Design of a Formal Estelle Semantics for Verification

J. Bredereke, R. Gotzhein, F. H. Vogt

Universität Hamburg, Vogt-Kölln-Str. 30, 2000 Hamburg 54, Germany

Abstract
One main purpose for the use of formal description techniques (FDTs) is formal reasoning and verification. This requires a formal calculus and a suitable formal semantics of the FDT. In this paper, we discuss the basic verification requirements for Estelle, and how they can be supported by existing calculi. This leads us to the redefinition of the standard Estelle semantics using Lamport's temporal logic of actions and Dijkstra's predicate transformers.

Keyword Codes: F.3.2; D.2.1; C.2.4
Keywords: Semantics of Programming Languages; Requirements/Specifications; Distributed Systems

1 Introduction

Formal description techniques (FDTs) serve two main purposes (see, e.g., [ISO88, ISO89]). Firstly, specifications written in an FDT shall be precise and unambiguous. This requires the semantics of the FDT to be defined in a mathematical way. Secondly, an FDT shall support formal reasoning and, in particular, the formal verification (i.e., exhaustive proof) that a specification meets its (more abstract) requirements. Again, this requires the semantics to be defined in a mathematical way. Additionally, the semantics has to fit a formal calculus, which is used to reason about the specification. While the first aspect is usually taken into account in the definition of FDTs, the second aspect plays only a subordinate role in some formalisms.

In this paper, we consider the suitability of Estelle for formal reasoning and verification. Estelle ([ISO89, Dia89]) is an FDT for the specification of distributed information processing systems, in particular communication services and protocols of the OSI Basic Reference Model. It has the status of an international standard since 1989.

In previous work, we have attempted to formally verify the specification of the InRes-communication protocol written in Estelle against the InRes-service specified in logic ([Bre90, Got90, Hog91]). During this case study, it turned out that large parts of the proof had to be conducted on the basis of informal rather than formal reasoning. We believe that this was mainly due to the style of the original Estelle semantics, which defines the underlying abstract automaton *explicitly*. Such a style supports model checking, testing, and execution. However, exhaustive verification of properties is less well supported.

In this paper, we propose to use a different style to capture the Estelle semantics, which is still operational (as the original Estelle semantics), but which defines the underlying abstract automaton *implicitly*. This means that the automaton is characterized by its properties. We argue that this style is more convenient for the exhaustive verification of system properties. To define this semantics, we use a logical formalism based on Lamport's Temporal Logic of Actions ([Lam91]) and on Dijkstra's predicate transformers ([DiSc90]).

The paper is organized as follows. In section 2, basic verification requirements are discussed. In section 3, we outline the definition of an Estelle semantics that is a suitable basis for verification. In a first step, possible techniques for defining the meaning of Estelle transitions are studied. Based on these results, an Estelle execution model is formally defined. In section 4, the results are briefly discussed, and a preview of future work is given.

2 Basic Verification Requirements

There are three basic requirements to formally verify an Estelle specification:

- A formal description of the properties the system is expected to have.
- A formal description of the system in Estelle.
- A way to formally relate the Estelle description to the system properties.

The first requirement says that one needs to have an (even more) abstract formal description of how the system should behave. For concurrent, distributed information processing systems, temporal logic (TL) is one suitable formalism ([Hai82, ScMe82, Lam91, GoVo91]; see also [Pnu86, Got92]). It is generally agreed that there are two basic kinds of system properties that should be specified: *safety* properties (*"nothing bad will happen"*) and *liveness* properties (*"something good will happen"*).

For the second requirement, one needs the specification written in Estelle plus a formal definition of its semantics. Estelle specifications are based on extended finite state automata (FSA), so their meaning should be expressed by some sort of transition system. One obvious possibility to give the semantics is to define an explicit transition system describing the behaviour of the Estelle specification. In the international standard, the semantics for Estelle is defined this way. Another possibility is to define this transition system only implicitly by a logical formula describing its properties.

Thirdly, one wants to check formally if the system meets its requirements. One proof technique is *model checking* ([ClEmSi86]). It can be performed if the system's requirements are specified by some TL formula, if the system is given as an explicit transition system (e.g. a FSA graph), and if this transition system has a finite number of states. (Since Estelle is based on extended FSA with possibly infinite state spaces, model checking is *not* suited for Estelle in general. But it may serve to check some aspects; if an appropriate abstraction step has yielded a finite number of states.) It works as follows: a TL formula can be *interpreted* in different models (i.e. trees of states[1]); the transition system defines a tree of states; and there exists an algorithm ([ClEmSi86]) that takes the finite transition system and checks if the formula is *satisfied* in this model.

Another proof technique is *theorem proving*. For this, the transition system has to be given implicitly by a logical formula Ψ. The required properties of the system have to be given as a logical formula Φ. Then, everything can be done in (temporal) logic; one has to prove the initial validity of $\Psi \Rightarrow \Phi$. Furthermore, this way it is not necessary to inspect all possible states individually.

The formal proof system for this has to support all important methods of proving properties. For safety properties, the basic method is a generalization of the one commonly used to prove partial correctness of the general iteration in sequential programming (called **do**...**od**-loop in [DiSc90]): find a suitable global invariant; prove it true for the system start; prove that every single transition preserves the truth of the invariant; prove that the invariant implies the property. Then one can conclude that the system possesses the property. According to Lamport ([Lam91]), the proofs of the different kinds of liveness properties are always reduced to the proof of *leads-to* properties. For this, one can (hopefully) use already given liveness properties (e.g. "fairness" of the system, ...). Otherwise one has to perform a basic counting-down argument based on a well-founded partial order, which is a straightforward generalization of the method used to prove termination of the general iteration: show the existence of a function as follows: it maps each state onto some element of the well-founded order; the expected condition is fulfilled for every state which is mapped onto the least element; except for those states with the least value, the function value is lesser for each successor state. Additionally, show that some next state must exist. Then one can conclude that eventually the expected condition will be fulfilled. (There are a lot of small variations of this method.)

All these proof methods are supported by Lamport's new Temporal Logic of Actions TLA ([Lam91]). Lamport observed that in all these methods one refers only to a single state or at most to a pair of states at a time, and no temporal reasoning is needed except for the last step of conclusion. Everything before it can be performed in common (first order) logic. The only thing Lamport added to the logic part of TLA is the concept of the *action*, which means the relation between two directly successive states. The first state is always referred to by unprimed variables, and the second state by primed variables. (Additionally, an action must be annotated by those variables which have to remain unchanged if the action is *not* taken. This achieves invariance under refinement of time steps.) TLA is formally defined in [Lam91].

With an implicit description of the transition system, all reasoning can be done in logic. In addition, a logical formula can be reduced by implication to those properties which are interesting for the moment. On the contrary, the use of an explicit transition system (like the one currently used for Estelle) would mean the explicit construction of the model describing the system and would therefore in most steps involve the handling of

[1]in branching time TL

the complicated nested tuples and sets which describe a global state. So, for verification purposes the implicit description of a transition system turns out to be more adequate.

To come back to the three basic requirements for a formal verification of an Estelle specification: the description of properties to prove can be done formally in TLA, and TLA also provides a suitable and elegant formal proof system. So only the second requirement remains to be investigated. A formal definition of the semantics of an Estelle specification has to be found that harmonizes with the formalism fulfilling the first two requirements.

3 Design of a Formal Estelle Semantics

To express the static part of an Estelle specification (properties implied by declarations of variables, definitions of channels, ...), first order logic suffices. (See [Bre92].) Therefore TLA is suitable for this task, too. The real question still to be investigated is how to define the part describing the dynamic behaviour. This part consists of the Estelle transitions and of the so-called *execution model*. The execution model is common to all Estelle systems, it determines the temporal development of the system and is directed by (the formal representation of) the specifications of the Estelle transitions.

As stated in section 2, the meaning of an Estelle specification should be expressed by a logical formula. This means that some denotation function has to be defined, which maps specification texts onto logical formulas. Consequently, the execution model is a subexpression which will be part of every denotation. The design of the denotation function is described in [Bre92].

3.1 Formal Semantics of Estelle Transitions

From the arguments in section 2 it is clear that some kind of first order logic should be sufficient to define formally the semantics of Estelle transitions, but which formalism is suitable?

An Estelle transition consists of two parts: the *transition condition* and the *transition body*. The formal representation of the transition condition, describing the circumstances under which the Estelle transition may be fired, is trivial since it (nearly) has already the form of a simple predicate.

The transition body describes the relation between a predecessor and a successor state; it is expressed in terms of Pascal-like sequential statements. (Nevertheless, it still describes an *atomic* operation.) Therefore it is possible to regard each transition body separately as a sequential, transformational program. To define its semantics, all methods using first order logic should be investigated that are available to define the semantics of such programs:

- A single predicate about the pre- and post-state. (E.g. a TLA action)
- A Hoare-style assertion with pre- and postconditions.

- A predicate transformer à la Dijkstra.
- A hybrid formalism from the above.

Since TLA was so well-suited for everything else, let us begin with a look at the first method. For practical usability, any semantic definition method should work as compositional as possible. This means that the semantics of bigger objects is composed of the semantics of smaller objects without refering to the structure of the smaller objects. In this case, predicates for simple Estelle statements should be defined, and also rules should be defined to combine these predicates step by step into one single predicate describing the entire transition body.

In [Hoa85], Hoare proposes that "*Programs Are Predicates*" and claims "that a computer program can be identified with the strongest predicate describing all relevant observations that can be made of a computer executing the program". Then, he starts to define predicates for simple statements like the assignment and the conditional. Alas, for the sequential composition ";" (which is very important for the Pascal-like transition bodies) he gives no definition, just some algebraic laws and the comment that "it is quite difficult to formulate the definition in a satisfactory fashion". The problem arises when the sequential composition encounters the general iteration (which is important for transition bodies, too). Without definition in terms of other operators, ";" must be a basic operator of the calculus, which therefore cannot be the common first order logic. According to [ZwRo89], the weakness in dealing with sequential composition and iteration is common to all formalisms of this kind.

To understand the origin of this problem, we look at how one treats sequential composition in proofs. Consider the following two assignment operations and their predicates:

`x:=y`	$x_1 = y_0 \wedge y_1 = y_0$
`x:=x+1`	$x_1 = x_0 + 1 \wedge y_1 = y_0$

To compose them sequentially, one identifies the variables refering to the post-state of the first operation with the variables refering to the pre-state of the second operation:

`x:=y; x:=x+1` $\quad x_1 = y_0 \wedge y_1 = y_0 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1$

Furthermore, one has to get rid of the "intermediate" state x_1 and y_1. In this simple example one easily transforms the predicate into an atomic action again:

$$x_1 = y_0 + 1 \wedge y_1 = y_0$$

But such a transformation is already part of a proof, it cannot be done (automatically) for arbitrary operations by a semantics definition. Another solution is always available: instead of eliminating the "intermediate" variables, one can hide them by existential quantification:

$$(\exists v, w :: v = y_0 \wedge w = y_0 \wedge x_1 = v + 1 \wedge y_1 = w)$$

With a lot of ";"-operators (like in an average transition body) this yields predicates that are very big and clumsy to handle but at least in principle it is feasible.

The problem arises when the general iteration joins in. It is characterized by a number of repetitions that is not statically determined. This makes it impossible to supply a sufficient number of existential quantifications. Even if the formalism is extended by an infinite existential quantification, reasoning with it would be awkward. Therefore, the approach using a single predicate is fine for a top-down refinement from specifications to programs. (TLA is intended for it, and [Hoa85, p. 153] passes similar remarks.) But this approach is not well suited for the definition of the semantics and for the analysis of programs since this is done bottom-up.

In search of a solution to our problem we turn to the second method for defining the semantics of a Pascal-like program: the Hoare-style assertion. This method ([Hoa69, Apt81]) annotates a program tr with a precondition P and a postcondition Q:

$$\{P\}\texttt{tr}\{Q\}$$

Provided P is true before execution, the result of tr will render Q true afterwards (if tr terminates). The sequential composition is handled very elegantly by the following proof rule:

$$\frac{\begin{array}{c}\{P\}\texttt{tr1}\{Q\}\\ \{Q\}\texttt{tr2}\{R\}\end{array}}{\{P\}\texttt{tr1;tr2}\{R\}}$$

Alas again, in this method the full semantics of a program is defined by the complete set of assertions that are provable for it. A single assertion $\{P\}\texttt{tr}\{Q\}$ may be sufficient to prove a certain property; but in general, P and Q do not contain everything that can be said about tr. For instance, it can be proven for an arbitrary tr:

$$\{\mathsf{True}\}\texttt{tr}\{\mathsf{True}\}$$

Nevertheless it is possible to choose P and Q such that they describe the full semantics of tr. It can be done ([Bre92]) by fixing one of these predicates to a form that describes the respective state of the system completely. (E.g., the value of every variable x, y, ... is captured by some constant x_1, y_1, ...: $x = x_1 \wedge y = y_1 \wedge \ldots$) By doing this, that predicate becomes redundant and could be dropped. Consequently, we would return to the previous method, all its problems included. The elegant proof rule above becomes clumsy because there is no more a common intermediate predicate Q since both tr1 and tr2 insist on *their* choice of Q. Summary: Hoare-style assertions are a method suitable for proving that a given program meets a given specification. But a *single* assertion is not suitable to define the entire semantics of a transition body.

Next, we investigate the third method for defining the semantics of a Pascal-like program: a predicate transformer à la Dijkstra. The above Hoare-style semantics was defined by a binary relation on predicates, but this relation was only implicitly given by a set of proof rules. The actual pairs had to be found by a (manual) proof. Finding the corresponding counterpart for an arbitrary predicate is automatized by Dijkstra's predicate transformer formalism ([DiSc90]). A predicate transformer is an *explicit* operator. It performs only layed-down syntactic manipulations on a predicate; therefore, it can be automated. In the (single) predicate transformer defining the semantics of a program, all the information about this program is contained. For instance, the predicate transformer *weakest liberal precondition* for the assignment statement `"x:=y+z"` is defined to be the textual substitution $(x{:=}y+z)$. It replaces every occurrance of the text "x" by the text "$y+z$". If the postcondition is "$x < 42$", then the weakest liberal precondition to assure it by the operation is "$y+z < 42$". As can be seen, the definition of this operation relies heavily on textual substitution, and most of the other predicate transformers do that, too. Dijkstra's notation deviates a little from the standard. Instead of

$$\mathsf{wlp}(\texttt{"x:=y+z"}, x < 42) = y + z < 42$$

he writes

$$[\mathsf{wlp}.\texttt{"x:=y+z"}.(x < 42) \equiv y + z < 42]$$

This "Curried" dot-notation of functions with just a single argument makes it easy to talk about the function "wlp.`"x:=y+z"`". (We will need it later on.) The sequential composition of operations is handled very elegantly by functional composition. For every predicate X:

$$[\mathsf{wlp}.\texttt{"op1; op2"}.X \equiv \mathsf{wlp}.\texttt{"op1"}.(\mathsf{wlp}.\texttt{"op2"}.X)]$$

And for the general iteration does exist a closed form of definition; its combination with the sequential composition renders no problem. In [Bre92], we investigated in depth that predicate transformers support all the verification methods mentioned in section 2. Here, only a very simple example can be given. Suppose that we want to verify a safety property by the method in section 2, and we just want to show that a certain Estelle transition tr preserves the global invariant I:

$$[I \quad \Rightarrow \quad \mathsf{wlp.tr}.I]$$

Assume the global invariant is $z = x2^y$, the transition body is `"x:=x div 2; y:=y+1"`, and the transition condition $2|x$. Therefore, we have to prove true:

$$[(z = x2^y \wedge 2|x) \quad \Rightarrow \quad \mathsf{wlp}.\texttt{"x:=x div 2; y:=y+1"}.(z = x2^y)] \tag{1}$$

This means:

$$[(z = x2^y \wedge 2|x) \quad \Rightarrow \quad (x{:=}x\mathsf{div}2).((y{:=}y+1).(z = x2^y))]$$

Which is equivalent to:

$$[(z = x2^y \wedge 2|x) \Rightarrow z = (x\mathsf{div}2)2^{y+1}]$$

The rest of the proof is trivial. Of course, this example is not entirely formal since the integration of the predicate transformer formalism into a TLA environment still has to be performed. Also, it could be noted that the proof was carried out by hand. But manual work was necessary only during verification, *not* during semantics definition. It is well known that somewhere in the verification process the verificator's ideas unavoidably are necessary. But here, it is not in the semantics definition; a denotation *function* exists. As a résumé can be stated that Dijkstra's predicate transformers are well suited both for a semantics definition of transition bodies and for verification. In particular, the ubiquitous sequential composition is handled elegantly.

Finally, we turn to hybrid forms out of predicates and predicate transformers for defining the semantics of transition bodies. In every case, some kind of hybridization is necessary because TLA proved to be an appropriate temporal framework but was not well suited for transition body semantics. One idea is to take a predicate describing the system's state unambiguously (like in the discussion of the Hoare-style assertions above) and stuff it into a predicate transformer describing the transition body. This yields a predicate that describes the entire semantics of the transition body. The idea is feasible, but in [Bre92] it turned out that verification with it is more complicated than with the next idea: a calculus comprising both predicate transformers and predicates (actions). An example can be found in [ZwRo89]; it allows explicit transformation between the different representations, but actually performing them is still as hard as any proof. So, for our purposes it seems better to restrict ourselves (1) to be able to prove with predicates, (2) to be able to prove with predicate transformers, and (3) to have some easier to handle, specialized proof rules which refer to both formalisms and which implement the proof methods from section 2.

In [Bre92] we design such a calculus (called TLA/PT). It includes TLA; and TLA's first order part is augmented by usual predicate transformers that are applied to a predicate. The action part is augmented by predicate transformers that describe actions, and the temporal part by specialized proof rules. The definition of the second kind of predicate transformers creates a problem ([Lam91a]): most predicate transformers specify that certain variables are modified and all others remain unchanged. On the other hand, to keep the simplicity of logic, a formula should only state something about those variables which are mentioned explicitly. Otherwise, $F = (\exists x :: F)$, with x not free in formula F, would not be valid anymore. As indicated earlier, every TLA action is annotated by an explicit *state function*; it contains the variables that must remain unchanged if the action is *not* taken. Consequently, we define annotated predicate transformers that state which variables are affected by them at most when they *are*[2] taken:

$$\mathsf{PT}_f \triangleq (\exists h :: f' = h \wedge \neg \mathsf{PT}.(f \neq h))$$

This way, nothing is stated about variables not explicitly mentioned. Finally, TLA/PT adds a proof rule that takes something like implication (1) as one of its premises and which formalizes the respective conclusion.

3.2 A Formal Estelle Execution Model

Using TLA/PT, we are able to give an entirely formal definition of the Estelle execution model. It fits on something more than a single page, and it can be found in figures 1

[2]Since annotated predicate transformers are actions, they are annotated a second time to tell what must remain unchanged if they are not taken.

$$
\begin{array}{l}
\mathsf{ExecutionModel} \triangleq \\
\qquad \mathsf{ExecutionModelWithoutLiveness} \\
\quad \wedge\ (\forall \mathsf{tr} : \mathsf{tr.Id} \in \mathsf{Sys.Trans} : \\
\qquad\qquad \mathsf{WF}_{\mathsf{tr.Selected}}(\quad \neg\mathsf{tr.Selected} \wedge \mathsf{tr.Selected}' \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathsf{ExecutionModelWithoutLiveness}) \\
\qquad\quad \wedge\ \Box(\mathsf{tr.Selected} \Rightarrow \Diamond\neg\mathsf{tr.Selected}))
\end{array}
$$

This definition of the execution model is based on the following subsidiary definitions:

$$
\begin{array}{l}
\mathsf{ExecutionModelWithoutLiveness} \triangleq \\
\qquad \Box\mathsf{LocExecutionModel}(\mathsf{Sys}) \\
\quad \wedge\ \mathsf{IniDefs} \\
\quad \wedge\ \Box[(\forall \mathsf{tr} : \mathsf{tr.Id} \in \mathsf{Sys.Trans} \cup \{\mathsf{Sys.Ini.Id}\} : \\
\qquad\qquad \mathsf{tr.Selected} \neq \mathsf{tr.Selected}')]_{\mathsf{All}}
\end{array}
$$

$$
\mathsf{All} \triangleq (\mathsf{Sys}, \mathsf{Attach}, \mathsf{Connect})
$$

$$
\begin{array}{l}
\mathsf{IniDefs} \triangleq \\
\qquad \Box(\mathsf{Sys.Ini.Selected} \in \mathsf{Boolean}) \\
\quad \wedge\ \mathsf{Sys.Ini.Selected} \\
\quad \wedge\ \Box[\neg(\neg\mathsf{Sys.Ini.Selected} \wedge \mathsf{Sys.Ini.Selected}')]_{\mathsf{Sys.Ini.Selected}} \\
\quad \wedge\ \Box(\mathsf{Sys.Ini.Selected} \Rightarrow (\forall \mathsf{tr} : \mathsf{tr.Id} \in \mathsf{Sys.Trans} : \neg\mathsf{tr.Selected})) \\
\quad \wedge\ \Diamond(\neg\mathsf{Sys.Ini.Selected}) \\
\quad \wedge\ \Box[\quad (\mathsf{Sys.Ini.Selected} \wedge \neg\mathsf{Sys.Ini.Selected}') \\
\qquad\quad \Rightarrow (\qquad (\qquad \mathsf{Sys.IniTrans} \neq \{\} \\
\qquad\qquad\qquad\qquad \Rightarrow (\exists \mathsf{tr} : (\mathsf{tr.Id} \in \mathsf{Sys.IniTrans}) : \\
\qquad\qquad\qquad\qquad\qquad ((\mathsf{tr.Effect\_PT}).(\mathsf{Change}.\{\mathsf{Sys.Ini.Selected}\}))_{\mathsf{All}})) \\
\qquad\qquad\quad \wedge\ (\qquad \mathsf{Sys.IniTrans} = \{\} \\
\qquad\qquad\qquad\qquad \Rightarrow (\mathsf{Change}.\{\mathsf{Sys.Ini.Selected}\})_{\mathsf{All}}))]_{\mathsf{Sys.Ini.Selected}}
\end{array}
$$

Figure 1: Formal definition of the entire execution model (except quantitative time aspects). First part.

and 2. As stated in the beginning of section 3, this formula is part of the output of the denotation function which maps specification texts onto TLA/PT formulas. Therefore, the formula has to be understood in the context of the other output. The definition of the complete denotation function would amount in describing the full semantics of Estelle; such a definition is beyond the scope of this paper for its space limitations alone. A design of the complete denotation function can be found in [Bre92]. Here, we restrict ourselves to defining the execution model. Due to space limitations, we stripped figures 1 and 2 off any quantitative time aspects (introduced — only — by "`Delay`-clauses"). For the complete definition refer to [Bre92].

Some more remarks on TLA/PT are in place. We assume traditional temporal logic and

$$
\begin{array}{ll}
25 & \mathsf{LocExecutionModel}(\alpha) \triangleq \\
26 & \qquad (\forall \beta : \beta.\mathsf{Type} \in \alpha.\mathsf{LocModDefs} \wedge \beta.\mathsf{Exists} : \mathsf{LocExecutionModel}(\beta)) \\
27 & \quad \wedge \;\; (\alpha.\mathsf{ModDefs} = \alpha.\mathsf{LocModDefs} \cup (\bigcup \beta : \beta.\mathsf{Type} \in \alpha.\mathsf{LocModDefs} : \beta.\mathsf{ModDefs})) \\
28 & \quad \wedge \;\; (\forall \mathsf{tr} : \mathsf{tr.Id} \in \alpha.\mathsf{LocTrans} : \\
29 & \qquad\qquad \mathsf{tr.Enab} = \mathsf{tr.Provided} \wedge \mathsf{tr.When} \wedge \alpha.\mathsf{State} \in \mathsf{tr.From}) \\
30 & \quad \wedge \;\; (\forall \mathsf{tr} : \mathsf{tr.Id} \in \alpha.\mathsf{LocTrans} : \\
31 & \qquad\qquad\qquad [\qquad (\neg \mathsf{tr.Selected} \wedge \mathsf{tr.Selected}') \\
32 & \qquad\qquad\qquad\quad \Rightarrow (\qquad \mathsf{tr.Enab} \\
33 & \qquad\qquad\qquad\qquad\qquad \wedge \;\; (\forall \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \alpha.\mathsf{Trans} : \neg \mathsf{tr}_2.\mathsf{Selected}) \\
34 & \qquad\qquad\qquad\qquad\qquad \wedge \;\; (\forall \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \alpha.\mathsf{LocTrans} \setminus \{\mathsf{tr.Id}\} : \neg \mathsf{tr}_2.\mathsf{Selected}') \\
35 & \qquad\qquad\qquad\qquad\qquad \wedge \;\; (\forall \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \alpha.\mathsf{LocTrans} \wedge \mathsf{tr}_2.\mathsf{Priority} < \mathsf{tr.Priority} : \neg \mathsf{tr}_2.\mathsf{Enab}) \\
36 & \qquad\qquad\qquad\qquad\qquad \wedge \;\; (\mathsf{Change}.(\bigcup \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \mathsf{Sys.Trans} : \\
37 & \qquad\qquad\qquad\qquad\qquad\qquad \{tr_2.\mathsf{Selected}\}))_{\mathsf{All}})]_{\mathsf{tr.Selected}} \\
38 & \qquad\qquad\quad \wedge \;\; [\qquad (\mathsf{tr.Selected} \wedge \neg \mathsf{tr.Selected}') \\
39 & \qquad\qquad\qquad\quad \Rightarrow ((\mathsf{tr.Effect\_PT}).(\mathsf{Change}.\{\mathsf{tr.Selected}\}))_{\mathsf{All}}]_{\mathsf{tr.Selected}}) \\
40 & \quad \wedge \;\; (\qquad (\exists \mathsf{tr} : \mathsf{tr.Id} \in \alpha.\mathsf{LocTrans} : \mathsf{tr.Enab}) \\
41 & \qquad\qquad\quad \Rightarrow (\forall \mathsf{tr} : \mathsf{tr.Id} \in \alpha.\mathsf{Trans} \setminus \alpha.\mathsf{LocTrans} : \\
42 & \qquad\qquad\qquad\quad [\neg(\neg \mathsf{tr.Selected} \wedge \mathsf{tr.Selected}')]_{\mathsf{tr.Selected}}) \\
43 & \quad \wedge \;\; (\forall \mathsf{tr} : \mathsf{tr.Id} \in \alpha.\mathsf{Trans} : \\
44 & \qquad\qquad\quad [\qquad (\neg \mathsf{tr.Selected} \wedge \mathsf{tr.Selected}') \\
45 & \qquad\qquad\qquad \Rightarrow (\qquad (\qquad \alpha.\mathsf{Class} \in \{\mathsf{SystemProcess}, \mathsf{Process}\} \\
46 & \qquad\qquad\qquad\qquad\qquad\quad \Rightarrow (\forall \beta : \beta.\mathsf{Type} \in \alpha.\mathsf{ModDefs} \wedge \beta.\mathsf{Exists} : \\
47 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\exists \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \beta.\mathsf{LocTrans} : \\
48 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Enabled}(\qquad \neg \mathsf{tr}_2.\mathsf{Selected} \wedge \mathsf{tr}_2.\mathsf{Selected}' \\
49 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \;\; \mathsf{LocExecutionModel}(\beta))) \\
50 & \qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow (\exists \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \beta.\mathsf{LocTrans} : \\
51 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \neg \mathsf{tr}_2.\mathsf{Selected} \wedge \mathsf{tr}_2.\mathsf{Selected}')) \\
52 & \qquad\qquad\qquad\qquad \wedge \;\; (\qquad \alpha.\mathsf{Class} \in \{\mathsf{SystemActivity}, \mathsf{Activity}\} \\
53 & \qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \neg(\exists \mathsf{tr}_2 : \mathsf{tr}_2.\mathsf{Id} \in \alpha.\mathsf{Trans} \setminus \{\mathsf{tr.Id}\} : \\
54 & \qquad\qquad\qquad\qquad\qquad\qquad\quad \neg \mathsf{tr}_2.\mathsf{Selected} \wedge \mathsf{tr}_2.\mathsf{Selected}')))]_{\mathsf{tr.Selected}})
\end{array}
$$

Figure 2: Continuation: formal definition of the local execution model.

its operators $\Box$, $\Diamond$ to be known[3]. The notation $(\forall x : F : G)$ is defined to be $(\forall x :: F \Rightarrow G)$, and $(\exists x : F : G)$ is defined to be $(\exists x :: F \wedge G)$. Unlike [Lam91], we don't use indentation as a substitute for parentheses, but still as a hint to the formula structure. Noteworthy is the use of compound variables. For records, the standard dot-notation is used (and should not be confused with the application of predicate transformers). The semantics definition of a record is based on that of an array as[4] in TLA$^+$ ([Lam91b][5]): $\mathsf{tr.Id} \triangleq \mathsf{tr}[\texttt{"Id"}]$ (And an array, for its part, is a special notation for a function.) In figure 2, line 31 to 37, one finds an expression of the form $[A]_{\mathsf{tr.Selected}}$. This is a TLA/PT action annotated by a state

[3] $\Box A$ (read: "always A") means that formula A will hold now and *always* in the future. $\Diamond A$ (read: "eventually A") means that A will hold now or *sometime* in the future.

[4] A minor change was necessary to accommodate to predicate transformers, see [Bre92].

[5] TLA$^+$ is a syntactic extension to TLA that facilitates the handling of large formulas. At the time of this writing, it is still under development.

function as described in the end of subsection 3.1: "tr.Selected" must remain unchanged if the action A is not taken.

The definition of the execution model in figures 1 and 2 is structured into several subsidiary definitions. The actual definition is found in the beginning (line 1 to 6). The longest subsidiary definition is "LocExecutionModel" (line 25 to 54). It states all safety properties of a single *module instance*. To understand it, one has to consider the module structure of Estelle systems. An Estelle system is a dynamic hierarchy of communicating module instances. A module instance can be dynamically created according to a generic template called *module (definition)*, and it can be destroyed again. Therefore, the formula defining the properties of a module instance will be part of a universal quantification over all existing module instances of its kind.

The hierarchical structure of an Estelle specification is preserved in the execution model. At the top of this hierarchy, there is a module instance that exists always, called "Sys" in the execution model. "LocExecutionModel(Sys)" (line 8) captures its properties. In this definition, the formula "LocExecutionModel(β)" appears recursively (see line 26), taking the immediate child module instances of "Sys" as argument, and so forth. Since the depth of the module hierarchy is finite, the recursion is finite, too. "α.LocModDefs" is α's set of identifiers of child module definitions, a corresponding equation is generated by the denotation function.

In Estelle, module instances are refered to by *module variables*. The denotation function generates for a module variable mv of module instance α with module definition mt the expression $\Box\alpha.\mathsf{mv.Type} = \alpha.\mathsf{mt}$. Together with the execution model, the properties of the module instance that mv is refering to can be derived. In order to define the properties of *all* existing module instances without loosing the simplicity of the formalism that makes reasoning feasible, we introduced one single restriction on the semantics of Estelle: for every module instance, there has to be exactly one module variable referencing it. This prohibits assignment of module variables, and the use of a module variable for the creation of a module instance while it refers to another one. The only actual restriction following from this is the loss of an unbounded creation of module instances.

In line 12, the variables Attach and Connect can be found. They describe the current communication structure of the Estelle system (which can be dynamic, too). Each is a set of pairs of interaction point identifiers. The Estelle statements `Attach/Detach` and `Connect/Disconnect` modify the respective variable, the communication operations use them. The variable All is defined to describe the entire system state.

In Estelle, transitions are not just simply fired. Each module instance is subject to a cycle of two phases. In the first phase, one transition (or several from different child module instances) are selected according to certain conditions, and in the second phase these transitions are fired in an arbitrary order. The end of the first phase is marked by the irrevocable selection of the transitions to be fired. In the second phase, no more selection takes place. It ends when all selected transitions have been fired. So, for every transition, there are two basic states: selected or not selected. To simplify reasoning about a single transition, we introduce a boolean variable tr.Selected attached to each transition tr. The action of selecting is expressed by $\neg\mathsf{tr.Selected} \wedge \mathsf{tr.Selected}'$, and the action of firing is expressed by $\mathsf{tr.Selected} \wedge \neg\mathsf{tr.Selected}'$. To impose conditions on these actions, or to connect them with (transition body) effects, logical implication can be used. Examples can be found in line 31 to 37 for conditions and in line 38 to 39 for effects.

In line 32 and in line 28 to 29, the association of transition selection with the Estelle transition clauses ("conditions") is formally expressed. For every transition, the denotation function generates predicates of the form $\Box(\ldots \wedge \mathsf{tr.Provided} = \mathit{boolterm})$ according to the specification text. Similarly, it generates a definition for the predicate transformer tr.Effect_PT which defines the effects of the transition body (see line 39).

A quantification over all transitions of the module instance α is made in line 28. "tr.Id" is an identifier for the transition tr. The denotation function generates expressions specifying that all these identifiers must be distinct. "α.LocTrans" is specified by a predicate (generated by the denotation function) to be the set of transition identifiers of α. "α.Trans" is the respective set union over this module instance, its child module instances, and all descendants. "Sys.Ini" is a special transition that initializes the entire system from a *preinitial state* ([ISO89]); see line 13 to 24.

The hierarchical structure of module instances is accompanied with a rule of priority between parent and child module instances. If a parent is enabled to select an Estelle transition, then none of its descendants is allowed to select an Estelle transition for firing. This is specified formally in line 40 to 42.

Estelle allows different degrees of parallelism between module instances. Near the top level of the hierarchy, *system* module instances (which have no active parent) act completely unsynchronized with respect to each other. This is expressed formally by simply not imposing any further restrictions on them. The child module instances of instances declared as *process* act in a parallel synchronized way. Except those children which are inhibited otherwise, e.g. by a transition clause or by the parent/child priority, all children have to be selected simultaneously for firing (or none). Afterwards, they may fire in any order, until all have done so. Then, a new cycle of selection begins. This behaviour is specified in line 44 to 51. The implication operator in line 45 puts appropriate restrictions on the selection of any Estelle transition belonging to a descendant (line 44). In line 48, the *Enabled* operator of TLA/PT checks if an Estelle transition of a certain module instance β *could* be selected in accordance with the remaining execution model. If this is the case, then the implication in line 50 demands the selection of an Estelle transition of β as a condition on the selection in line 44.

Child module instances of instances declared as *activity* progress in a sequentially synchronized mode. At most one of them may be selected at the end of the selection phase. In line 52 to 54, it is specified that none of the other Estelle transitions in question may be selected together with the one in line 44.

In the international standard, liveness properties are indicated only a few times, and only informally. We decided to include the indicated properties in our formal specification. In line 4 to 6, most of them can be found. In line 6, it is assured that every transition, when selected, eventually will be fired (compare [ISO89, ch. 5.3.4]). And in line 4 to 5, the selection of transitions is treated. "$\mathsf{WF}_f(A)$" is a TLA/PT notation denoting *weak fairness*. Informally[6] but illustrative, it can be explained by

[6]For the corresponding formal definition refer to [Bre92], or to [Lam91].

$\Box((\Diamond$"action A is taken"$) \vee (\Diamond$"action A is impossible"$))$. Here, this action is the selection of a transition in accordance with the safety properties. If it is possible long enough without interruption, it must be taken. But since the selection of one transition often disables others, this is only a "weak" property. In every new cycle, the same other transitions may be disregarded. Only where nothing can block an entire module instance (i.e. near the top of the module instance hierarchy), this liveness property indeed enforces progress (compare [ISO89, ch. 5.3.1/3, "NOTE"]).

As this example intimates, our new design of a formal Estelle semantics cannot be exactly equivalent to the formal semantics in the international standard. Based on a different approach, we defined a new semantics, which of course resembles the one in the standard as much as possible.

The definition of the operations related to Estelle statements like `Output`, `Init`, etc., does belong to the denotation function (see "tr.Effect_PT") and not to the execution model; it is therefore not mentioned in its formula.

With the preceding explanations, we hope that the reader can now work out on his own how the formula in figures 1 and 2 formally describes the rules that determine the development of an Estelle system.

4 Discussion and Future Work

We investigated the requirements for a calculus that is suitable for formal verification in Estelle. To reach this goal, a redefinition of the formal Estelle semantics turned out to be necessary. Among other things, we found out that such a calculus should be based on first order logic. Lamport's new Temporal Logic of Actions ([Lam91]) proved suitable for most of the requirements. Still, a satisfactory representation of transition bodies (which are important for the dynamic behaviour of a system) could be done best by Dijkstra's predicate transformers ([DiSc90]). In [Bre92], we defined the syntax and semantics of a calculus integrating both formalisms. It still retains the simplicity of logic, which is necessary for any practical usability. We also designed some proof rules, but they still have to be elaborated further. Based upon this formalism, we sketched a redefinition of the Estelle semantics. Then, we defined the entire execution model in a completely formal way.

The new Estelle semantics is operational like the one in the international standard, but we defined the underlying abstract automaton implicitely by just giving the automaton's properties. This is more convenient for the verification of required system properties, since the automaton is characterized in the same formalism as the system's properties. In [Lam91], Lamport stresses that it is important to use only one formalism, because "reasoning is practical only if the underlying formalism is simple". If different aspects are dealt with using different formalisms, then either one can leave their relationship informal, loosing all formal rigour; or one has to pay the formalistic expenses of the transition from one formalism to the other on each reasoning step, so that verification is not practicable anymore.

Despite the diverging goals of expressibility of a specification technique on the one hand

and of simplicity for ease of reasoning on the other hand, we succeeded in designing a formal semantics reconciling both.

Some topics are still left for future work. Our calculus has to be completed, and as soon as Lamport has finished TLA$^+$ (a syntactic extension that facilitates the handling of large formulas, see [Lam91b]), we should migrate to it, too. Finally, the Estelle denotation function has to be elaborated according to our design.

Nevertheless, the formal verification of Estelle specifications, being a main purpose of a formalization, turned out to be possible, provided that some mechanical support is available. Although the formalization of the Estelle execution model in figures 1 and 2 is quite compact, it is still too large for manual reasoning. For the practical feasibility of verification, mechanical support is necessary. (But this is true for the verification of any real system. Unavoidably, their complexity leads to lengthy formulas.) Verification needs some wit to get the right ideas, but it also needs a lot of painstaking labour to work things out. The latter can be handed over to computers, for instance the handling of formulas and the check that a transformation indeed obeys the respective axioms and rules. For TLA, first experiments with a prototype mechanical verification system have already been performed (see [Lam91]). The system described there is even able to prove (very) simple theorems by itself. After further development and progress in the heuristic capabilities of such tools, eventually the manual share could be reduced far enough (to the strategic proof decisions), to make the formal verification of Estelle specifications more interesting for practioners.

It can be expected that it will be feasible to define the semantics of other FDTs using the same style and the same formalism. For instance, SDL ([CCITT87]) seems to be a suitable candidate, since its semantics is also based on the model of extended FSA. A detailed treatment of these issues is left for future work.

Formal verification renders possible very high standards of reliability. Furthermore, it provides a much deeper understanding of a system's properties and problems ([Bre90]). Although it has to be supported by appropriate tools to be of practical relevance, even at present a system designer aiming for verification will write specifications that are better readable and understandable. If something cannot be verified in principle, probably it is not understood well and therefore may be a source of errors.

Acknowledgements

We wish to thank Leslie Lamport for fruitful discussions on our work, and for keeping us posted on the development of TLA. Additional thanks go to Peter Ladkin for helpful discussions on our work and on TLA, and for his encouragement. The Estelle study group at the University of Hamburg provided for exchange and information on current Estelle activities.

References

[Apt81] Apt, K. R.: *Ten Years of Hoare's Logic — A Survey.* In: ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, 1981, pp. 431–483

[Bre90] Bredereke, J.: *Specification and Verification of the InRes-Protocol Using Estelle and Temporal Logic.* Studienarbeit, Department of Computer Science, University of Hamburg, 1990 (in German)

[Bre92] Bredereke, J.: *Design of a Formal Semantics for Estelle Using TLA with Predicate Transformers.* Master thesis, Department of Computer Science, University of Hamburg, 1992 (in German)

[CCITT87] CCITT Recommendation Z.100: *Specification and Description Language SDL.* CCITT SG X, Contribution Com X-R15-E, 1987

[ClEmSi86] Clarke, E. M., Emerson, E. A., Sistla, A. P.: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.* ACM TOPLAS Vol. 8, No. 2, 1986, pp. 244–263

[Dia89] Diaz, M. et al. (eds.): *The Formal Description Technique Estelle.* North-Holland, 1989

[DiSc90] Dijkstra, E. W., Scholten, C. S.: *Predicate Calculus and Program Semantics.* Springer, New York, 1990

[Got90] Gotzhein, R.: *Specifying Communication Services with Temporal Logic.* In: Logrippo, L., Probert, R. L., Ural, H. (eds.): Protocol Specification, Testing, and Verification, X, North-Holland, 1990, pp. 295–309

[Got92] Gotzhein, R.: *Temporal Logic and Applications — A Tutorial.* Computer Networks and ISDN Systems 24, 1992

[GoVo91] Gotzhein, R., Vogt, F. H.: *The Design of a Temporal Logic for Open Distributed Systems.* International Workshop on Open Distributed Systems, Berlin, 8–11 October 1991

[Hai82] Hailpern, B. T.: *Verifying Concurrent Processes Using Temporal Logic.* LNCS 129, Springer, 1982

[Hoa69] Hoare, C. A. R.: *An Axiomatic Basis for Computer Programming.* In: Communications of the ACM, Vol. 12, 1969, pp. 576–583

[Hoa85] Hoare, C. A. R.: *Programs Are Predicates.* In: Hoare, C. A. R., Shepherdson, J. C. (eds.): Mathematical Logic and Programming Languages. Prentice-Hall, Englewood Cliffs, New Jersey, 1985, pp. 141–155

[Hog91] Hogrefe, D.: *OSI Formal Specification Case Study: The InRes Protocol and Service.* Report No. IAM-91-012, University of Bern, Switzerland, 1991

[ISO88] *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* ISO/TC97/SC21, IS 8807, 1988

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

[Lam91] Lamport, L.: *The Temporal Logic of Actions.* Report No. 79, Digital Equipment Corporation, Palo Alto, California, USA, 1991

[Lam91a] Lamport, L.: Private correspondence.

[Lam91b] Lamport, L.: *Introducing TLA$^+$.* Preliminary draft, 1991

[Pnu86] Pnueli, A.: *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.* In: Current Trends in Concurrency, LNCS 224, Springer, 1986, pp. 510–584

[ScMe82] Schwartz, R. L., Melliar-Smith, P. M.: *From State Machines to Temporal Logic: Specification Methods for Protocol Standards.* IEEE Transactions on Communications, No. 12, 1982, pp. 2486–2496

[ZwRo89] Zwiers, J., de Roever, W.-P.: *Predicates Are Predicate Transformers: A Unified Compositional Theory for Concurrency.* In: proceedings of the 8th ACM Symp. on Princ. of Distributed Computing, Edmonton, 14–16 August 1989, pp. 265–279