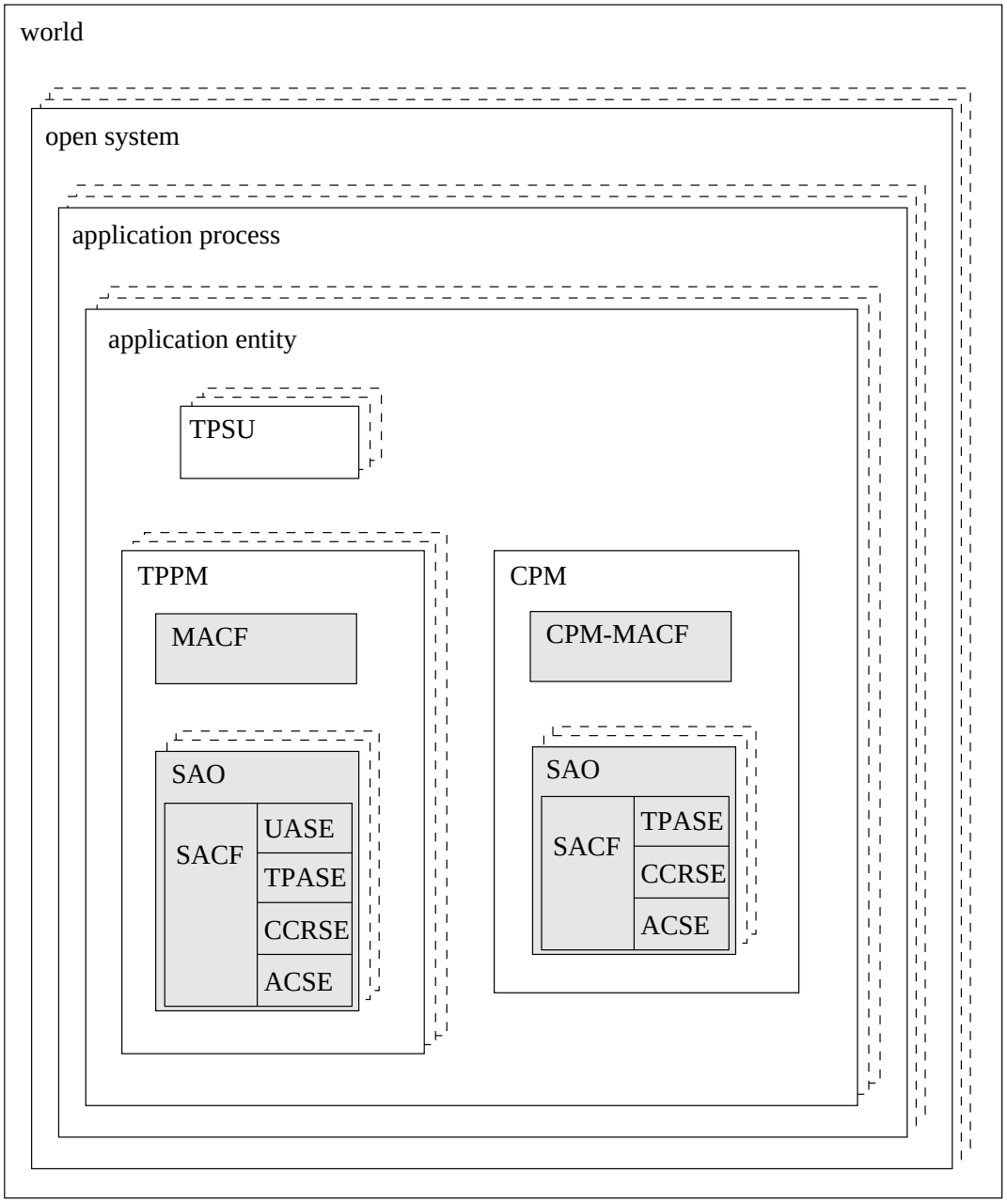

Figure 3. Conceptual structure of ISO-TP

An alternative would be to attribute the A-module by `SYSTEMPROCESS` (see Figure 4). Then, the A-module instances could perform the necessary changes of the Estelle architecture. But this has as a consequence that all B- and C-module instances of an A-module instance would be part of the same subsystem. They would have to synchronize after *every* firing of a transition, even if this would not be necessary with respect to the needs

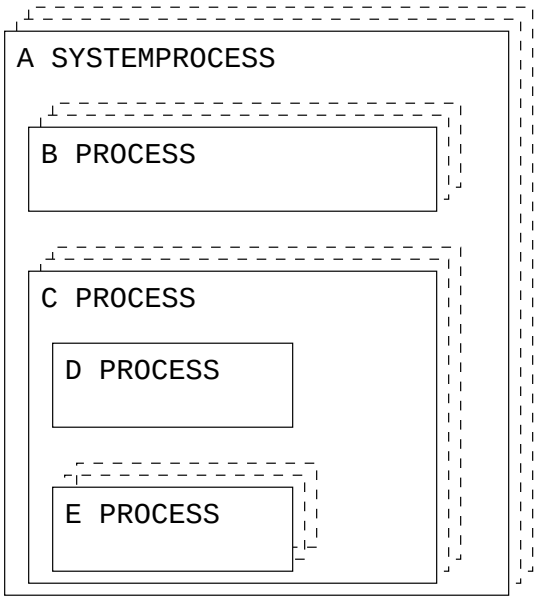

Figure 4. Module A specified as system module

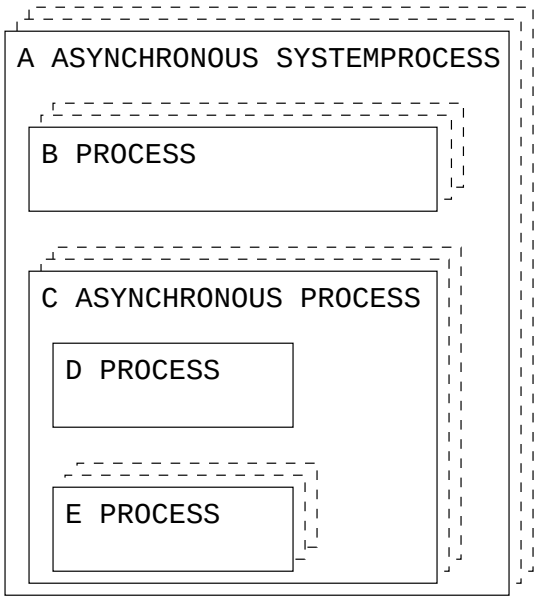

Figure 5. Solution by extending Estelle

of the protocol. So, this solution would have in general[1] the disadvantageous consequence of reducing the problem-inherent explicit concurrency in the specification drastically.

Similar considerations can be made for the mapping of the architecture of the C-component into Estelle. Both discussed alternatives would amount in the D- and E-module instances not to be attributable as system modules, therefore an asynchronous execution being impossible. Even if the first alternative is taken, at this point a considerable reduction of the problem-inherent explicit concurrency would be inevitable, and with it an irrevocable loss of efficiency for the implementation.

3. A FULLY COMPATIBLE ESTELLE EXTENSION FOR INCREASED CONCURRENCY

The difficulties in preserving the problem-inherent explicit concurrency during the specification phase, as discussed in Section 2.2, are of general nature and limit the usability of Estelle as a specification language for *distributed* systems. Concurrency lost in the specification phase cannot be simply reintroduced in the implementation phase, since this would violate the semantics of Estelle. Also, it is not obvious from the specification text which constraints are not required by the application and could actually be dropped. As a consequence, there is the risk of violating the semantics of the application, losing the benefits of the formal approach. It follows that the search for relief should focus on the specification language itself.

To increase the concurrency in Estelle, we propose to

[1] An exact statement can be made only with respect to a specific protocol. E.g., in an individual case the synchronization of the above attributing could meet the needs of the protocol.

extend the attributing of an arbitrary module instance by an option to specify asynchronously parallel child module instances

even if this parent module instance is active and/or dynamically created. These child module instances will not be part of the transition rounds (see Section 2.1) of their parent. Semantically, the proposed extension leads to an *increased set of possible execution paths*, because asynchronously parallel child module instances may run at different speeds. Concerning the example of Section 2.2, the solution to the discussed problem which becomes feasible this way is shown in Figure 5. The details of how such an extension may be put into work will be discussed in the remainder of this section.

3.1. Technical considerations

Access conflicts

In Section 2.1, we said that the rule of priority of a parent module instance over its children has been introduced in order to avoid a number of possible access conflicts. Nevertheless, after reading up to this point, you may suspect that other access conflicts between module instances can still happen, e.g., if one system module instance reads an interaction queue and another one writes to it "at the same time".

With all these kinds of access conflicts, we have to distinguish between two aspects: how they are resolved in the semantics definition, and how a (parallel, concurrent) implementation can be found that implements this resolution efficiently. Up to now, all access conflicts are resolved in the semantics definition by two basic concepts: atomicity of transition firing and interleaving. The concept of atomicity demands that the (visible) effect of a transition can be observed by other parts of the system either completely or not at all. Transitions fire in an instant. But this alone is not sufficient. To resolve the situation when transitions fire "in the same moment", the concept of interleaving defines the possible combined effects of those transitions by the set of all arbitrary sequential orderings of the transition effects.

The concepts of atomicity and interleaving not only resolve all access conflicts in the existing semantics definition of Estelle, they are so powerful that they will also resolve all conflicts introduced by a total asynchrony between a parent and its children. A problem will remain only when an exported variable of a child may change its value after the parent has read it in a `PROVIDED`-clause (cf. Section 2.1).

The aspect of an efficient distributed implementation of the semantics is more difficult. If we drop the parent-child priority for total asynchrony, some additional measures have to be taken in an implementation (e.g., additional synchronization messages), which synchronize those components that otherwise might get into a conflict.

Let us suppose we already have got the means to obtain a distributed implementation respecting the existing Estelle semantics (e.g., Estelle compilers such as [SiSt90], [KrGo93]). Then, besides simply allowing the new child module instances to run asynchronously, an extended version needs to resolve possible conflicts in

- transmitting interactions between parent and child,
- modifying the communication structure,
- creating or releasing a module instance, and
- accessing exported variables.

Transmitting interactions (and, e.g., preserving atomicity) in a distributed implementation renders no new synchronization problem. It has to be solved anyway for the communication between (the implementations of) system module instances. If the implementation of the communication structure is modified, we have to take care that no collision occurs with the transmission of an interaction. This serialization problem is similar to the problem of concurrently writing and reading an interaction queue. Therefore, the same solutions may be applied. The creation of an asynchronous child module instance renders no problem anyway. The release could necessitate a slightly more complex protocol between the implementations of parent and child to assure the atomicity of the `RELEASE`-statement, when it is mapped onto message passing, see [BrGo93] for more details. But since usually both creation and release of module instances are rather rare events, this does not affect the overall efficiency significantly. Exported variables represent the hardest problem. If the distributed implementations of parent and child can communicate only by messages, the necessary synchronization will lead to ineffiency, especially if exported variables are used extensively. In summary, it turns out that an efficient implementation is still possible, except in the case of exported variables.

Therefore, in our extension of Estelle we disallow the declaration of exported variables for the new asynchronous child module instances.[2] (Of course, this does *not* apply to the old "`PROCESS`" or "`ACTIVITY`" child module instances.) By this, we also circumnavigate the only remaining problem in the semantics definition we discussed above.

Firing of transitions

There is a further consequence if we drop the parent-child priority. Up to now, it guarantees two properties:

(a) After all conditions on a transition have been checked and after it has successfully been selected for firing, it may do so without *any* further conditions.

(b) After a transition has been selected for firing, its module instance cannot be released (or modified in another way) by its parent until the transition has been fired. (This assures the implementation of a child module instance to be in a "clean" state when released. It cannot already have started the computation of the next state.)

The second property is invalidated by dropping the parent-child priority. It could be reestablished by imposing conditions on the execution of a `RELEASE`-statement. But this would invalidate property (a), which is a very basic property of the Estelle semantics. The semantics divides the entire transition processing into two phases. In the first phase, *all* firing conditions are checked, and in the second phase the effects of firing are determined. Property (b) only relates to implementations. But there are no fundamental problems in implementing the extension since the implementation of a transition must take care of its atomicity anyway.

So, we decide to preserve the first property and drop the second.

Atomicity of the `output`-operation

The `output`-statement allows a module instance to transmit an interaction, entering it via an interaction point of its own and sending it to another module instance. Consider-

[2] This is no substantial loss, since one can use interactions (i.e., message passing) instead.

ing the formal definition of the transmission procedure in [ISO89], and the requirement that the transmission shall happen atomically, we encounter the need for a technical adjustment.

In an Estelle specification, the effects of a transition are expressed in a sequential manner, like in Pascal. For ease in the formal definition of these effects, [ISO89] therefore uses sub-states which are not visible to the outside world[3], thus preserving the atomicity of transitions. (Also, this kind of definition facilitates sequential implementations.) In the formal semantics definition, interactions sent via "internal interaction points", i.e, those sent inside the same subtree of module instances, are transmitted instantaneously in terms of the sub-states. Those sent via "external interaction points", i.e., maybe to modules progressing concurrently, are first collected. Only at the end of the transition firing, they are flushed to the concurrent "outside world". This is necessary since the destination is determined based on non-local information, which must be accessed atomically only.

Since we drop the parent-child priority principle for asynchronous child module instances, sending to certain internal interaction points now means sending to concurrent module instances, which cannot be regarded as being "local". Therefore, the "collecting of interactions" should also take place for these certain internal interaction points. In Section 3.4, we will present the details.

3.2. Extended syntax

The definition of Estelle in [ISO89] allows five possible attributings of (parent) modules:

- `PROCESS`
- `SYSTEMPROCESS`
- `ACTIVITY`
- `SYSTEMACTIVITY`
- — without attribute —

We add one keyword and thereby two more possibilities:

- `ASYNCHRONOUS PROCESS`
- `ASYNCHRONOUS SYSTEMPROCESS`

Formally, the definition of the non-terminal **class** must be modified to admit the additional module attributes. (All changes to [ISO89] are typeset in *italic*.)

class = *["asynchronous"]* "systemprocess" | "systemactivity"
| *["asynchronous"]* "process" | "activity"

Furthermore, a trivial modification to the **key-words** production must be made to add the new key word, "*asynchronous*"; we do not record the modified production here.

3.3. Additional constraints

The optional ***exported-variable-declaration*** *shall not be used in a* ***module-header-definition*** *closest-contained in a* ***module-body-definition*** *of which the associated* ***module-header-definition*** *used the keyword* ***asynchronous****.*

The optional keywords ***asynchronous systemprocess*** *and* ***asynchronous process*** *may be used exactly where otherwise the keywords* ***systemprocess***

[3]These sub-states are hidden by existential quantification, see [ISO89, clause 9.5.1].

*and **process** would be admissible, respectively. Likewise, all restrictions implied by the use of the keywords **systemprocess** and **process** are also implied by the use of the keywords **asynchronous systemprocess** and **asynchronous process**, respectively.*

The second paragraph refers to the usual constraints on the use of attributes in Estelle. ("No subsystem inside of another subsystem", ...)

In the context of this text insertion, a little more trivial editing is required to add references to "***asynchronous (system)processes***", where only "**(system)processes**" are mentioned. Due to space limitations, we leave this editing out here. For more details, e.g., the exact placement of the added text in the surrounding text, refer to [BrGo93].

3.4. Extended semantics

The proposed extension is not merely a syntactical hint how to generate efficient implementations. As stated in the beginning of Section 3, inserting one of the new attributes into an old specification may lead semantically to an increased set of possible execution paths.

On the other hand, while extending the semantics we took great care to preserve compatibility with regard to existing specifications. Their semantics is touched in no way. In Section 2.1, we already mentioned the execution model of Estelle, i.e., the rules determining the firing of transitions. We stressed the difference between asynchronous system module instances on the one hand and their descendants on the other hand, which have to follow the "rounds" of their respective system module instance. This distinction is reflected in the structure of the formal definition of the Estelle semantics in [ISO89], too.

There, a set of asynchronous system module instances $S_1, \ldots, S_n$ is defined. This set is static. And for every (system) module instance P, a global instantaneous description gid_P is defined, which contains, amongst other things, the description of all child module instances (recursively defining a tree of descendants). Since it is possible in Estelle to create additional child module instances or to release existing ones, this set of child module instances is dynamic.

In the heart of the extension, we decouple the transition selection of the (dynamically created) asynchronous module instances from their parents and add them in this respect to the set of system module instances $S_1, \ldots, S_n$. The decoupling is done in a way such that the rules describing the firing of transitions can be left untouched, e.g., a firing round remains exactly the same. The asynchrony of the new kind of child module instances is described by the same means as for the system module instances. In detail, we propose the following modifications of [ISO89].

In [ISO89, clause 5.3.3], the definition of $AS(gid_P)$ is given, i.e., of the set of transitions selected for execution in a certain gid_P. We add one more subitem to exclude any transitions of asynchronous child module instances from this set:

(b)*(4) If t_P = null and P is an asynchronous process (or asynchronous systemprocess), then $AS(gid_P)$ is empty.*

The case "$t_P \neq$ null" is already handled by subitem (b)(1), which covers parent priority. Both cases together state that a parent of asynchronous children behaves like a leaf in-

stance. In the surrounding "Remark"s and "NOTE"s, some obvious escorting adjustments are required, for details see [BrGo93].

In [ISO89, clause 5.3.4], the progress of an entire system is described, i.e., of a set of system module instances. For a specification module instance SP, the set of system module instances $S_1, \ldots, S_n$ is defined, where n is static during the lifetime of the system because of the static structure of the system module instances. We extend this list by the dynamically created, asynchronous child module instances. The length of the list becomes dynamic, but otherwise the text of [ISO89] can remain nearly unchanged. (This is achieved by denoting the *entire* list by $S_1, \ldots, S_n$ from now on and by denoting the static first part of the list by $S_1, \ldots, S_m$.) For example the rule determining the next global situation remains unchanged.

Only in a few places, a reference to the static set of system module instances must be substituted for the reference to $S_1, \ldots, S_n$. In detail, starting from [ISO89, clause 5.3.4, second paragraph] the text shall be:

> Denote these system modules $S_1, \ldots, S_m$, where $m \geq 1$, and m may vary depending on the initial gid of SP.
>
> *A module may be specified to run in an asynchronous parallel fashion by attributing its parent* ***asynchronous process*** *or* ***asynchronous systemprocess****. Denote the module instances created according to these module definitions by $S_{m+1}, \ldots, S_n$, where $n \geq m$, and n may vary depending on the dynamic structure of module instances.*
>
> Notice that, for every gid_{SP}, its part rooted at S_i ($i = 1, \ldots, m, \ldots, n$) is a [...]
>
> By a global situation of SP, we mean a tuple: (gid_{SP}; $A_1, \ldots, A_m, \ldots, A_n$), where each A_i is a set of transitions of the component instances of the system rooted at S_i.
>
> A global situation of SP is said to be initial if and only if
>
> > gid_{SP} is initial, and
> > $A_i = \emptyset$, for $i = 1, \ldots, m$.

In Section 3.1, we discussed two properties that could not be preserved both under our extension. Accordingly, we perform the following change one paragraph after the last quote:

> The following property is assumed below:
>
> > if two transitions t_i and t_j are selected by S_i and S_j, respectively, and both are defined as partial functions for gid_{SP}, then t_i is also defined for any $t_j(gid_{SP})$ and similarly, t_j is defined for any $t_i(gid_{SP})$, *except if t_j releases S_i or t_i releases S_j*, respectively.
>
> NOTE — For a given gid_P, there are only three ways the execution of a transition t_i may make another transition t_j undefined: [...] In each of these cases, t_i must be a transition of the parent instance of the module instance P_j in gid_{SP}. This is not possible by

the parent/children priority principle of the definition of AS(gid_{SP}) from 5.3.3, *except for asynchronous children.*

Later in Section 3.1, we stated that the "collecting" of sent interactions during transition firing must be applied also to those internal interaction points which transmit to asynchronous descendant module instances. Therefore, we extend the formal definition of the `output`-statement in [ISO89, clause 9.6.6.5] by inserting a test on this condition:

[output p.m(E_1, ..., E_n)]$_P$(s) =
    [...]
    else (i.e., ip $\in$ IIP_P)
        if *asyn-linked(ip, $gid_{P,s}$)* ***then***
            s.out := append (<ip, m, $val_P(E_1)(s)$, ..., $val_P(E_n)(s)$>, s.out)
        else
            if linked(ip, $gid_{P,s}$) [...]

The new predicate "*asyn-linked*" shall be defined similar to the predicate "linked" in [ISO89, clause 9.5.3], except that it shall yield true only for internal interaction points linked to an asynchronous descendant module instance:

asyn-linked(ip, $gid_{P,s}$) iff
    there exist a sequence ip_0, ..., ip_j ($j \geq 1$) of interaction points and a module instance P' such that:
        (a) <ip, ip_0> $\in$ Conn(gid_P),
        (b) <ip_{i-1}, ip_i> $\in$ Att(gid_P), for i = 1, ..., j,
        (c) $ip_{j-1} \in IP_{P'}$, and
        (d) the module definition of P' is attributed as ***asynchronous****.*

3.5. Model

[ISO89, clause 5] describes the general model on which Estelle is based. Here, references to the additional attributes and to the extended expressive power have to be merged into the text in some places. Since this merely ends up in trivial editing, we just refer to the annex of [BrGo93].

3.6. Implementation issues

Prototype compilers

The hitherto existing synchronous parallelism is a special case of the new asynchronous parallelism. In the set of possible firing sequences, the Estelle execution model does not preclude any permutation of the transitions of asynchronous child module instances. (Only constraints specified explicitely may do so.) For synchronous child module instances, only a subset of these permutations is admissible.

An implementation is correct (with respect to "safety"), if, in every execution run, it selects one of the admissible firing sequences. Therefore, any implementation of the hitherto existing synchronous parallelism also is a correct (with respect to "safety") implementation of the asynchronous parallelism! Or, to put it more practically: if we just comment out the key words `ASYNCHRONOUS` in a specification text and feed the result into an existing

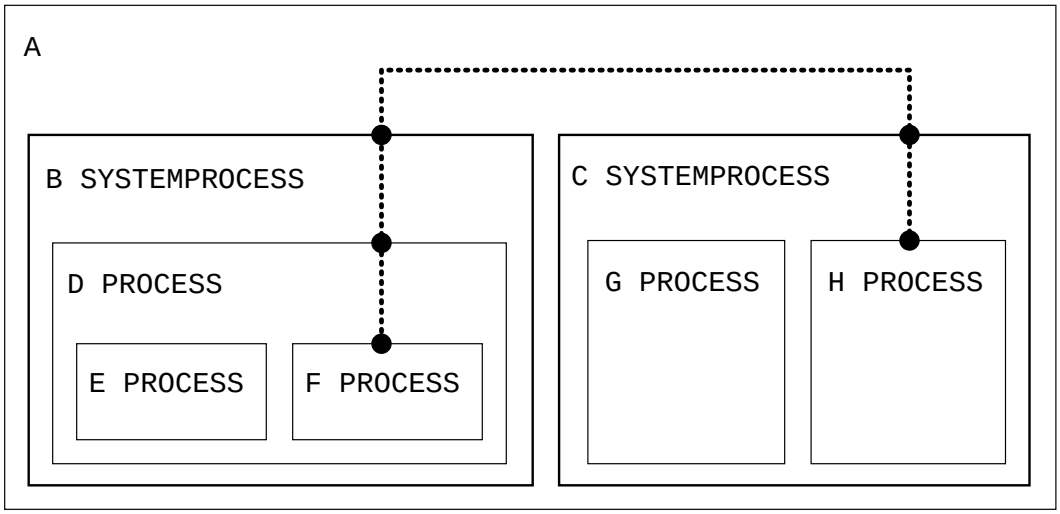

Figure 6. Example of a communication link in Estelle

Estelle compiler, we obtain a correct (with respect to "safety") implementation. But of course, this implementation will probably be quite inefficient.

Up to this point, we used a rather restricted notion of correctness. Essentially, safety requirements ("nothing wrong may happen") on their own can be met by just doing nothing at all. Therefore, one may also put up liveness requirements, i.e., that certain things *must* happen under certain conditions. Possibly, such requirements would not be met anymore by an implementation reduced in the described way.

But the formal definition of Estelle in [ISO89] does not prescribe any liveness requirements. Therefore, many "prototype" tools for our extension are available already, namely, all existing tools for the semantics of [ISO89]. (For generating distributed implementations, e.g., [SiSt90] and [KrGo93]). Of course, the efficiency advantages of our proposed extension are achieved only if real asynchronous parallelism is added to the existing compilers.

Implementation optimization

Figure 6 presents a typical communication link between two module instances. According to the Estelle semantics, all interactions must pass through the interaction points of the module instances which established the link. In a hierarchically structured specification, the interactions may thus pass far up the hierarchy just to be handed down again afterwards. As a consequence, the topmost module instance(s) may easily become a communication bottleneck in a distributed implementation.

Therefore, both the Pet/Dingo toolset for distributed implementations of Estelle specifications ([SiSt90]) and the UHH compiler ([KrGo93]) perform an optimization: Since the communication structure between system module instances is entirely static, the implementations of the system module instances are notified of their respective communication peers once at startup time, enabling them afterwards to address their peers directly. Note

that, for non-system module instances, direct communication would not result in better performance since synchronization messages have to be passed up and down the hierarchy anyway to implement the transition rounds, as explained in Section 2.1.

The described optimization is desirable for the new asynchronous child module instances, too. Otherwise, our gain in efficiency is lost again by the need to synchronize child and parent for just passing through interactions. Of course, this optimization is a little more difficult, for the communication structure now may change dynamically.

So, we take advantage of the observation that changes of the communication structure are in most applications less frequent than transmissions of interactions through it. In these cases, even a relatively expensive synchronization does not impair the overall efficiency. Such a synchronization becomes necessary in order to preserve the atomicity of both a sending and a restructuring transition, if the sending transition of the child circumvents its parent, which in turn may just try to restructure the communication links used by the child.

Therefore, our Estelle extension allows in distributed implementations for a more efficient communication, too, as a consequence of the ability to specify more explicit concurrency.

4. CONCLUSION

In general, the potential parallelism expressed in a specification should be as high as possible in order to allow for most efficient implementations. We have argued that there are currently important limitations of expressing concurrency in Estelle. Therefore, we have proposed a minor extension of Estelle, which allows a significantly higher degree of concurrency. The syntactical extension consists of additional module attributes, which allow to specify that child modules of active modules may behave asynchronously. We have studied the semantical implications of this extension, and have presented the required changes to the Estelle semantics.

Our extension is fully compatible with already existing Estelle specifications. In addition to this, there are even a large number of tools supporting the extension. Every implementation generated by a hitherto existing compiler represents a correct although inefficient implementation of the extended semantics (after a simple syntactic substitution in the specification text).

But this does not mean that our extension is merely a syntactical hint how to generate efficient implementations. Semantically, inserting one of the new attributes into an old specification may lead to an increased set of possible execution paths.

This additional expressiveness for concurrency renders possible the generation of distributed implementations which are substantially more efficient in the case of a high degree of problem-inherent concurrency. This is achieved by removing the need for the transitions of asynchronous child module instances to synchronize after every transition step. In a distributed implementation, up to now this led to a large number of synchronization messages. They were neccessary due to the execution model of Estelle, but in many cases not due to the semantics of the respective application.

Additionally, an optimization in the implementation of the message transfer between the asynchronous (distributed) child module instances becomes possible. The semantics of

the extension does not neccessarily imply that this message transfer is performed through the bottleneck of the parent module instance. Up to now, such an optimization was possible only for the static asynchronous system module instances.

Concerning the complexity of the firing rules for Estelle transitions, our added rule often opens a choice for more *simplicity*. If a specifier restricts his use of module attributes exclusively to those describing asynchronously parallel behaviour, he will obtain a trivial transition selection scheme which just arbitrarily fires any transition that is enabled at any time. No (parent-child) priority or synchronization rules have to be obeyed. Up to now, this restriction meant that other important Estelle features[4] could not be used.

Currently, the described extension is incorporated into the Pet/Dingo tool set ([SiSt90]) as part of a joint research project at the Universities of Hamburg, Magdeburg, and Mannheim. Based on the modified tool set, it is planned to develop parallel implementations of application layer protocols, such as OSI-TP ([ISO92]). Further extensions of Estelle to enhance the efficiency of these implementations are under investigation.

REFERENCES

[BrGo93] Bredereke, J. and Gotzhein, R. *An Estelle extension for increased concurrency*. Tech. Rep. FBI-HH-M-219/93 (in German, Annex in English), University of Hamburg, Dept. of Comp. Sce. (Feb. 1993).

[DeBu89] Dembinski, P. and Budkowski, S. *Specification language Estelle*. In Diaz et al. [Di+89], pp. 35–75.

[Di+89] Diaz, M. et al., editors. *The Formal Description Technique Estelle*. North-Holland (1989).

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

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

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

[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-Series", GI/ITG-Fachgespräch June 1992 (1993). Saur-Verlag.

[SiSt90] Sijelmassi, R. and Strausser, B. *NIST integrated tool set for Estelle*. In Quemada, J., Maños, J., and Vazquez, E., editors, "Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols — FORTE '90", Madrid, Spain (5–8 Nov. 1990). North-Holland.

[Tur92] Turner, K. *Using Formal Description Techniques — An Introduction to Estelle, LOTOS, and SDL*. Wiley (1992).

[4] I.e., hierarchical substructuring of modules and dynamic modification of the module and communication structure.