A Formal Syntax and a Formal Semantics for Open Estelle

J. Thees, R. Gotzhein

Internal Report No. 292/97

A Formal Syntax and a Formal Semantics for Open Estelle

J. Thees, R. Gotzhein

Internal Report No. 292/97

University of Kaiserslautern
Department of Computer Sciences
P.O. 3049
D-67653 Kaiserslautern

Email: {thees, gotzhein}@informatik.uni-kl.de

A Formal Syntax and a Formal Semantics for Open Estelle [1]

J. Thees, R. Gotzhein

University of Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern, Germany

Email: {thees, gotzhein}@informatik.uni-kl.de

Abstract

Estelle is an internationally standardized formal description technique (FDT) designed for the specification of distributed systems, in particular communication protocols. An Estelle specification describes a system of communicating components (module instances). The specified system is *closed* in a topological sense, i.e. it has no ability to interact with some environment. Because of this restriction, *open* systems can only be specified together with and incorporated with an environment.

To overcome this restriction, we introduce a compatible extension of Estelle, called "*Open Estelle*". It allows the specification of (topologically) *open systems*, i.e. systems that have the ability to communicate with any environment through a well-defined external interface. We define a *formal syntax* and a *formal semantics* for Open Estelle, both based on and extending the syntax and semantics of Estelle. The extension is compatible syntactically and semantically, i.e. Estelle is a subset of Open Estelle. In particular, the formal semantics of Open Estelle reduces to the Estelle semantics in the special case of a closed system. Furthermore, we present a tool for the textual integration of open systems into environments specified in Open Estelle, and a compiler for the automatic generation of implementations directly from Open Estelle specifications.

1 Introduction

Estelle [ISO89, DeBu89] is a *formal description technique* (FDT), internationally standardized since 1989. It has been designed for the description of *distributed, communicating, concurrent systems*, in particular communication protocols. An Estelle specification describes a dynamically modifiable hierarchical system of indeterministic components called *module instances*. Every module instance has a well-defined external interface, through which it may interact with other module instances of the same system. However, systems formally specified in Estelle are *closed* in a topological sense, i.e. they have no abilities for external communication. Because of this restriction, *(topologically) open systems* can be described *formally* only together with and embedded into a *concrete environment*. This means that the environment has to be known in advance, and that it must already be determined when the open system is specified.

In [LaFiVe96, GoRoTh96], pragmatic approaches to the specification of systems with the ability to interact with pre-existing real-world environments are proposed. These approaches are based on the use of primitive functions and procedures or pragmatic compiler modifications. According to the Estelle standard, these specifications do not possess a formal semantics[2]. Also, the incorporation of systems specified in such a manner into a formal context is not supported by these approaches.

---

1. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of project Go 503/4-1.

2. Primitive functions and procedures (and also the specification containg them) have no semantic meaning, unless a "rigorous, implementation independent (e.g. mathematical) definition of the relevant block is supplied by the specifier" (Clause 8.2.4.3 of [ISO89]). Such a definition may, however, be difficult to obtain for communication with some environment realized, for instance, via primitive functions.

The ability to formally describe open systems independently of any environment, and to specify their incorporation with different concrete environments, would have the following benefits:

- The *expressiveness* and the *abstraction level* of Estelle would be enhanced. For instance, protocol machines could be formally described and analysed *independently* of concrete user modules or network modules. Their formal semantics would take every possible environment behaviour into account, and would not have to be restricted to the concrete descriptions of users or networks as in case of Standard Estelle[3] specifications.
- Another important application would be the *decomposition of large systems* into a hierarchical set of syntactically independent components, which can then be developed and analysed separately. This would improve the practical development of large systems with Estelle, as it would allow a clean separation between individual components and their interaction in the global system. Also, the *reuse of components* would be supported, because their description could be independent of any concrete environment.
- The formal description of an open system could also serve as a basis to generate an *implementation* with the ability to communicate with pre-existing environments. Different from the practical approaches mentioned before, the Estelle specification on which the implementation is based would have a formal semantics.

In this paper, we introduce a syntactic and semantic extension of Estelle called "*Open Estelle*". It allows the specification of (topologically) open systems, i.e. systems that have the ability to interact with any environment through a well-defined external interface, and their incorporation with different environments. Since the external interface of an open system is given as a separate description, a (compatible) environment can be specified just by means of a reference to this interface description.

In Section 2, we introduce the language elements of Open Estelle. The definition of the Open Estelle syntax can be found in Appendix A. In Section 3, we introduce a formal semantics for Open Estelle based on the Standard Estelle semantics. Section 4 addresses implementation issues, including tool support for Open Estelle, and a case study with the "Xpress Transport Protocol" (XTP). Finally, Section 5 gives a summary and an outlook.

2 The Syntax of Open Estelle

Open Estelle is a compatible extension of Estelle that allows the formal specification of open systems with a well-defined external interface. These open systems can then be incorporated into different contexts, such as further Open Estelle specifications. The description of an open

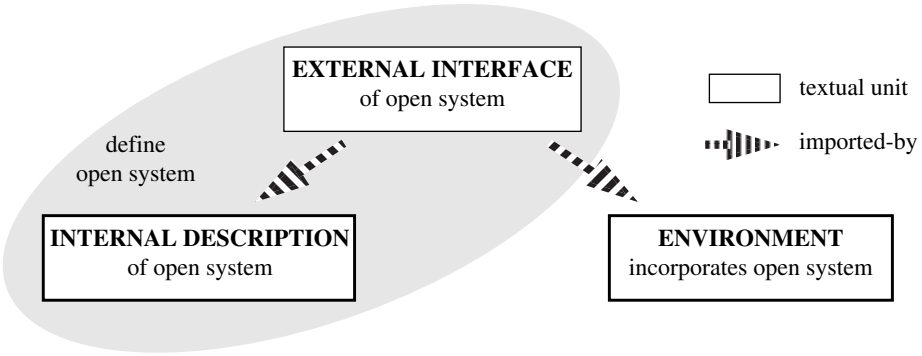

Figure 1: Independent Descriptions for Open System and Environment

3. By *Standard Estelle*, we refer to the Estelle language defined in [ISO89], whereas the proposed extension is called *Open Estelle*.

system consists of the *external interface* and the *internal description*, which are textually separated[4] (see Figure 1). Both are syntactically independent of any possible environment using the open system. Furthermore, Open Estelle environments incorporating an open system only refer to its external interface. This supports the separation of concerns between open systems and their possible environments.

A basic design decision of Open Estelle concerns the representation of an open system. Standard Estelle already includes an abstraction to describe encapsulated systems with well-defined external interfaces and their aggregation into more complex systems: *modules*. Module instances[5] could in principle supply a suitable representation of open systems in Estelle. A module header formally describes the external interface of the module without regard to its internal description. A module body describes the module behaviour. However, in Standard Estelle, a module can only be defined and used inside a specification. Furthermore, being a fragment of a specification, it has no formal semantics of its own.

In the following, we will introduce the syntactical extensions of Estelle for the description of the external interface of open systems (Section 2.1), the internal description of open systems (Section 2.2), and the incorporation of open systems with different environments (Section 2.3). These extensions are based on the syntactical concept of modules. A formal definition of the syntax of Open Estelle can be found in Appendix A.

2.1 Interface Description of Open Systems

The declaration (and thus the external interface) of an open system is given in an INTERFACE-DEFINITION residing in a *separate textual unit*[6] (see Appendix A.1). An INTERFACE-DEFINITION is a container for the *declaration* (not definition) of a set of open systems and for all definitions that are necessary to describe their type and thus their external interface (i.e. module headers, channels, types and constants).

```
INTERFACE binaryService;
    TYPE
        tOperand = RECORD x1, x2: REAL; END;
        tResult = REAL;
    CHANNEL binaryServiceChannel(user, provider)
        BY user: request(x: tOperand);
        BY provider: respond(y: tResult);
    MODULE binaryOperatorHeader ACTIVITY;
        IP toUser: binaryServiceChannel(provider) COMMON QUEUE;
        END;
    BODY binaryOperator FOR binaryOperatorHeader;   { this declares the open system }
        EXTERNAL;                                   { "binaryService::binaryOperator" }
END. { end interface "binaryService" }
```

Figure 2: Example for an Interface Definition

The INTERFACE-DEFINITION is one of the two new start non-terminals of the Open Estelle syntax. Interfaces are uniquely defined, independently of any importing environment (see Section 2.3). This is important for a formal description of the exported definitions. An interface starts with the new keyword "**INTERFACE**", followed by its name. The *declaration* of an open

4. The textual separation between external interface and internal description is similar to Modula-2 [Wir85].
5. A module *describes* an open system, whereas every instance of a module *is* an open system.
6. Since the Estelle standard does not state a representation for a specification text, we use the term "*textual unit*" for self-contained syntactical objects (e.g. a specification).

system inside of an interface is syntactically represented by a MODULE-BODY-DECLARATION, which is a MODULE-BODY-DEFINITION containing the keyword "**EXTERNAL**" (see Figure 2). This declaration refers to a module header that describes the external interface, but it gives no internal description of the module body.

Analogous to Standard Estelle, the appearance of the keyword "**EXTERNAL**" inside a MODULE-BODY-DECLARATION of an INTERFACE-DEFINITION leads to a syntactically correct, but incomplete description. But in contrast to a SPECIFICATION or a BEHAVIOUR-DEFINITION, where only a textual modification can remove this incompleteness, the attachment[7] of an appropriate BEHAVIOUR-DEFINITION (see Section 2.2) leads to a complete description of the declared open systems.

The reader should note that the interface definition itself describes no behaviour (no transitions) and no state (no variables or control states). Consequently, it is only a container for a set of definitions that may be *imported* by other Estelle components (see Section 2.3). It is even possible for an interface definition to import another interface definition. This allows two (otherwise independent) interface definitions to share common definitions such as types, constants and channels (see Figure 5) and therefore allows a clean separation of concerns between open systems that refer to common definitions.

2.2 Internal Description of Open Systems

The internal description of an open system is given by a MODULE-BODY-DEFINITION inside of a "BEHAVIOUR-DEFINITION" residing in a *separate textual unit* (see Appendix A.2). As suggested by the term "BEHAVIOUR-DEFINITION", from an abstract point of view the internal description of an open system only determines the *behaviour* of the open system, because the syntactical interface is already determined by the interface definition.

The BEHAVIOUR-DEFINITION is the second new start non-terminal of the Open Estelle syntax. It begins with the new keyword "**BEHAVIOUR**", followed by its name and a reference to its interface (see Figure 3). The definitions of its interface are imported implicitly (see Section 2.3).

```
BEHAVIOUR binaryAdder FOR binaryService;
    BODY binaryOperator FOR binaryService::binaryOperatorHeader;
        { this defines the open system "binaryService::binaryOperator" }
        TRANS
            WHEN toUser.request(x: tOperand)
            BEGIN
                OUTPUT toUser.respond( x.x1 + x.x2 );
            END; { end of transition-block }
    END; { end of module-body "binaryOperator;" }
END. { end of interface "binaryAdder" }
```

Figure 3: Example for a Behaviour-Definition (see Figure 2)

A BEHAVIOUR-DEFINITION is a container for a set of open system definitions: for every MODULE-BODY-DECLARATION of its interface, it contains exactly one *matching*[8] MODULE-BODY-DEFINITION, and vice versa. The module body defining the open system can make use of all (Open) Estelle constructs for module bodies in its internal description, for example, it can be

7. The attachment of a BEHAVIOUR-DEFINITION to an INTERFACE-DEFINITION is a matter of the interpreting context, since this operation involves different textual units (e.g. UNIX text-files).
8. They have the same name and refer to the same header-definition.

substructured into child modules and it can even import and use other open systems. This allows a very flexible internal structuring of open systems.

2.3 Incorporation of Open Systems into Open Estelle Environments

To be able to incorporate[9] an open system into an environment specified in Open Estelle, the interface that declares this open system has to be *imported* into this environment. The import of an interface is described syntactically by an IMPORT-STATEMENT (see Appendix A.3) inside either a specification, a module-body, an interface, or a behaviour-definition (we will refer to each of them as *importing environment*). An IMPORT-STATEMENT consists of the new keyword "**IMPORT**", followed by a list of identifiers (see Figure 4), which uniquely identify the imported interfaces with the respective names.[10]

The import of an interface defines a *qualified visibility* in the importing environment for all definitions of the interface. This includes (apart from constants, types, channels and module headers) also the open systems declared inside the interface. These definitions can be referred to by means of their *qualified name*, which consists of their unqualified name (given in their definition), preceded by the name of their defining interface and the new symbol "**::**". For example the identifier "**binaryService::binaryOperator**" refers to the definition of "**binaryOperator**" inside the interface "**binaryService**". The reader should note that since the interface identifier uniquely identifies an interface, and since the unqualified name is unique for this interface, every qualified identifier is globally unique.

The open system itself is originated when a new instance of its describing module-definition is created by an appropriate INIT-statement (see Figure 4). It is possible to dynamically create several open systems (i.e. module instances) from the same open system description (i.e. module). In general, all mechanisms that Standard Estelle offers for the handling of instances of locally defined child-modules can also be used with respect to an open system, including communication with the open system or its termination.

```
SPECIFICATION test;
    IMPORT binaryService; { imports definition of interface "binaryService" }
    MODVAR mv: binaryService::binaryOperatorHeader;
    INITIALIZE
        BEGIN
            { create open system by instantiating module "binaryService::binaryOperator": }
            INIT mv WITH binaryService::binaryOperator;
            { the open system can now be handled like a regular module-instance }
        END;
    { ... }
END. { end of specification "test" }
```

Figure 4: Example for the incorporation of an open system (see Figure 2)

Importing an interface is not only useful to get access to the open systems defined inside of it. Since the import of an interface makes *all* of its definitions visible to the importing environment, interfaces are also useful for the global definition of constants, types, channels and headers,

9. By "*incorporate*", we refer to the embedding of the open system (i.e., the *instance*) into the instance of an environment. This should be distinguished from a *textual embedding* of the description of the open system into the description of an Open Estelle environment.
10. The unique mapping of an interface-identifier to an interface is a matter of the interpreting context, since this operation involves different textual units (e.g. UNIX text-files).

which can be used in different specifications, behaviour definitions, or even interfaces. This aspect is particularly important for the separation of concerns of independent interfaces with common definitions; e.g. in Figure 5 the separated interfaces **intC** and **intS** declare open systems with compatible external interaction points. This is possible because both of them import interface **intA**, which contains the appropriate channel-definition. This it allows to specify a system of interfaces that exactly models the type dependencies of the global system.

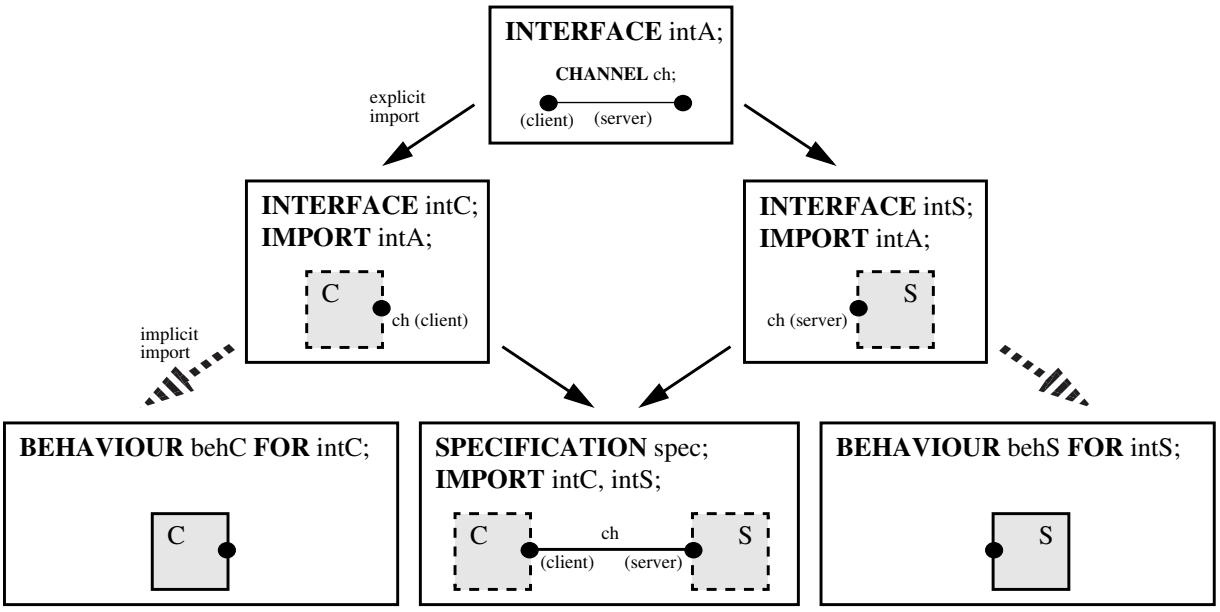

Figure 5: Multiple interfaces with common definitions

2.4 Module Attribution

The module attribution rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO89] are not directly suitable for the application in Open Estelle, since they only relate to the textual nesting of modules. Therefore, we introduce extended module attribution rules (see Appendix A.4), which imply the same attribution scheme for the dynamic module instance tree as the original rules. They allow to specify both, open systems that contain one or more subsystems[11] (which can be incorporated into unattributed environments) and open systems that can be part of other subsystems. In particular, it is possible to specify open systems that can be incorporated into *arbitrarily attributed importing environments*.

3 Semantics of Open Estelle

Estelle qualifies as a formal description technique, meaning that it has both a formal syntax *and* a formal semantics. The proposed language extension should therefore cover both aspects to preserve the formality of Estelle. In order to define the meaning of Open Estelle specifications formally, we have devised a semantics that allows to model interaction between an open system and an environment. More precisely, the semantics captures all potential behaviour of an open system in *all possible* environments; this potential behaviour is reduced when the open system is incorporated with a particular environment.

In the following, we will describe this semantics in some detail. We will argue why despite the different nature of Open Estelle it is feasible to define its semantics in terms of computations as

11. With the term *"subsystem"* we refer to instances of system modules.

in the Standard Estelle semantics. This has the advantage that the language extension is fully compatible with Standard Estelle, and that it will be straightforward to argue why the incorporation discussed above is complete and sound.

3.1 Formal definition of the Open Estelle semantics

According to the Estelle standard [ISO89], the semantics of a Standard Estelle specification SP is formally given by the set of its *computations*. Computations are sequences $<sit_0,sit_1,....>$ of so-called *global situations*, where sit_0 is initial, and for all $j>0$, sit_j is a possible next global situation of sit_{j-1} with respect to the next-state relation. Roughly speaking, a global situation comprises the local states of all module instances (state of input queues, values of local variables, module structure, connection structure) and sets of transitions selected for firing. A next global situation results from either firing a previously selected transition, or selecting a set of transitions according to certain rules. Thus, an Estelle specification defines a *transition system* (S, δ), where S and δ are related to the set of global situations and the next-state relation, respectively. Concurrency is modeled by interleaving.

We will now generalize this semantics to capture the meaning of Open Estelle specifications. In particular, the formal semantics of Open Estelle will reduce to the Estelle semantics in the special case of a closed system, namely systems without external interaction points and without exported variables.

To start with, let us consider the definition of the *next global situations* and *computations* as given in the Estelle standard:

> **Definition 1 (*next global situations*, *computation*; Clause 5.3.4, [ISO89]):**
>
> Given a global situation, $sit = (gid_{SP}; A_1,...,A_n)$, the *set of next global situations* is described as follows:
>
> (a) For every $i = 1,...,n$: if $A_i = \emptyset$, then for every $AS(gid_{SP}/S_i) \in AS^*(gid_{SP}/S_i)$, $(gid_{SP}; A_1,...,AS(gid_{SP}/S_i),...,A_n)$ is a next global situation of sit.
>
> (b) For every $i = 1,...,n$: if $A_i \neq \emptyset$, then for every $t \in A_i$, $(t(gid_{SP}); A_1,...,A_i\backslash\{t\},...,A_n)$ is a next global situation of sit.
>
> NOTE — There are as many next global situations of the situation sit as there are possible choices of next transition t (and its results) in case (b), and different empty sets A_i in sit, for the case (a). In addition, in case (a), all possible choices of the set AS resulting from nondeterminism of each component process in the system rooted at S_i must be taken into account.
>
> NOTE — [...]
>
> A sequence of global situations of SP, $sit_0,sit_1,...,sit_j,...$ is called a *computation of SP* if and only if sit_0 is initial, and for every $j > 0$, sit_j is one of the next global situations of sit_{j-1} as described by (a) or (b) above.

For each subsystem represented by a system module, either a new set of fireable transitions is selected, or an already selected transition is executed. If a transition t of module instance P is fired, the outputs of t are transmitted to the destination queues as part of the global effects of t, as defined by $transmission_P(gid'_{SP})$, where $gid'_{SP} \in [t]_P(gid_{SP})$ (see Clause 9.5.4, [ISO89]). If, for a given output, no destination queue is defined, then the interaction is discarded. Note that all destination queues are represented in gid'_{SP}, therefore, all global effects of t can be applied directly. Furthermore, exported variables may be modified as part of the local effects of t.

In order to generalize this definition to Open Estelle specifications, we have to incorporate the interaction of the specified system with its environment in some way. More specifically, we have to incorporate receptions from, transmissions to, and modifications of local variables by the environment. Recall that these interactions occur through external interaction points and exported variables (see Appendix A.1). However, as the environment is not determined, we have to consider all possible environments, which amounts to taking all possible receptions, transmissions, and assignments to exported variables into account[12]. We will first deal with receptions from the environment and assignments to exported variables by extending Definition 1. Then, we will address transmissions to the environment.

All interactions with the environment occur through external interaction points and exported variables. As these interaction points and the exported variables are typed, the set of interactions that may be received from the environment by module instance P through an external interaction point ip ($receive_P(ip)$, see Clauses 9.3.1 and 9.4.3, [ISO89]), as well as the set of values that may be assigned to an exported variable e [13], are determined. To take all environments into account, we model all possible receptions and assignments by extending the set of next global situations as defined below. Furthermore, we consider global situations w.r.t. an arbitrary module instance P, not just the specification SP.

Definition 2 (*next global situations*, *potential computation*; Open Estelle):

Given a global situation, $sit_P = (gid_P; A_1,...,A_n)$, of a module instance P, the *set of next global situations* is described as follows:

(a) For every i = 1,...,n: if $A_i = \emptyset$, then for every $AS(gid_P/S_i) \in AS^*(gid_P/S_i)$:
$(gid_P; A_1,...,AS(gid_P/S_i),...,A_n)$ is a next global situation of sit_P.

(b) For every i = 1,...,n: if $A_i \neq \emptyset$, then for every $t \in A_i$:
$(t(gid_P); A_1,...,A_i\backslash\{t\},...,A_n)$ is a next global situation of sit_P.

(c) For every $gid_P' \in env_mod^+(gid_P)$: $(gid_P'; A_1,...,A_n)$ is a next global situation of sit_P.
env_mod is defined as follows[14]:

(c1) For every $ip \in EIP_P$: for every $<m,v_1,...,v_k> \in receive_P(ip)$:
$received_P(gid_P, <m,v_1,...,v_k>) \in env_mod(gid_P)$.

(c2) For every $e \in$ EV-id_M, where $P \in INST(M,B,E)$, and e is of type T:
for every $v \in E(T)$: $assign_P(gid_P, e, v) \in env_mod(gid_P)$.

NOTE — $sit_P = (gid_P; A_1,...,A_n)$ is the global situation of module instance P, where gid_P is defined as usual, and each A_i is a set of transitions of the component instances rooted at (α) P, if P is attributed, or (β) S_i, if P is not attributed, and S_i are system modules. This generalizes the definition of global situations to arbitrary module instances. If P = SP, the definition of sit_P is identical to that of sit in the Estelle standard.

NOTE — There are as many next global situations of the situation sit_P as there are possible choices of (a) different empty sets A_i in sit_P and possible choices of the set AS resulting from nondeterminism of each com-

12. Note that the effects of a "terminate" or "release" statement referring to the open system have no impact on the set of computations. In this paper, for reasons of simplicity, we do not consider the effects of "attach" and "detach" statements of the environment involving (directly or indirectly) external interaction points of open systems. Also, for the same reasons, we do not consider the effects of connecting two external interaction points of the same open system.

13. We assume that the type of e is well-defined, i.e. its declaration does not include type-identifiers associated with the "..." construct.

14. env_mod^+ is the transitive closure of env_mod.

ponent process in the system rooted at S_i, (b) next transitions t (and their results), (c) inputs from the environment and assignments to exported variables of P by the environment.

NOTE — For closed systems (i.e. module instances that have no external interaction points and no exported variables) Clause (c) of the definition above has no effects. Consequently, the "next global situations" relation given above reduces to the one of Definition 1.

NOTE — EIP_P is the set of external interaction points of instance P (Clause 9.4.3, [ISO89]). If P = SP, then EIP_P is empty, and no inputs will be received from the environment. $EV\text{-}id_M$ is the set of exported variables of P as declared in its module header M (see Clause 9.4.1, [ISO89]). If P = SP, then no exported variables are defined. If EIP_P and $EV\text{-}id_M$ are both empty, Definition 2 is equivalent to Definition 1 of the Estelle standard.

NOTE — $receive_P(ip)$ is the subset of Interactions (Clause 9.3.1, [ISO89]) that the instance P can receive through interaction point ip (Clause 9.4.3, [ISO89]). $received_P(gid_P)$ is obtained from gid_P by replacing s'.ie(ip') by append(<ip',ip,m,v_1, ..., v_k>, s'.ie(ip')), where downattach(ip) = ip', ip' in $EIP_{P'}$, and s' is the local state of P' (see Clause 9.5.4, [ISO89]). This includes the special case that ip is not attached, i.e. downattach(ip) = ip.

NOTE — $assign_P(gid_P,e, v)$ is a new gid of P where the difference with gid_P is expressed by s.Loc($alloc_B$(e)) := v, where s is the local state of P (see Clause 9.5.4, [ISO89]).

A sequence of global situations of P, <sit_0,sit_1,...,sit_j,...> is called a *potential computation of P* if and only if sit_0 is initial[15], and for every j > 0, sit_j is one of the next global situations of sit_{j-1} as described by (a), (b), or (c) above.

Compared to the Standard Estelle semantics, there are two important differences. The first difference concerns the definition of global situations sit_P for arbitrary module instances P, i.e. not only for the distinguished module instance SP as in Definition 1. In order to model the execution of an open system, we generalize the notion of global situation by combining gid_P with sets of transitions selected for execution.

The second difference concerns the reception of interactions from the environment and the assignments to exported variables of module instance P. This is described by the transitive closure of a function env_mod in (c). Interactions can be received through and only through the external interaction points of module instance P, as expressed in (c1). Note that env_mod ranges over all ip $\in$ EIP_P and all <m,v_1,...,v_k> $\in$ $receive_P$(ip). Thus, all possible environments are taken into account. If an external interaction point of P is attached, then any reception is appended to the destination queue at the end of the attach chain. Assignments to exported variables of module instance P are expressed by (c2). Note that env_mod ranges over all e $\in$ $EV\text{-}id_M$ and all v $\in$ E(T), where e is of type T. Thus, again, all possible environments are taken into account.

Since the behaviour that is described by the definition of the next global situation may in general only occur when the open system is composed with some environment, we use the term *potential computation of P* to denote a sequence of global situations satisfying the aforementioned condition. The reason is that in our definition, the behaviour of the environments as captured by (c) does not yet exist. Once the environment is completely determined, the *set of potential computations* is reduced to a *set of computations* in the sense of [ISO89]. This is, for instance, the case if we consider a closed system. Here, the next global situations are completely described by (a) and (b), which gives evidence that the semantics of Open Estelle is indeed compatible with the semantics of Standard Estelle.

Having dealt with receptions from the environment and assignments to exported variables, we will now address transmissions to the environment. In the Standard Estelle semantics, all out-

15. The initial global situations of an open system take into account all possible actual module parameters and all possible modifications by the environment according to (c) above, which may take place during the execution of a transition executing the init-statement of the open system.

puts of a transition t of P are collected in the local state component s.out as the result of its local effects defined by $[t]_P(s)$ (see Clause 9.6.6.5, [ISO89]). The function $transmission_P$ then defines the global effects by appending, in the same order, the elements of the sequence s.out to the destination queues (Clause 9.5.4, [ISO89]). If for some output at interaction point ip, there is no destination queue, i.e. linked(ip,gid_{SP}) is false, then that output is discarded (see Clauses 9.5.3 and 9.5.4, [ISO89]).

The treatment of transmissions in Open Estelle directly follows the Standard Estelle semantics. If the destination queue of an output belongs to the open system, the meaning of a transmission is just the same. However, if the destination queue is outside the open system, i.e. an output is made at some external interaction point of the root module P, or at some interaction point that is attached to it, then the output leaves the open system[16]. This means that the effects of the output on a possible environment are not visible in the open system, and therefore not represented in its state. Transmissions to the environment can therefore be modeled by discarding the corresponding outputs. As this is already handled by the definitions in [ISO89] (see Clauses 9.5.3 an 9.5.4), the definitions of the Standard Estelle semantics can be used[17].

3.2 Completeness and Soundness of the Open Estelle Semantics

In this section, we provide arguments why we consider the Open Estelle semantics defined in Section 3.1 to be complete and sound. In Figure 6, open systems Pa and Pb are shown, where Pb is incorporated into Pa. This means that Pa, being open itself, partially determines the environment of Pb. *Completeness* of the Open Estelle semantics means that the potential computations of Pa, when "projected onto" Pb, are a subset of the potential computations of Pb. In other words, the meaning of the open system Pb is reduced when it is incorporated into some environment. *Soundness* means that only potential computations are defined for Pb that are possible in some environment.

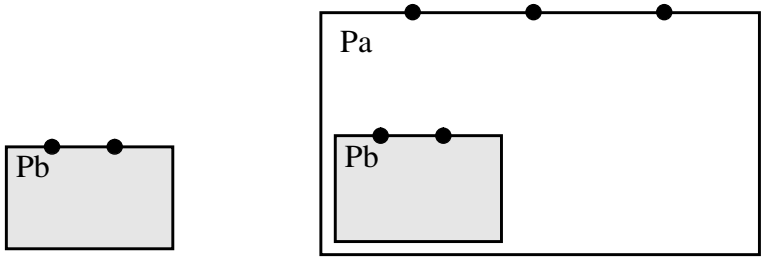

Figure 6: Open systems Pa and Pb

The proof of this claim is by construction and induction. First we show how we project potential computations of Pa onto Pb.

To start with, we define the *projection of global situations* of Pa onto Pb. Let $sit_{Pa} = (gid_{Pa}; A_1,...,A_n)$. Let $P_1,...,P_n$ be the module instances associated with $A_1,...,A_n$, respectively. Then the *projection of sit_{Pa} on Pb*, denoted as $\pi_{Pb}(sit_{Pa}) = (gid_{Pb}; A_1',...,A_m')$, is defined as follows:

- gid_{Pb} is the subtree of gid_{Pa} with root module instance Pb
- if there exists $i \in \{1,...,n\}$ such that Pb is a descendant of P_i or $Pb = P_i$, then $m = 1$ and $A_1' = \{t \in A_i \mid t \text{ is transition of Pb}\}$

16. As already noted above, we do not consider the effects of connecting two external interaction points of the same open system.
17. To be precise, the functions $sent_P$, $received_P$, and $transmission_P$ have to be applied to $gid_{P'}$, where P' is the root module instance of the open system, instead of gid_{SP} (see Clause 9.5.4, [ISO89]).

- if for every i $\in$ {1,...,n}, Pb is no descendant of P_i and Pb $\neq P_i$,
  then there exists a maximal (possibly empty) set of subsystems $\{P_1', ..., P_m'\}$
  $\subseteq \{P_1, ..., P_n\}$ of Pb, and a strict monotonic function f: {1,...,m} $\rightarrow$ {1,...,n} such that
  for every j $\in$ {1,...,m}: $P_j' = P_{f(j)}$, with $A_1', ..., A_m'$ being the associated sets of transitions.

The projection of global situations of Pa onto Pb is extended to potential computations by applying it to each global situation of the computation in which Pb is part of Pa, and by *eliminating* all steps without modification of $\pi_{Pb}(sit_{Pa})$, i.e. "*stuttering steps*".

The projection of potential computations of Pa onto Pb can be understood as the *semantics of the open system* Pb *when incorporated* into the environment Pa. To show why the semantics defined in Section 3.1 is complete, we argue that the projections of potential computations of Pa onto Pb are potential computations of Pb. The proof is by structural induction, based on Definition 2 (see Figure 7). For each case (a) through (c), it is straightforward to show that for each sit_{Pa} and next global situation sit_{Pa}' according to Definition 2, $sit_{Pb}' = \pi_{Pb}(sit_{Pa}')$ is a next global situation of $sit_{Pb} = \pi_{Pb}(sit_{Pa})$ according to Definition 2, or $sit_{Pb} = sit_{Pb}'$ (i.e. a stuttering step).

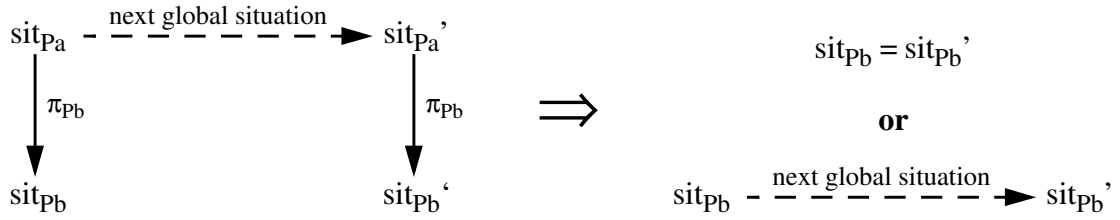

Figure 7: Proof sketch for completeness of the semantics

To show that the semantics is *sound*, we argue that for each potential computation $<sit_0, sit_1, ...>$ of an open system Pb, there exists an environment Pa incorporating Pb and a computation of Pa whose projection onto Pb is $<sit_0, sit_1, ...>$. The proof is by construction of an environment that can indeterministically send any possible interaction through the external interaction points of the open system and assign any possible value to the exported variables of the open system.

4 Implementation Issues

Estelle not only allows the formal specification of systems, but can also serve as a basis for their automated implementation. Several tools for the automatic implementation of Estelle specifications are currently available. Consequently, the automatic creation of implementations directly from specifications is also an important objective of the extension of Estelle, especially since it is just the openness of systems described with Open Estelle that allows to establish a well-defined relationship with their implementations: open systems formally specified with Open Estelle can be implemented with the ability to interact with real-world systems (such as operating systems, communication networks or applications) through their *well-defined external interface* and with a *well-defined semantics*. This opens the possibility for a direct practical incorporation of formally specified open systems with real-world environments.

4.1 Tool Support

To create a platform for practical experiments, we have developed a tool set for the processing of Open Estelle sources. The *front end* of this tool set is a compiler that translates Open Estelle sources (i.e. interfaces, specifications, and behaviour-definitions) into a binary intermediate

form, which can be processed by the other tools. This front end was developed out of the existing Estelle compiler front end PET (“Portable Estelle Translator”, [SiSt93]).

Further, we have developed a completely new optimizing *code generator* for Estelle and Open Estelle, which allows to create C++-code for open systems and importing environments independently of one another. It is possible to compile these components separately and to delay the “fusion” of open systems and their environments to the moment of linking the created machine-object-files into an executable. The compiled open systems can be used also by hand-crafted environments and therefore can be used incorporated and communicating with a real-world environment.

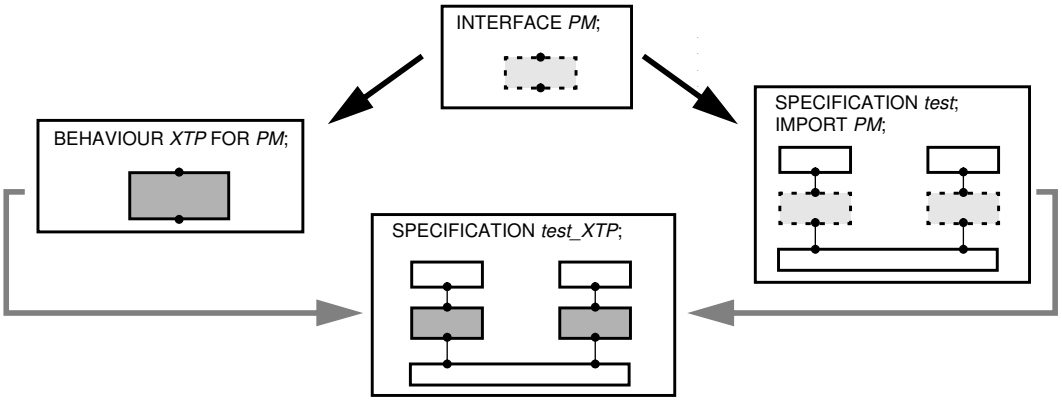

Figure 8: Embedding of an open system into a compatible environment

Another tool for the processing of open systems allows the *textual embedding* of a set of open systems into an Open Estelle environment that incorporates them. This method finally leads to an equivalent *closed Standard Estelle specification*, which includes the formerly open systems in form of local module bodies (see Figure 8). This tool allows the application of existing formal methods for Standard Estelle to systems consisting of a set of Open Estelle descriptions.

4.2 Case Study

As a case study for the practical use of Open Estelle and of the tools presented so far, we have split an existing, relatively large Estelle specification (more than 7500 lines of specification text) of the *Xpress Transport Protocol* (XTP, [XTP95], see Figure 9) into a set of open systems specified in Open Estelle. We have incorporated the XTP protocol machines (XTP_PM) with different Open Estelle and real-world environments, and with different client and network implementations without having to re-compile the generated machine object file.

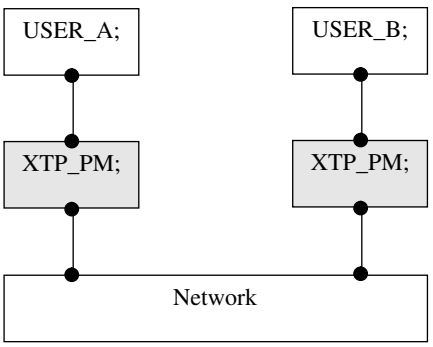

Figure 9: An application structure for XTP

The case study has demonstrated the benefits of Open Estelle for the specification of large systems: one single compilation of the relatively complex protocol machines can serve all implementations of Estelle or real-world environments that incorporate it. A further structuring of the XTP protocol machine itself into smaller open systems additionally supported the maintenance and development of the protocol machine, because it simplified the individual compilation units, shortened the turn-around times after local modifications of components of the protocol machine and clarified the type dependences between the components. In particular, it allowed a clean separation between private definitions of a module and definitions which are exported to the former child modules.

5 Summary and Outlook

In this paper, we have introduced an extension of Estelle, called "*Open Estelle*". It allows the specification of (topologically) *open systems*, i.e. systems with the ability to communicate with any environment through a well-defined external interface. We have defined a *formal syntax* and a *formal semantics* for Open Estelle. The extension is compatible with Estelle both syntactically and semantically, i.e. Estelle is a subset of Open Estelle. In particular, the formal semantics of Open Estelle reduces to the Estelle semantics in the special case of a closed system. Furthermore, we have developed a set of tools supporting Open Estelle, including a new code generator that allows to create efficient implementations of open systems, which can be incorporated with different Open Estelle and hand-crafted environments. We have demonstrated the use of Open Estelle and of our tools by means of a case study with the Xpress Transport Protocol.

There are still some aspects of Open Estelle that need further consideration. An important issue that is directly related to the Estelle semantics is the validation of correctness, based on an implements-relation. We expect that the Open Estelle semantics will give rise to an abstract semantics in terms of input/output-behaviour, which may serve as a basis for a correctness notion. We plan to investigate this issue in more detail.

Another issue is the automatic generation of efficient implementations of formally specified open systems and their integration into pre-existing real-world environments. We are currently extending the specification of the XTP protocol machines in order to be able to communicate directly even with existing hand-crafted XTP implementations (such as SandiaXTP, [SNL96]), by means of the binary packet format of the XTP protocol.

Acknowledgements

The case study discussed in Section 4.2 was based on an Estelle specification of XTP 4.0 that was kindly made available by Stanislaw Budkowski (INT, France) and Octavian Catrina (PU Bucharest, Romania).

References

[GoBo95] Gotzhein, R., Bochmann, G.v.: *Specialization in Estelle*. In: S. T. Vuong, S. T. Chanson (Eds.), *Protocol Specification, Testing, and Verification XIV*, Chapman & Hall, 1995, 21-36

[DeBu89] Dembinski, P., Budkowski, S.: Specification Language Estelle, in: M. Diaz et al. (eds.), *The Formal Description Technique, Estelle*, North-Holland, 1989, pp. 35-75

[GoRoTh96] Gotzhein, R., Rößler, F., Thees, J.: *Towards "Open Estelle"*, in: *6. GI/ITG Workshop "Formal Description Techniques for Distributed Systems"*, Erlangen-Nürnberg, Germany, May 1996

[ISO89] ISO/TC97/SC21: *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21, IS 9074, 1989

[LaFiVe96] Lallet, E., Fischer, S., Verdier, J.-F.: *A new approach for distributing Estelle specifications*, in: Formal Description Techniques VIII, Chapman & Hall, 1996

[RiCl89] Richard, J. L., Claes, T.: *A Generator for C-Code for Estelle*, in: M. Diaz et al (eds.), The Formal Description Technique Estelle, North-Holland, 1989, pp. 397-420

[SNL96] Sandia National Laboratories: *SandiaXTP Reference Manual*, Rev. 1.4, Sandia National Laboratories, USA, 1996

[SiSt93] Sijelmassi, R., Strausser, B.: *The PET and DINGO tools for deriving distributed implementations from Estelle*, Computer Networks and ISDN Systems 25, 1993, pp. 841-851

[Wir85] Wirth, N.: *Programming in Modula-2*, Springer, Stuttgart, 1985

[XTP95] XTP Forum, *Xpress Transport Protocol Specification*, XTP Rev. 4.0, XTP Forum, Santa Barbara, USA, 1995

Appendix A: Language Elements of Open Estelle

This appendix defines the language elements of Open Estelle based on and extending the Standard Estelle language definition given in [ISO89]. Since Open Estelle is a proper extension of Standard Estelle, only new productions and extensions of existing productions are given. In the latter case, the productions and constraints given in this appendix take precedence over the Standard Estelle productions.

Besides the start symbol SPECIFICATION of the Standard Estelle grammar, the Open Estelle grammar has two additional start symbols: INTERFACE-DEFINITION (Appendix A.1) and BEHAVIOUR-DEFINITION (Appendix A.2). Accordingly there are three distinct[18] types of textual units[19] containing a terminal string produced from one of the three start symbols: *Specification-textual-units* (in case of SPECIFICATION), *interface-textual-units* (in case of INTERFACE-DEFINITION) and *behaviour-textual-units* (in case of BEHAVIOUR-DEFINITION).

In Clauses 7.1.2.1, 7.1.2.2, 7.1.2.9, and 7.1.2.10 of [ISO89], the occurrences of "SPECIFICATION" shall be replaced by "SPECIFICATION or INTERFACE-DEFINITION or BEHAVIOUR-DEFINITION". In Clauses 7.1.2.1, 7.1.2.3, and 7.1.2.9 of [ISO89], the occurrences of "BODY-DEFINITION" shall be replaced by "BODY-DEFINITION or INTERFACE-DECLARATION-PART or BEHAVIOUR-DECLARATION-PART". In Clause 7.1.2 of [ISO89] all occurrences of "IDENTIFIER" shall be replaced by "IDENTIFIER or QUALIFIED-IDENTIFIER".

NOTE — All identifiers contained by a SPECIFICATION, INTERFACE-DEFINITION or BEHAVIOUR-DEFINITION are
(1) required constants, types, procedures, or functions of Standard Estelle (see Clause 7.1.2.10 of [ISO89]) or
(2) have their defining-point inside the containing text (see Clause 7.1.2.3 of [ISO89]) or
(3) have their defining-point inside an *imported interface* (see Appendix A.3).

A.1 Interface Definition

A.1.1 Syntax

```
INTERFACE-DEFINITION =          "INTERFACE" IDENTIFIER ";"
                                [ DEFAULT-OPTIONS ]
                                [ IMPORT-OPTIONS ]
                                INTERFACE-DECLARATION-PART
                                "END" "." .
INTERFACE-DECLARATION-PART =    { INTERFACE-DECLARATIONS } .
INTERFACE-DECLARATIONS =        CONSTANT-DEFINITION-PART
                                | TYPE-DEFINITION-PART
                                | CHANNEL-DEFINITION
                                | MODULE-HEADER-DEFINITION
                                | MODULE-BODY-DECLARATION .
MODULE-BODY-DECLARATION =       "BODY" IDENTIFIER "FOR" HEADER-IDENTIFIER ";"
                                "EXTERNAL" ";" .
```

A.1.2 Constraints

With the exception of module attribution rules[20] (Clause 7.3.6.2 of [ISO89]), all constraints and interpretation rules for SPECIFICATION (Clause 7.2 of [ISO89]) shall also be valid for INTERFACE-DEFINITION, the ones for DECLARATIONS (Clause 7.3 of [ISO89]) shall also be valid for

18. The first keyword of a textual unit ("**SPECIFICATION**", "**INTERFACE**" or "**BEHAVIOUR**") uniquely identifies its type.
19. Since the Estelle standard does not state a representation for a specification text, we use the term "*textual unit*" for self-contained syntactical objects (e.g. a specification).
20. Module attribution is handled in Appendix A.4.

INTERFACE-DECLARATIONS, and the ones for MODULE-BODY-DEFINITION (Clause 7.3.7 of [ISO89]) shall also be valid for MODULE-BODY-DECLARATION. Applying scope rules of Clause 7, Clause 8, and Annex C of [ISO89] the INTERFACE-DEFINITION shall be handled like a SPECIFICATION.

NOTE — Syntactically a MODULE-BODY-DECLARATION is a specialized MODULE-BODY-DEFINITION.

The IDENTIFIER of the INTERFACE-DEFINITION defines the *interface name*. It is a matter of the interpreting context to uniquely map any interface name (INTERFACE-IDENTIFIER) to a syntactically valid INTERFACE-DEFINITION that has the given name, or to the unique token "⊥". The interface name shall have no further significance within the INTERFACE-DEFINITION.

NOTE — A UNIX environment could use a file-naming convention and a search-path to implement this mapping.

A.1.3 Informal Semantics

An INTERFACE-DEFINITION is a container for the *declaration* of a set of open systems together with all necessary underlying definitions[21] as described above. It is one of the two additional start symbols of the Open Estelle grammar and does not appear on the right-hand side of any production. An INTERFACE-DEFINITION is intended to be represented inside a separate textual unit (*interface-textual-unit*).

NOTE — An INTERFACE-DEFINITION can import other INTERFACE-DEFINITIONs and apply their definitions.

Every MODULE-BODY-DECLARATION contained in an INTERFACE-DEFINITION *declares* an open system. In doing so, it defines the external interface of the open system by referring to a MODULE-HEADER-DEFINITION, which describes a set of typed interaction-points and exported variables.

The *definition* of an open system is given in a BEHAVIOUR-DEFINITION inside a separate textual unit (see Appendix A.2). The pair of a BEHAVIOUR-DEFINITION and its referred INTERFACE-DEFINITION completely defines the declared set of open systems.

NOTE — An INTERFACE-DEFINITION may be referred by any number of BEHAVIOUR-DEFINITIONs.

NOTE — Analogous to Standard Estelle the appearance of the keyword "**EXTERNAL**" inside a MODULE-BODY-DECLARATION of an INTERFACE-DEFINITION leads to an syntactically correct, but incomplete description (see clauses 7.3.7.3 and 9 of [ISO89]). But in opposition to a SPECIFICATION or a BEHAVIOUR-DEFINITION, where only a textual modification can abolish this incompleteness, the attachment of an appropriate BEHAVIOUR-DEFINITION (see Appendix A.2) leads to a complete description of the declared open systems.

NOTE — The definitions of constants, types, channels, and module-headers are independent of any BEHAVIOUR-DEFINITION that refers the INTERFACE-DEFINITION. Therefore an INTERFACE-DEFINITION without any MODULE-BODY-DECLARATIONs is a complete description by itself.

A.2 Behaviour Definition

A.2.1 Syntax

```
BEHAVIOUR-DEFINITION =          "BEHAVIOUR" IDENTIFIER "FOR" INTERFACE-IDENTIFIER ";"
                                [ DEFAULT-OPTIONS ]
                                [ TIME-OPTIONS ]
                                [ IMPORT-OPTIONS ]
                                BEHAVIOUR-DECLARATION-PART
                                "END" "." .
BEHAVIOUR-DECLARATION-PART =    { BEHAVIOUR-DECLARATIONS } .
BEHAVIOUR-DECLARATIONS =        MODULE-BODY-DEFINITION .
INTERFACE-IDENTIFIER =          IDENTIFIER .
```

21. An interface can gain access to the definitions and declarations given in other interfaces by *importing* these interfaces (see Appendix A.3).

A.2.2 Constraints

With the exception of module attribution rules[22] (Clause 7.3.6.2 of [ISO89]), all constraints and interpretation rules for SPECIFICATION (Clause 7.2 of [ISO89]) shall also be valid for BEHAVIOUR-DEFINITION, and the ones for DECLARATIONS (Clause 7.3 of [ISO89]) shall also be valid for BEHAVIOUR-DECLARATIONS. Applying scope rules of Clause 7, Clause 8, and Annex C of [ISO89], the BEHAVIOUR-DEFINITION shall be handled like a SPECIFICATION.

The IDENTIFIER of the BEHAVIOUR-DEFINITION shall be the name of the BEHAVIOUR-DEFINITION, which shall have no significance within the BEHAVIOUR-DEFINITION.

The INTERFACE-IDENTIFIER of the BEHAVIOUR-DEFINITION shall be mapped by the interpreting context to a valid INTERFACE-DEFINITION with this name (see also Appendix A.1.2). This *referred* INTERFACE-DEFINITION is implicitly imported as if its name appeared in the IMPORT-OPTIONS (see Section A.3).

NOTE — The name of the BEHAVIOUR-DEFINITION and the name of the INTERFACE-DEFINITION it refers to may be identical. This is especially useful if there is only one BEHAVIOUR-DEFINITION for an INTERFACE-DEFINITION.

A BEHAVIOUR-DEFINITION has to directly contain exactly one *matching* MODULE-BODY-DEFINITION for each MODULE-BODY-DECLARATION within the referred INTERFACE-DEFINITION, i.e. they have the same name and refer to the same MODULE-HEADER-DEFINITION.

NOTE — For a given BEHAVIOUR-DEFINITION this leads to a 1:1 relationship between the open system *definitions* in this BEHAVIOUR-DEFINITION and the open system *declarations* in its referred INTERFACE-DEFINITION.

A.2.3 Informal Semantics

A BEHAVIOUR-DEFINITION is a container for the *definition* of a set of open systems. It is one of the two additional start symbols of the Open Estelle grammar and does not appear on the right-hand side of any production. A BEHAVIOUR-DEFINITION is intended to be represented inside a separate textual unit (*behaviour-textual-unit*).

A BEHAVIOUR-DEFINITION refers to an INTERFACE-DEFINITION, which *declares* a set of open systems. For each of these *open system declarations*, the BEHAVIOUR-DEFINITION contains exactly one *matching* MODULE-BODY-DEFINITION, which *defines* the behaviour and internal structure of the previously declared open system.

NOTE — MODULE-BODY-DEFINITIONS inside a BEHAVIOUR-DEFINITION may contain the keyword "**EXTERNAL**". Analogous to Standard Estelle such a BEHAVIOUR-DEFINITION is syntactically correct but *incomplete* (see clauses 7.3.7.3 and 9 of [ISO89]) and therefore has no formal semantics.

Declaration and definition of the open system have the same name and refer to the same MODULE-HEADER-DEFINITION. Consequently the open system definition has the same external interface as the matching open system declaration. A MODULE-BODY-DEFINITION containing a BODY-DEFINITION supplements a behaviour to the open system, which formerly was only described in terms of its external syntactical interface.

It is possible to define different open system *definitions* for the same open system *declaration* by defining several BEHAVIOUR-DEFINITIONS (in separate textual units), each of them referring to the same INTERFACE-DEFINITION (which declares the open system). Since an *importing environment* (see Appendix A.3) only imports an INTERFACE-DEFINITION, the attachment of a matching BEHAVIOUR-DEFINITION to the importing environment is a matter of the interpreting context.

---

22. Module attribution is handled in Appendix A.4.

NOTE — This 1:n relationship between the declaration of an open system and its several (behaviour) definitions allows some kind of *polymorphism* at the application of open systems.

NOTE — The formal representation of a completed system of open system definitions and importing environments could be a set of Open Estelle textual units that is (1) closed in respect to imported of INTERFACE-DEFINITIONS and (2) contains for every (incomplete) INTERFACE-DEFINITIONS exactly one matching BEHAVIOUR-DEFINITION.

NOTE — An Estelle Compiler under UNIX could use a script-file or command-line parameters to denote the BEHAVIOUR-DEFINITIONS to be linked. By default the BEHAVIOUR-DEFINITION with the same name as the imported INTERFACE-DEFINITION could be uniquely located[23] and linked (see Appendices A.1.2 and A.2.2).

A.3 Import of Interfaces

A.3.1 Syntax

IMPORT-OPTIONS =	"**IMPORT**" INTERFACE-IDENTIFIER { "**,**" INTERFACE-IDENTIFIER } "**;**" .
BODY-DEFINITION =	[IMPORT-OPTIONS] DECLARATION-PART INITIALIZATION-PART TRANSITION-DECLARATION-PART .
CONSTANT-IDENTIFIER =	IDENTIFIER | QUALIFIED-IDENTIFIER .
TYPE-IDENTIFIER =	IDENTIFIER | QUALIFIED-IDENTIFIER .
CHANNEL-IDENTIFIER =	IDENTIFIER | QUALIFIED-IDENTIFIER .
HEADER-IDENTIFIER =	IDENTIFIER | QUALIFIED-IDENTIFIER .
BODY-IDENTIFIER =	IDENTIFIER | QUALIFIED-IDENTIFIER .
QUALIFIED-IDENTIFIER =	INTERFACE-IDENTIFIER "**::**" IDENTIFIER .
INTERFACE-IDENTIFIER =	IDENTIFIER .

A.3.2 Constraints

Any INTERFACE-IDENTIFIER of the IMPORT-OPTIONS shall be mapped by the interpreting context to a valid INTERFACE-DEFINITION with this name (see also Appendix A.1.2). All of these INTERFACE-DEFINITIONs are *imported* into the closest containing SPECIFICATION, MODULE-BODY-DEFINITION, INTERFACE-DEFINITION or BEHAVIOUR-DEFINITION. (We will refer to these as the "*importing environment*s").

The *import* of an INTERFACE-DEFINITION defines a *qualified visibility* in the importing environment for all IDENTIFIERs that have a defining-point whose region is the imported INTERFACE-DEFINITION.

To define a *qualified visibility* for an IDENTIFIER means that applying the scope rules of Clause 7, Clause 8, and Annex C of [ISO89] the QUALIFIED-IDENTIFIER for this IDENTIFIER is handled like an identifier that has a defining-point whose region is the importing environment. Applying compatibility rules (e.g. type- or assignment-compatibility), a reference to an imported QUALIFIED-IDENTIFIER refers to its native definition (inside a INTERFACE-DEFINITION).

NOTE — The import of items from INTERFACE-DEFINITIONs into importing environments defines the *imports-relation* over the set of valid Open Estelle textual units.

NOTE — The *imports-relation* is not transitive: If INTERFACE-DEFINITION I_1 is the native defining point of IDENTIFIER n, INTERFACE-DEFINITION I_2 imports INTERFACE-DEFINITION I_1, and an different importing environment E imports I_2 but not I_1, then n is qualified visible in I_2, but not in E.

NOTE — A necessary condition for the syntactical correctness of an importing environment is the syntactical correctness of all imported INTERFACE-DEFINITIONs. The foundation of this requirement forbids any direct or indirect recursion of imports between INTERFACE-DEFINITIONs, i.e. the transitive closure of the *imports-relation* shall be irreflexive.

---

23. A UNIX environment could use a file-naming convention and a search-path to implement this mapping.

QUALIFIED-IDENTIFIERs are only valid for *applied occurrences* of imported identifiers (see Clause 7.1.2.8 of [ISO89]). For an IDENTIFIER that has a defining-point whose region is an INTERFACE-DEFINITION, the following shall be valid: (1) the INTERFACE-IDENTIFIER of the QUALIFIED-IDENTIFIER shall be the name of the INTERFACE-DEFINITION and (2) the IDENTIFIER of the QUALIFIED-IDENTIFIER shall be the native IDENTIFIER.

NOTE — The names of SPECIFICATIONs, INTERFACE-DEFINITIONs and BEHAVIOUR-DEFINITIONs have no defining-point in the sense of clause 7, clause 8, and annex C of [ISO89].

NOTE — A QUALIFIED-IDENTIFIER *globally* identifies at most one unique item, because every INTERFACE-IDENTIFIER is uniquely mapped to at most one valid INTERFACE-DEFINITION (see Appendix A.1.2) and every IDENTIFIER that has a native defining-point whose region is the INTERFACE-DEFINITION, identifies a unique item inside the INTERFACE-DEFINITION (see Clause 7.1.2 of [ISO89]).

A.3.3 Informal Semantics

The import of an INTERFACE-DEFINITION gains the importing environment access to its definitions and declarations. This includes the MODULE-BODY-DECLARATIONs contained in the INTERFACE-DEFINITION, each of them declaring an open system. These can be handled like normal incompletely defined MODULE-BODY-DEFINITIONs in Standard Estelle, i.e. several instances of them can be created with INIT-STATEMENTS and can be further handled like usual module instances.

In contrast to Standard Estelle, an importing environment incorporating an imported open system declaration (which syntactically is an incompletely defined MODULE-BODY-DEFINITION) can be completed without any textual modifications, if for every imported INTERFACE-DEFINITION an appropriate BEHAVIOUR-DEFINITION is attached. In this case the instantiation of the open system (declared in an INTERFACE-DEFINITION) leads to the instantiation of the MODULE-BODY-DEFINITION given inside the appropriate BEHAVIOUR-DEFINITION.

NOTE — An open system definition *formally describes* an open system. The open system itself is an *instance* of this description (see also Clause 7.2.4 of [ISO89]). Consequently an importing environment may create several open systems of an open system definition by creating several module instances.

NOTE — It is possible to import the same INTERFACE-DEFINITION into several BODY-DEFINITIONS of an importing environment simultaneously. Consequently several instances of the same open system can be part of the module instance tree of an specification instance at independent positions.

A.4 Module Attribution Rules

The following module attribution and nesting rules shall replace the rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO89]:

1. Each active module shall be attributed.
2. A module[24] that is not attributed or attributed "**SYSTEMACTIVITY**" or "**SYSTEMPROCESS**" shall be a SPECIFICATION or be directly contained in an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or a not attributed module.
3. A module that is attributed "**ACTIVITY**" shall be directly contained inside an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or an attributed module.
4. A module that is attributed "**PROCESS**" shall be directly contained inside an INTERFACE-DEFINITION, a BEHAVIOUR-DEFINITION or a module that is attributed "**PROCESS**" or "**SYSTEMPROCESS**".
5. Within an importing environment that is a not attributed module, any imported MODULE-BODY-DECLARATION that is attributed "**PROCESS**" or "**ACTIVITY**" is handled like it was

24. With the term "module" we refer to a SPECIFICATION or a MODULE-BODY-DEFINITION.

attributed "**SYSTEMPROCESS**" (in case of "**PROCESS**") or "**SYSTEMACTIVITY**" (in case of "**ACTIVITY**").

6. If the BODY-IDENTIFIER of an INIT-STATEMENT denotes an imported MODULE-BODY-DECLARATION, this MODULE-BODY-DECLARATION has to be attributed in such a manner that the module closest containing the INIT-STATEMENT could contain a child module with this attribution (according to the preceding attribution rules).

NOTE — All rules can be validated statically.

NOTE — This rules relax the restrictions to module attribution given in Clauses 5.2.1 and 7.3.6.2 of [ISO89]: They do not limit the possible attributions of MODULE-HEADER-DEFINITIONS but directly limit the possible attributions of nested modules and the possible instantiation of imported modules. The resulting restrictions to the attribution of the *static module nesting* (1-4) are identical to the ones in [ISO89].

NOTE — The module attribution rules (1-6) lead to the same restrictions to the attribution of the *dynamic module-instance tree* like the rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO89].

NOTE — There are no restrictions to the attribution of MODULE-BODY-DECLARATIONs or MODULE-BODY-DEFINITIONs that are directly contained in an INTERFACE-DEFINITION or a BEHAVIOUR-DEFINITION. Moreover there are no restrictions to the attribution of MODULE-HEADER-DEFINITIONs or MODULE-BODY-DECLARATIONs imported into an arbitrarily attributed module.

NOTE — Only the attribution of those imported MODULE-BODY-DECLARATIONs that are refered by an INIT-STATEMENT in the importing environment are restricted by module attribution rules. This allows the definition of interface-definitions that contain arbitrarily heterogeneous attributed MODULE-BODY-DECLARATIONs, which can be imported into any importing environment. The rules only restrict which of the imported MODULE-BODY-DECLARATIONs can be instantiated.

NOTE — Because of (5), an open system that is attributed "**ACTIVITY**" can be imported and applicated (i.e. instantiated) by any (arbitrarily attributed) module.