Towards "Open Estelle"

R. Gotzhein, F. Rößler, J. Thees

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

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

Abstract

In the development of reliable software systems, the use of Formal Description Techniques (FDTs) is indispensable. An FDT especially designed for the specification of communication systems is Estelle. A weakness of Estelle is the fact that it is not possible to formally specify non-closed systems, i.e. systems embedded into and interacting with some environment that either preexists or shall be developed without the use of Estelle. In this paper, measures to overcome this problem are identified. To begin with, pragmatical approaches using the features of standard Estelle to describe and embed non-closed systems are investigated. The experiences of this study lead to a compatible language extension, a first step towards "Open Estelle". Finally, the semantical implications of this extension are addressed.

1 Motivation

Formal Description Techniques (FDTs) have gained more and more importance in the development of correct software in general and of communication systems in particular. The industrialization of some FDTs is well documented by the availability of expensive commercial tool environments. Usually, these tools support the creation, maintenance, simulation, and validation of formal specifications as well as the automatic generation of executable code directly from the specification.

Among the FDTs used in the communications field is Estelle ([DeBu89]). Estelle has been especially designed for the formal specification of communication protocols, and is internationally standardized since 1989 ([ISO89]). An Estelle specification describes a hierarchically structured system of indeterministic, sequential components called *module instances*. Interaction between module instances takes place asynchronously. System structure and connection structure can be dynamically modified.

A weakness of Estelle is that Estelle specifications describe self-contained systems ("closed worlds") ([ISO89]). This means that systems specified in Estelle do not possess interaction points to some environment into which they shall be embedded later on. As a consequence, the environment has to be part of the specification. It is syntactically possible to leave the specification incomplete by, for instance, declaring the behaviour part of a module as external. This technique may be used to incorporate a system environment without actually giving a precise specification. The drawback, however, is that incomplete Estelle specifications do *not* possess a formal semantics ([ISO89])! This has turned out to be a severe obstacle for the practical application of Estelle. Usually, only some parts of a system are specified in Estelle, while other parts, such as the basic communication system or application-specific components, either preexist or are developed without the use of Estelle. Here, Estelle can still be used to provide a suitable syntax, however, due to the lack of a formal semantics, the specification is not formal in the sense of an FDT.

In this paper, we outline measures to overcome the problems discussed in the previous paragraph. We start with a systematical survey of pragmatical approaches for using standard Estelle to describe and embed open systems in Section 2. The presented solutions still suffer from the lack of a formal semantics, and from language restrictions. In Section 3, we present a compatible language extension leading towards "Open Estelle", a language supporting the specification and composition of open[1] systems syntactically. In Section 4, the semantical implications of the extension are addressed, and a formal semantics for Open Estelle is sketched. Section 5 gives an outlook. This paper describes ongoing work.

2 Pragmatical approaches

With the availability of tools for automatic code generation, we have two system descriptions at one blow: the closed specification itself and the resulting code, which, however, implements both the embedded system and the environment. As a first consideration we therefore have to distinguish between interfacing the embedded system at the specification level and the implementation level, respectively. The former will be addressed in Sections 3 and 4 whereas this section solely focuses on the implementation aspects. Strictly speaking we propose a method for deriving an embedded implementation from a closed system specification. That makes a big difference compared with present Estelle compilers, which additionally generate an artifical environment and therefore a self-contained implementation. It will turn out that this pragmatical approach implies certain extensions of Estelle, which, however, match with the ideas of Sections 3 and 4.

2.1 Deriving an embedded implementation

As a starting point consider how a self-contained implementation generated by a conventional Estelle compiler could be transformed into the corresponding embedded one. Basically two steps are needed. Firstly the embedded system has to be *extracted*, i.e. the embedded part of the generated code has to be *located* and the remaining environment *removed*. To provide proper linkage with the real environment the resulting code must be *adapted* in the second step. In general,

[1] "open" in a topological sense

this approach is unrealistic, of course, as the embedded system could be arbitrarily linked with the environment and is therefore difficult to extract and to adapt. As an example, the environment could be nested deeply in the module hierarchy of the embedded system, which requires special synchronization of both parts.

What we are looking for are proper means to make the generated code be well-structured in an embedded system part and an environment part. This is easily achieved by a suitable specification style: if we want to partition a given Estelle specification into the embedded and the environment part, we will do this at the coarsest of the following levels: subsystem, module, transition or statement. The more coarse-grained the partition gets the clearer the interface between embedded system and environment will be in the generated code. We therefore assume that the specification can be devided at the subsystem level, which is also the only way to avoid Estelle specific synchronization of the two parts. Apart from that no further demands are to be made on the specification style. It is important to mention that by this assumption, the interface between embedded system and environment is reduced to the exchange of Estelle interactions.

So far we simplified the location phase of the transformation process sketched above by the means of a proper specification style. Further manipulation at the specification level is necessary for removing the environment. Therefore we take a closer look at the generic interface between embedded system and environment. As depicted in Figure 1, there are three possibilities to erase the environment from the specification without violation of Estelle syntax:

1. Erase the environment including the channel and the interaction points. Substitute proper primitive procedures and functions for the output-statements and the when-clauses of the embedded system.
2. Substitute a proper environment stub for the specified environment and keep the channel and the interaction points. The environment stub handles interactions to and from the environment with the help of primitive procedures and functions.[2]
3. Erase the environment, the associated interaction point, and the channel. Nothing is substituted.

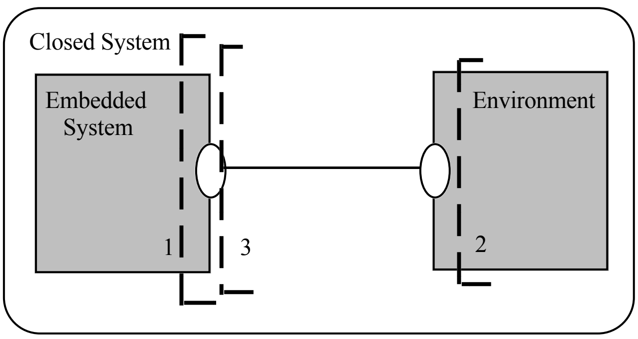

Figure 1: Generic Interface (Several channels are possible)

2 Note that the approach for distributing Estelle specifications in [LaFiVe95] could be viewed this way, because a switchboard there actually serves a similar purpose to an environment stub.

After transforming the original specification in one of the above ways, the generated code will implement the extracted embedded system. But the link to the real environment is still missing.

Depending on the selected method for removing the environment, different measures have to be taken. For cases 1 and 2, we leave the specification incomplete, as primitive procedures and functions are introduced. However, they enable the user to link routines for adaptation, which are defined in the target language of the Estelle compiler. For case 3, minor modifications of the compiler itself and the runtime system are necessary to implant the adaptation routines in the code generated for the retained interaction point. For all three alternatives it depends on the concrete environment and the used Estelle compiler how the adaptation code itself or the mechanisms to integrate it within the generated code look like.

2.2 Realization example

In [Sch96], an audio protocol was specified to generate an embedded prototype implementation as proposed above. Figure 2 shows the specifications after erasure of the environment. Note that the sender and receiver side build their own embedded system, respectively, i.e. the receiver belongs to the sender's environment and vice versa. As depicted, stubs substitute the audio devices (see Figure 1, case 2), and there is an interaction point for direct linkage with the underlying network service (see Figure 1, case 3). Actually, slightly different specifications were used in [Sch96], where the audio stubs share the same subsystem with the protocol entities and therefore the stylistic demands on the initially closed specification are not fulfilled. That's a crucial point as we actually did not intend to have a synchronization between protocol entity and audio device and are not even able to enforce it in the generated implementation. We accepted this for tuning purposes, as the selected Estelle compiler handles communication between subsystems very inefficiently, and satisfactory performance is essential for an audio application. In the following we make some remarks about the realization of the network linkage and skip the audio stubs, because there only standard facilities of the Estelle compiler were used.

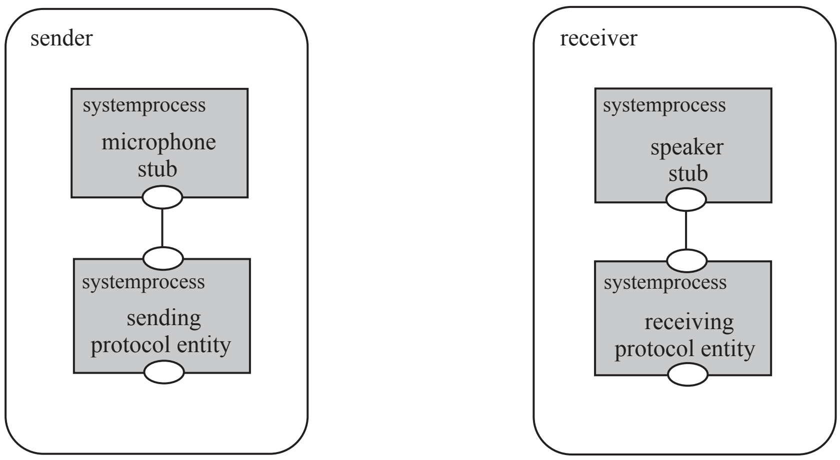

Figure 2: Specifications for extracted audio protocol entities

While UDP sockets were employed as a basic communication service, we used Pet/Dingo [SiSt93] for code generation. Pet/Dingo generates C++ code from Estelle specifications and defines separate classes for the main Estelle language elements. The ones relevant to our discussion are module instances, interaction points, and interactions. Here, new classes were derived to handle the interactions to and from the outside [Sch95]. It is worth mentioning that the actual adaptation code for translation between interactions and socket system calls is hidden inside the interaction classes, whereas the other modifications separate normal interaction processing from interactions directed to the environment. Additionally, the compiler needed some means to differentiate the newly introduced interaction points from the conventional ones. Therefore, a special naming convention for interaction points connected to UDP was introduced, as we didn't want to extend the Estelle syntax for this purpose only. Note finally that the design permits to incorporate other system environments than UDP by user provided libraries.

3 Syntactical aspects of Open Estelle

In Section 2 we presented some pragmatical approaches to the specification of open systems in Estelle. The main drawback of these pragmatical solutions from a formal point of view is the lack of a formal semantics: *solutions 1 and 2* make use of primitive functions[3] to directly access real-world resources. This hampers the derivation of a formal semantics, because the primitive functions as well as the accessed environment resources are outside the formal Estelle semantics. *Solution 3* avoids the use of language elements without formal semantics and models the external interface of the derived open system as unconnected (module-external) interaction points. Such a specification definitely has a formal semantics, however not the intended one: the formal Estelle semantics implies that such unconnected interaction points never receive interactions from outside the module, and interactions by the module via its unconnected external interaction points are discarded. So the formal semantics of such an open system is not the intended one: it has no ability to communicate through its external interaction points.

This result is not surprising, since standard Estelle has no means to specify open systems with an adequate formal semantics. To overcome this deficiency, we have to extend the expressiveness of Estelle at two levels: on a syntactical level to get a means for the formal description of open systems and on a semantical level to get the formal semantics of such an open specification. We will now discuss syntactical aspects of Open Estelle in respect to the representation and the embedding into environments. Semantical aspects are adressed in Section 4.

3.1 Formal representation of open systems

The description of open systems is no absolute new concept to standard Estelle. Estelle already includes an abstraction to describe encapsulated systems with well-defined external interfaces (thus open systems) and their aggregation into more complex systems: *modules*.

---

[3] With *'functions'* we denote *'functions and procedures'* in this section.

Estelle modules supply a good representation of open systems in Estelle. A *module header* formally describes the external interface of an open system without regard of its possible environments or its internal description. A *module body* describes the behavior of the open system. Specifying both aspects, we can build upon the syntactical and semantical definitions given in standard Estelle. Especially there is no difference between specifying the body of an open system and specifying a normal module body in standard Estelle.

A weakness of standard Estelle is that it is not possible to specify a module body stand-alone. At this point we will introduce an Estelle extension to overcome this deficiency.

It is obvious that the specification of an open system in form of a stand-alone *module body* itself is insufficient, because we also have to specify its external interface. In Estelle this has to be done by header definitions and their underlying channels, types and constants. To specify these, we need some kind of container to carry all definitions we need to formally describe the interface of the open system. We will call this container *the interface of an open system*.

To be able to separate the interface and the definition of an open system, we specify both in separated *textual units*.[4] That will make it possible to specify different open systems for the same interface. We need this to be able to compare the semantics of different open systems concerning terms like correctness or an implements-relation (see Section 4). It will also allow to specify a syntactically correct Estelle specification that embeds an open system by only referring to its interface part (see Section 3.2).

The *interface* of an open system is a textual unit that starts with the new keyword 'INTERFACE' and an identifier as a name for this interface. Then constant-, type- and channel-definitions are given. At last, we specify a module-header (that makes use of the previous definitions) and an 'external' body-definition (that is of the type of the previous header-definition). The module-header describes the interface of the open system and the 'external' body-definition declares the open system itself. The scoping rules are the same as scoping rules inside specifications. Example for an interface textual unit:

```
INTERFACE adder_interface;
    TYPE
        adder_operands = RECORD x1, x2: INTEGER; END;
        adder_result = INTEGER;
    CHANNEL adder_channel(user, provider);
        BY user: adder_request(x: adder_operands);
        BY provider: adder_reply(x: adder_result);
    MODULE adder_header ACTIVITY;
        IP toUser: adder_channel(provider) COMMON QUEUE;
        END;
    BODY adder_body FOR adder_header;
        EXTERNAL;       (* declares the open system 'adder_body' *)
    END.      (* INTERFACE adder_interface *)
```

---

[4] Since the Estelle standard does not state a representation for a specification-text, we introduce the term *textual unit* for self-contained syntactical objects (e.g. a specification).

The *definition* of an open system is a textual unit that consists of a body-definition with a qualified header-identifier. The qualification denotes the name of its interface. This interface must already include a corresponding 'external' body definition. The body-definition has the same scoping as the 'external' body definition in the interface part, i.e. it has access to all definitions of its interface. Example for a body file:

```
BODY adder_body FOR adder_header IN adder_interface;
     TRANS
          WHEN toUser.adder_request(x)
          BEGIN OUTPUT toUser.adder_reply( x.x1 + x.x2 ) END;
     END.      (* BODY adder_body *)
```

With this slight extension we are able to syntactically describe the interface of an open system and also of multiple open systems, which have this external interface. It is backward compatible with standard Estelle (except of name collisions of identifiers with the new keyword). The semantical aspects of our extension will be discussed in Section 4.

3.2 Embedding of open systems

The possibility for the formal specification of open systems is not only interesting for semantical aspects of the open system itself. The next step is to describe the embedding of a formally described open system into an environment. We will consider two kinds of environments: *Estelle environments* to describe formal aspects of the embedding and *real-world environments* to derive implementations of the open system, which may communicate with real-world resources.

To embed a formally described open system into an *Estelle environment*, we have to extend Estelle by an appropriate *import statement*, which references an interface (see specification below). This import statement (given inside a declaration-part) has the same meaning as if the definitions inside the referenced interface file were given instead of it.[5] This allows the embedding Estelle specification to access these definitions as we are used to do in standard Estelle. In particular, we can create instances of the imported open systems and can handle them like normal module instances.

Note that the importing specification only refers to the interface of the open system. This makes it possible to describe the embedding of an open system without regard to its internal definition. It allows to combine different open systems (which have the same external interface) with the same Estelle environment without modification of this environment.

By embedding open systems into importing (closed) environments we can derive the description of a *closed world*. We are currently developing a tool to derive a (self-contained) standard Estelle specification, which fuses these definitions into one. With this step it is possible to describe and investigate the usage of an open system inside an Estelle environment with all existing formal methods for standard Estelle.

[5]Except some situations with redefinitions of identifiers and name collisions.

```
SPECIFICATION adder_usage SYSTEMACTIVITY;
    IMPORT adder_interface;
    IP toAdder: adder_channel(user) COMMON QUEUE;
    MODVAR  adder_modvar: adder_header;
    (* ... *)
    INITIALIZE
        BEGIN
            INIT adder_modvar WITH adder_body;        (* create open system *)
            CONNECT adder_modvar.toUser TO toAdder;
        END;
    (* ... *)
    END.     (* SPECIFICATION adder_usage *)
```

The embedding of an implementation of a formally described open system into a *real-world environment* is similar to solution 3 of Section 2. The generation of prototype implementations of open systems can be automated by an appropriate extended Estelle compiler. At the implementation level the main benefit of our Estelle extension is the existence of a well-defined interface and of a formal (external) semantics. This provides a sound foundation for testing the conformance of an implementation.

4 On the semantics of "Open Estelle"

Estelle qualifies as a *formal* description technique, meaning that it has both a formal syntax *and* a formal semantics. Any proposed language extension should therefore cover both aspects to preserve the formality of Estelle. As we will see, the incorporation of extensions to make Estelle a suitable language for the specification of non-closed systems requires a fundamental change of the conventional, standardized Estelle semantics. In this section, we will explain why, and sketch a suitable semantics for Open Estelle.

4.1 Assessment of the standardized Estelle semantics

According to the Estelle standard ([ISO89]), the semantics of an Estelle specification 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 modelled by interleaving.

It is important to observe at this point that transition systems in their basic form have no externally visible behaviour. Instead, behaviour is defined by referring to the internal system structure. Since Estelle specifications describe closed worlds, i.e. self-contained systems having no interaction with their environment, a transition system (S, δ) is a suitable and adequate semantical model.

The discussion so far shows that it is not straightforward to incorporate the proposed language extension into the standardized Estelle semantics. Since an Open Estelle specification describes a non-closed world, i.e. a system that may be embedded into and therefore may have interaction with some environment, it will be necessary to introduce a notion of externally visible behaviour. Because this requires a different semantical model, the involved change has to be of fundamental nature.

Another important issue that is directly related to the Estelle semantics is the notion of correctness, captured by an *implements-relation*. Interestingly, no such relation has ever been proposed for Estelle in the literature. How could an implements-relation between, for instance, two Estelle specifications look like? Formally, it would have to be founded on the underlying semantical model, which amounts to the comparison of two sets of computations. It is straightforward to introduce an equivalence relation by requiring these sets be identical. However, this is not of much use, since it would enforce that both specifications have identical module and connection structure throughout their computations, and that corresponding module instances have the same local state components and transitions with identical firing conditions and effects. This leaves no room whatsoever for an implementor besides renaming, and - to a small degree - restructuring of transitions. It is not clear whether non-symmetrical implements-relations, defined in terms of reduction or extension of the set of computations and of global situations, could be defined to capture correctness. The envisaged change of the Estelle semantics may shed some new light on this issue.

4.2 Sketch of a semantics for Open Estelle

Intuitively, an Estelle specification describes a system of communicating extended finite state automata, or - more precisely - Mealy machines. A Mealy machine is given by a finite set of states S, finite input and output alphabets I and O, an initial state s_0, and a transition/output-relation $\lambda \subseteq S \times I \times O \times S$. By considering inputs and outputs only, it is possible to define the visible behaviour of a Mealy machine as the set of its input/output-sequences. Furthermore, this allows a Mealy machine to be embedded into and to define interaction with some environment. Note that this differs in a fundamental way from the standardized Estelle-semantics, which is based on a transition system (S, δ) and neither defines visible behaviour nor abstracts from internal states (see Section 4.1).

As a consequence of the previous discussion, the question arises whether the intuitive conception of the Estelle semantics based on Mealy machines, which apparently provides a better basis for defining visible behaviour and openness, can be carried into a formal semantics for Open Estelle. As in the case of Mealy machines, this semantics should consist of a set of input/output-sequences, augmented by interaction parameter values, such that only interactions occurring at external interaction points (see Section 3) are considered visible. Each element of an input/output-sequence corresponds to one transition, and may contain zero or one input interaction and zero or more output interactions, each qualified by the identifier of the external interaction point. Elements with no visible interactions can be removed from the sequence.

A complication arises from the fact that a specification now defines an open system, which means that the system environment may not be known. Consequently, all possible system environments have to be covered by the formal semantics. This may be achieved by considering all possible input scenarios at all points of execution, which will yield all possible input/output-sequences. This set will be reduced when the system environment is determined.

An abstract semantics as sketched in this section has been proposed in [GoBo95] for single module instances without descendants, specified in standard Estelle. In that work, an Estelle specification of a closed system is taken as starting point. By means of projection, an abstract semantics consisting of input/output sequences of a single module instance is derived from the set of computations. Thus, it was possible to avoid a complete development of an abstract Estelle semantics from scratch. We expect that the results obtained in [GoBo95] can be generalized to systems specified in Open Estelle. Also, it has to be investigated how well this semantics supports the composition of specifications in Open Estelle.

5 Outlook

The work reported in this paper has led to substantial insight into the problems that need to be solved in order to make Estelle an FDT for the specification of *open* systems. While a number of pragmatical results are already available and a compatible extension of Estelle has been conceived both from a syntactical and semantical point of view, more efforts are required on the way towards "Open Estelle". In particular, implementation support for the embedding into real environments should be provided. Furthermore, the syntactical and semantical aspects of composing open systems specified in "Open Estelle" should be studied in depth. Also, the work presented here may shed some new light on formal notions of correctness related to Estelle.

References

[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

[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, pp. 21-36, 1995

[ISO89] ISO/TC97/SC21: Estelle - A Formal Description Technique Based on an Extended State Transition Model, ISO/TC97/SC21, IS 9074, 1989, 179p

[Koh78] Kohavi, Z.: Switching and Finite Automata Theory, McGraw Hill, USA, 1978

[LaFiVe95] Lallet, E., Fischer, S., Verdier, J.-F.: A new approach for distributing Estelle specifications, in: 8th International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, October 17.-20., 1995

[Sch95] Schaible, P.: Einbettung automatisch generierter Protokollimplementierungen in eine reale Netzumgebung, Projektarbeit, Department of Computer Science, University of Kaiserslautern, 1995

[Sch96] Schulz, A.: Spezifikation und (halb-) automatische Implementierung einer Audioanwendung über Ethernet, Projektarbeit, Department of Computer Science, University of Kaiserslautern, 1996

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