Open Estelle —
An FDT for Open Distributed Systems [1]

J. Thees, R. Gotzhein
University of Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
e-mail: {thees, gotzhein}@informatik.uni-kl.de

Abstract

An Estelle specification describes a system of communicating components (module instances). The specified system is *closed*, 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 into an (Estelle) environment.

To overcome this restriction, we introduce a compatible extension of Estelle, called '*Open Estelle*'. It allows one to specify *open systems* (i.e. systems that have the ability to communicate with any environment through a well-defined external interface), as well as their *formal incorporation* into different environments. We define a *formal syntax* and a *formal semantics* for Open Estelle, both based on and extending the syntax and semantics of Estelle. Furthermore, we present a set of tools, including a compiler for the automatic generation of implementations directly from Open Estelle specifications.

Keywords

FDTs, Estelle, extensions of FDTs, semantic model, FDT-based implementation

1. Introduction

Estelle [ISO97, DeBu89] is a formal description technique (FDT), internationally standardized since 1989. An Estelle specification describes a 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*, i.e. they have no abilities for external communication. Because of this restriction, *open systems* can be described *formally* only together with and embedded into a *concrete environment*[2] such that the resulting system is again closed in the described sense. This means

1. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Go 503/4-1.

that the environment has to be known in advance, and that it must already be determined when the open system is specified.

In [GoRoTh96], pragmatic approaches to the specification of systems with the ability to interact with pre-existing real-world environments are studied. 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[3]. Also, the *incorporation* of systems specified in such a manner into a formal context (e.g. another Estelle specification) is not supported by these approaches.

The formal description technique *SDL* [ITU94] supports the specification of open systems syntactically, since it is possible to connect channels to the system boundary. Also, open systems are assigned a formal semantics in terms of a set of communicating Meta-IV processes. However, communications are purely internal, because signal exchange with the environment is not expressed. Furthermore, the composition of open system specifications is neither supported syntactically nor semantically.

The ability to *formally* specify open systems in Estelle independently of any environment, and to specify their incorporation into different concrete environments, has the following benefits:

- The *expressiveness* and the *abstraction level* of Estelle is enhanced. For instance, protocol machines can be formally described and analyzed *independently* of concrete user modules or network modules. Their formal semantics takes all possible environment behavior into account, and is not restricted to a single, explicitly specified environment.
- The *decomposition of large systems* into sets of components is supported. This improves the practical development of systems with Estelle, as it allows one to specify and analyze system components separately, and to compose them afterwards. Also, *component reuse* is supported, because components are specified independently of their environment.
- The formal description of an open system can serve as a basis to generate an *implementation* with the ability to communicate with pre-existing environments. Unlike the pragmatic approaches mentioned before, the Estelle specification on which the implementation is based has a formal semantics.

In this paper, we introduce a syntactic and semantic extension of Estelle called '*Open Estelle*'. It makes it possible to specify open systems (i.e. systems that have the ability to interact with any environment) through a well-defined external interface, as well as their incorporation into different environments. In Section 2, we explain the basic concepts of Open Estelle. Section 3 introduces the syntactic language elements of Open

2. An environment specified manually or generated with tools like the *'Universal Test Drivers Generator'* (UTDG, [LLM91,INT97]); see Sections 2.2 and 2.4.
3. Primitive functions and procedures (and also the specification containing them) have no 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 [ISO97]). Such a definition may, however, be difficult to obtain for communication with some environment realized, for instance, via primitive functions.

Estelle. In Section 4, we define a formal semantics for Open Estelle, which is based on the Standard[4] Estelle semantics. Section 5 addresses implementation issues, including tool support for Open Estelle.

2. Basic Concepts of Open Estelle

The main objective of Open Estelle is to provide language support for the formal description of open systems and their incorporation into different environments. This means that an open system not only has a well defined syntactic and semantic interpretation *independently of its environment*, but also *as part of a compound system*. In the following, we introduce some basic concepts of the syntactic and semantic interpretation of open systems, and their incorporation in Open Estelle.

2.1. Representation of Open Systems

A basic design decision of Open Estelle concerns the representation of an open system. Standard Estelle already possesses an abstraction to describe encapsulated systems with well-defined external interfaces and their aggregation into more complex systems: *modules*. Module instances could in principle supply a suitable representation of open systems in Estelle: the module header describes the external interface of the module instance without regard to its internal description, the module body describes the module instance behavior.

Note that a module definition *describes* an open system, while a module instance *is* an open system. At any point in time, a module instance (i.e. an open system) is surrounded by a collection of other module instances. We call this collection of other module instances *environment*, and we say that the open system is *incorporated* into an environment when placing it into a collection of surrounding module instances.

Obviously, modules are a suitable and natural means to describe open systems, and in fact Open Estelle basically describes open systems as Standard Estelle modules. However, in Standard Estelle, a module can only be defined and used inside the specification of a closed system. Furthermore, being a fragment of a specification, it has no formal semantics. Therefore, based on Standard Estelle, the following aspects have to be addressed:

- syntactic extensions to define modules that describe open systems independently of their environment,
- a formal semantics for open systems independent of their environment, and also as part of a compound system,
- a mechanism to allow other specifications (e.g. specifications that define a specific environment) to import the definitions of open systems.

---

4. By *Standard Estelle*, we refer to the Estelle language defined in [ISO97], whereas the proposed extension is called *Open Estelle*.

2.2. Formal Import vs. Textual Inclusion

As noted above, the formal description of an open system must have a *unique syntactic and semantic interpretation* for the open system itself, as well as for its incorporation into all potential environments. While the semantic interpretation of an open system (its *'behavior'*) depends on the possible interactions with all environments, its syntactic interpretation should be independent of the incorporating context. In particular, internal definitions of any potential environment incorporating an open system should not have any impact on the interpretation of the open system or its external interface.

The following examples show that this requirement rules out solutions that are based on *simple textual inclusion* of open system specifications into environment specifications. In these examples, we assume that open systems are defined by an Estelle specification fragment[5]. We then consider the consequences of textually including (preprocessor '#include' statement) these fragments into an incorporating Estelle specification.

Example 1: *Interference between an imported interface and its importing environment.*

Let the Estelle specification fragment OS define an open system that refers to a predefined function, procedure, or type. Furthermore, let Estelle specification S redefine this identifier, e.g like 'integer' shown in the following specification fragment:

```
SPECIFICATION S;
    TYPE integer = (zero, one, two); (* predefined type integer is redefined in this scope *)
    MODULE M FOR {...};
        #include OS                (* textually include specification text fragment OS *)
    END;
END.
```

The textual inclusion of OS into module S.M causes the definitions given inside OS to be interpreted in the context of module M, i.e. with a redefined type 'integer'. Since OS refers to the identifier 'integer', it will no longer refer to the predefined type, but to the redefinition. This may have a massive impact on the syntactic and semantic interpretation of OS.

Example 2: *Interference between two imported open systems.*

Let the Estelle specification fragments OS1 and OS2 define two open systems that (by coincidence) define the same identifier (e.g. different types named PDU for different 'Protocol Data Units'). Textual inclusion of both open systems into the same context leads to a syntactically incorrect specification, since it contains an inadmissible identifier redefinition.

Obviously, the syntactic interpretation of the definitions given inside the textually included open system descriptions depends on the including context and the combina-

5. e.g., a module definition together with all necessary constant, type, channel and module header definitions

tion of included open system descriptions. This makes the syntactic definition of these open systems *non-formal*. To obtain a *formal* definition of an open system, it has to be interpreted strictly independently of any environment that makes use of it; in particular, the name-spaces of different open systems and importing environments must be separated. In Section 3, we will see that Open Estelle avoids these problems, since

a) the syntactic definitions given inside the description of an open system are *interpreted in a unique way*, independently of a concrete importing environment (see Section 3.1 and 3.2);
b) the definitions given inside an open system are not included textually, but they are *imported* in a well-defined way; this formal import only grants access to the (unique) definitions of the open system (see Section 3.3);
c) identifiers imported from an open system description are *qualified* by the name of the open system (e.g. '**OS1::PDU**' and '**OS2::PDU**' in Example 2); this avoids naming conflicts between different interfaces or with identifiers inside the importing environment; furthermore, it improves the readability of the specification, since the origin of each identifier is explicitly stated (see Section 3.3).

2.3. Separation between Declaration and Definition of Open Systems

In Open Estelle, the description of an open system consists of the *external interface* and the *internal description*, which are textually separated. 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 (see Figure 1).

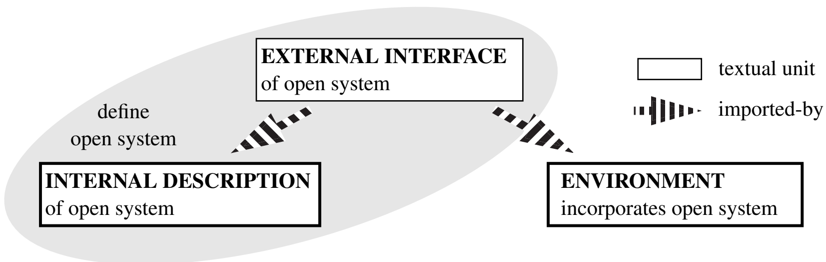

Figure 1. Independent Descriptions for Open System and Environment.

An 'external interface' *declares* open systems (i.e. module body declarations) together with all necessary underlying definitions (i.e. module headers, channels, types and constants; see Section 3.1).[6] An 'internal description' *defines* these open systems by adding an internal description (see Section 3.2). This separation makes it possible to develop and compile open systems and importing environments independently, as soon as their common interface has been defined. Furthermore, it supports the separation of concerns between open systems and their possible environments.

6. External interfaces may import other external interfaces and make use of their definitions (Section 3.3).

The textual separation between the external interface and the internal description of a system is similar to the concepts found in Modula-2 [Wir85]. Also, the distinction between a declaration and a definition of a syntactic object is similar to Modula-2, C or C++: a *declaration* defines name and type (i.e. the external interface) of an object, whereas a *definition* adds internal details.[7]

2.4. Semantics of Open Systems

Open Estelle has been designed to formally describe open systems and their incorporation into any environment. Consequently, an open system should have a formal semantics independent of an environment, and also as part of a compound system.

The *incorporation* of an open system into an Estelle environment is already semantically covered by the standard Estelle semantics, since the instantiation of an imported open system (i.e. of an Estelle module) creates the same module structure as the instantiation of a normal (textually included) local child module. This produces an intuitive semantics of the application of open systems. Even more, this semantics makes it possible to fuse the description of an importing environment and its imported open systems into one single specification that is semantically equivalent. If textual interferences (see Section 2.2) are excluded (e.g. by appropriate renaming of identifiers defined in open systems and their importing environment), the import of an open system can be replaced essentially by a semantically equivalent textually included child module definition. In Section 5.1 we will present a tool that implements this fusion.

An important aspect of Open Estelle is the definition of a semantics for an open systems *independent of its environment*. In the special case of an open system *without* any external interaction points and exported variables, its semantics is identical to the semantics of a corresponding (closed) Estelle specification, independent of any concrete environment, since there are no means for interactions with an environment. But if the open system *has* external interaction points or exported variables, a concrete environment can interact by modifying exported variables and/or sending messages through external interaction points. Since the semantics of an open system can not refer to the specific restrictions of a concrete environment, every possible influence through the external interface of the open system has to be considered, i.e. every possible sequence of messages received through external interaction points and every possible modification of exported variables[8]. If the open system is incorporated into a concrete environment, this maximum set of possible interaction is reduced to a subset of interactions with this particular environment. In Section 4, we will describe this concept in detail.

---

7. E.g. in C 'int succ(int n);' is a *function declaration* and 'int succ(int n) {return n+1;}' its *function definition*.

8. This kind of maximum interaction is also the foundation of the *'Universal Test Drivers Generator'* (UTDG, [LLM91,INT97]). UTDG generates modules that can send or accept any interaction through their external interaction points or execute any modification of their external variables. It was developed to simulate unspecified system components during tests of Estelle specifications. But since it does not solve the syntactic problems presented in Section 2.2, it is not directly suitable for the formal specification of open systems.

3. The Syntax of Open Estelle

In this section, we introduce the syntactic extensions of Estelle for the description of the external interface of open systems (Section 3.1), the internal description of open systems (Section 3.2), and the incorporation of open systems into different environments (Section 3.3) through examples. These extensions are based on the syntactic concept of modules. The formal definition of the Open Estelle syntax is given in [ThGo97a].

3.1. 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*[9]. 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 of 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 3.3). 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 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 BEHAVIOR-DEFINITION, where only a textual modification can remove this incom-

9. Since the Estelle standard does not state a representation for a specification text, we use the term '*textual unit*' for self-contained syntactic objects (e.g. a specification). In a UNIX environment such a textual unit is usually represented as an ASCII-text file.

pleteness, the attachment[10] of an appropriate BEHAVIOR-DEFINITION (see Section 3.2) leads to a complete description of the declared open systems.

The reader should note that the interface definition itself describes no behavior (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 3.3). 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 supports a clean separation of concerns between open systems that refer to common definitions.

3.2. Internal Description of Open Systems

The internal description of an open system is given by a MODULE-BODY-DEFINITION inside of a BEHAVIOR-DEFINITION residing in a *separate textual unit*. As suggested by the term BEHAVIOR-DEFINITION, from an abstract point of view the internal description of an open system only determines the *behavior* of the open system, because the syntactic interface is already determined by the interface definition.

The BEHAVIOR-DEFINITION is the second new start non-terminal of the Open Estelle syntax. It begins with the new keyword '**BEHAVIOR**', followed by its name and a reference to its interface (see Figure 3). The definitions of its interface are imported implicitly (see Section 3.3).

```
BEHAVIOR 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 Behavior-Definition (see Figure 2).

A BEHAVIOR-DEFINITION is a container for a set of open system definitions: for every MODULE-BODY-DECLARATION of its interface, it contains exactly one *matching*[11] 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 sub-structured into child modules and it can even import and use other open systems. This supports a very flexible internal structuring of open systems.

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

3.3. Incorporation of Open Systems into Open Estelle Environments

To be able to incorporate[12] 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 inside either a specification, a module-body, an interface, or a behavior-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.[13]

```
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).

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.

---

12. 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 inclusion* of the description of the open system into the description of an Open Estelle environment.
13. 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).

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, which can be used in different specifications, behavior 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. Thus it supports the specification of a system of interfaces that exactly models the type dependencies of the global system.

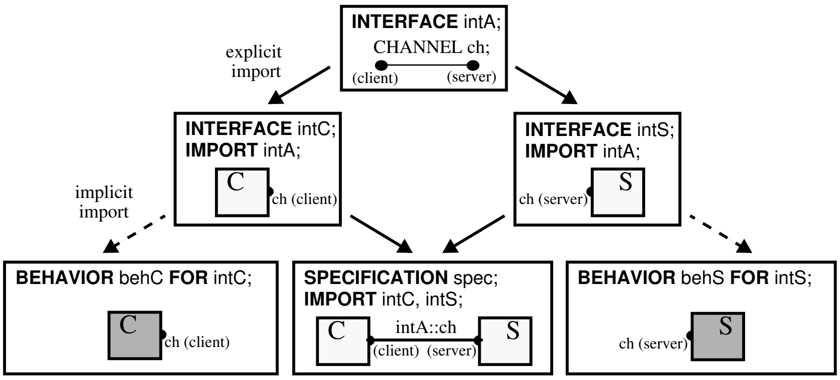

Figure 5. Multiple interfaces with common definitions.

3.4. Module Attribution

The module attribution rules given in Clauses 5.2.1 and 7.3.6.2 of [ISO97] are not directly suitable for the application in Open Estelle, since they only relate to the textual nesting of modules. Therefore, in [ThGo97a] we introduce extended module attribution rules, which imply the same attribution scheme for the dynamic module instance tree as the original rules. It is possible to specify both, open systems that contain one or more subsystems[14] (which can be incorporated into un-attributed 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*.

4. 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 models the

14. With the term *'subsystem'* we refer to instances of system modules.

interaction between an open system and an environment. More precisely, the semantics captures all potential behavior of an open system in *all possible* environments; this potential behavior is reduced when the open system is incorporated into a specific environment.

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

4.1. Formal definition of the Open Estelle semantics

According to the Estelle standard [ISO97], 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, [ISO97]):
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 $A_i = \varnothing$, 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 $A_i \neq \varnothing$, 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 non-determinism 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 $\text{transmission}_P(\text{gid'}_{SP})$, where $\text{gid'}_{SP} \in [t]_P(\text{gid}_{SP})$ (see Clause 9.5.4, [ISO97]). 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 into 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. 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[15]. 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 ($\text{receive}_P(\text{ip})$, see Clauses 9.3.1 and 9.4.3, [ISO97]), as well as the set of values that may be assigned to an exported variable e [16], 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, $\text{sit}_P = (\text{gid}_P; A_1,...,A_n)$, 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 $AS(\text{gid}_P/S_i) \in AS^*(\text{gid}_P/S_i)$: $(\text{gid}_P; A_1,...,AS(\text{gid}_P/S_i),...,A_n)$ is a next global situation of sit_P.
(b) For every i = 1,...,n: if $A_i \neq \emptyset$, then for every $t \in A_i$: $(t(\text{gid}_P); A_1,...,A_i\backslash\{t\},...,A_n)$ is a next global situation of sit_P.
(c) For every $\text{gid}_P' \in \text{env\_mod}^+(\text{gid}_P)$: $(\text{gid}_P'; A_1,...,A_n)$ is a next global situation of sit_P. env_mod is defined as follows[17]:

---

15. 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.
16. We assume that the type of e is well-defined, i.e. its declaration does not include type-identifiers associated with the '...' construct.

(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 $\in$ $EV\text{-}id_M$, 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 non-determinism of each component 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, [ISO97]). 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, [ISO97]). 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, [ISO97]) that the instance P can receive through interaction point ip (Clause 9.4.3, [ISO97]). $received_P(gid_P)$ 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, [ISO97]). 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, [ISO97]).

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[18], 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

17. The function env_mod: $EIP_P \rightarrow \wp(EIP_P)$ defines a relation $env\_mod_R=\{(x,y) \in EIP_P \times EIP_P \mid y \in env\_mod(x)\}$. Let $env\_mod_R^+$ be the transitive closure of env_mod_R. Then the function **env_mod$^+$**: $EIP_P \rightarrow \wp(EIP_P)$, $x \rightarrow \{y \in EIP_P \mid (x,y) \in env\_mod_R^+\}$ defines a *transitive closure* of function env_mod, i.e. the result of one or more consecutive events as described in (c1) and/or (c2).

18. 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.

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 behavior 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 behavior 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 [ISO97]. 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 outputs 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, [ISO97]). 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, [ISO97]). 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, [ISO97]).

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[19]. 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 [ISO97] (see Clauses 9.5.3 an 9.5.4), the definitions of the Standard Estelle semantics can be used[20].

---

19. As already noted above, we do not consider the effects of connecting two external interaction points of the same open system in this paper.

20. 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, [ISO97]).

4.2. Completeness and Soundness of the Open Estelle Semantics

As already noted in Section 2.4, the semantics of the *incorporation* of an open system into an Estelle environment is reduced to the Standard Estelle semantics by construction, since the instantiation of an imported open system (i.e. of an Estelle module) creates the same module structure as the instantiation of a normal (textually included) local child module. It remains to show that the semantics of an open system on its own as defined in Section 4.2 is correct and sound.

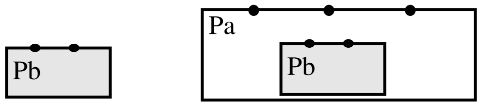

Figure 6. Open systems Pa and Pb.

If we consider the semantics of open system Pb on its own in comparison to its incorporation into an (open or closed) environment Pa (see Figure 6), module Pa partially (if open itself) or completely (if closed) determines the environment of Pb. *Completeness* of the Open Estelle semantics means that the potential computations of Pa, when 'projected onto' the incorporated Pb, are a subset of the potential computations of Pb on its own. In other words, the computations of the open system Pb are reduced when it is incorporated into some environment. *Soundness* means that only potential computations are defined for Pb that are possible in some environment Pa. The proof of these properties is addressed in [ThGo97a].

5. Implementation Issues

Estelle specifications do not only describe systems formally, they can also serve as a basis for an automated implementation. Several tools for the automatic implementation of Estelle specifications are currently available (e.g. EDT [Bud92], PET/DINGO [SiSt93]). 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 establishes 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 interfaces* and with a *well-defined semantics*. This opens the possibility for a direct practical incorporation of formally specified open systems into real-world environments.

5.1. 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 behavior-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 the *eXperimental Estelle Compiler* (*XEC*, [ThGo97b, ThGo98, The98]), a completely new optimizing code generator for Estelle and Open Estelle, which creates C++-code for open systems and importing environments independently of one another (see Figure 7). 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 incorporated into and communicating with real-world environments.

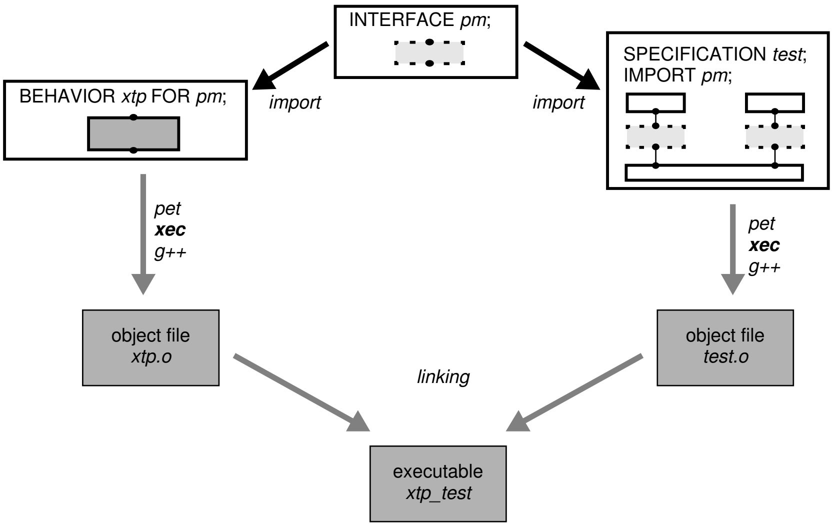

Figure 7. Independent compilation of open systems and a compatible environments.

Another Open Estelle tool supports the *textual embedding* of a set of open systems into an Open Estelle environment that incorporates them (see Section 2). 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). The tool enables the application of existing formal methods for Standard Estelle to systems consisting of a set of Open Estelle descriptions.

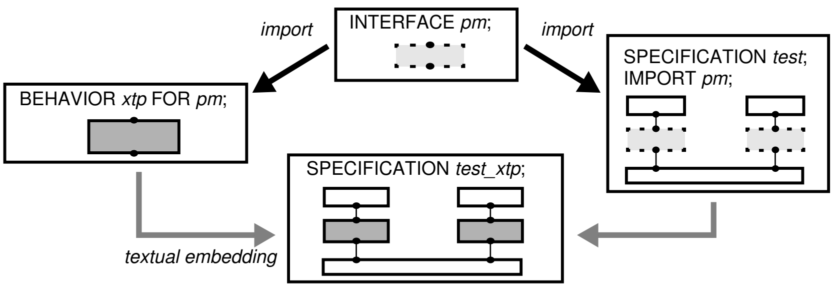

Figure 8. Embedding of an open system into a compatible environment.

5.2. Case Study

As a case study for the practical use of Open Estelle and of the tools presented above, we have split an existing, relatively large Estelle specification (more than 7500 lines of specification text) of the *Xpress Transport Protocol* (XTP, [XTP95]) into a set of open systems specified in Open Estelle. We have incorporated the XTP protocol machines into different Open Estelle and real-world environments, and with different client and network implementations without having to re-compile the generated machine object file (see Figure 7).

In particular, it is not necessary to re-compile the whole system after a modification that only affects parts of the system. In our case study the relatively complex XTP protocol machine was specified as open system, independently of the application context (e.g. specification 'test' in Figure 7) that import it. After a modification of the application context (including all user and network modules), only the context had to be re-compiled. This reduced the turnaround-times significantly: the re-compilation and linking of specification 'test' took less than 10 seconds[21] (1.0 seconds PET/XEC, 5.8 seconds g++, and 3.0 seconds linking). This is an effective **speedup of factor 13** compared to the otherwise necessary complete re-compilation of the original Standard Estelle specification with the textually embedded XTP protocol machine (17.1 seconds PET/XEC, 110.0 seconds g++, and 4.6 seconds linking).

The case study demonstrated some of the benefits of Open Estelle for the specification of large systems: one single compilation of the relatively complex protocol machine 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 turnaround times after local modifications of components of the protocol machine and clarified the type dependences between the components. In particular, it led to a clean separation between private definitions of a module and definitions that are exported to the former child modules.

6. Summary and Outlook

In this paper, we have introduced an extension of Estelle, called '*Open Estelle*'. It allows one to specify *open systems* and their *formal incorporation* into different environments. 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 the new code generator *XEC (eXperimental Estelle Compiler)* that creates efficient implementations of open systems, which can be incorporated into different Open Estelle and hand-crafted

21. Platform: Sun SPARCstation 20, Solaris v2.5, PET v2.0, XEC v1.14, g++ v2.8 (without optimizations)

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 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-behavior, which may serve as a basis for a correctness notion. We plan to investigate this issue in more detail.

7. Acknowledgments

The case study discussed in Section 5.2 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).

8. References

[Bud92] Budkowski, S.: *Estelle Development Toolset*, Computer Networks and ISDN Systems, Vol. 25, No.1, 1992

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

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

[ISO97] ISO/TC97/SC21: *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21, IS 9074, 1997

[INT97] INT: *Universal Test Drivers Generator — User Reference Manual,* Estelle Development Toolset 4.1, Institut National des Telecommunications, France, 1997

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

[LLM91] E. Lallet, Ch. A. Lebrun, J.-F. Martin, *Un outils de génération automatique de l'environnement d'exécution de specification Estelle*, Acte du 'Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'91)', Pau, 17-19 Sept. 1991

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

[The98] J. Thees: *The eXperimental Estelle Compiler — User Manual*, Univ. of Kaiserslautern, Germany, 1998, (see also '**http://rn.informatik.uni-kl.de/activities/Estelle/XEC.html**')

[ThGo97a] J. Thees, R. Gotzhein: *A Formal Syntax and a Formal Semantics for Open Estelle*, Internal Report No. 292/97, Dept. of Comp. Sci., Univ. of Kaiserslautern, Germany, 1997

[ThGo97b] J. Thees, R. Gotzhein: *Generation of Efficient Protocol Implementations - an Experimental Code Generator for Estelle and its Application to XTP,* 6th Open Workshop on 'High Speed Networks', Stuttgart, October 8-9, 1997

[ThGo98] J. Thees, R. Gotzhein: *The eXperimental Estelle Compiler - Automatic Generation of Implementations from Formal Specifications*, Second Workshop on 'Formal Methods in Software Practice (FMSP'98)', Clearwater, USA, 1998

[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