Specialization in Estelle [1]

R. Gotzhein[+], G. v. Bochmann[*]

Abstract. We present a constructive approach for the *incremental specialization* of Estelle module definitions. As a formal basis, a general model of object behaviour and two notions of specialization are chosen. An abstract semantics of Estelle module definitions is defined in this general model, which makes the notions of specialization applicable to Estelle. Several modifications of Estelle module definitions with the property that the resulting module definition specializes the starting module definition are introduced, which allows for incremental specialization. Finally, fusion of Estelle module definitions is introduced and defined in terms of incremental specialization. The presented approach is of particular interest in the software maintenance phase, because it can reduce the total effort of adding or modifying user requirements in certain situations.

0 Introduction

The software development process is usually modeled as a sequence of activities that lead from the problem to a correct software solution. Intermediate steps can include problem analysis, system design, system implementation, functional validation, performance checks, installation, and the acceptance by the customer. Once the software is accepted, the maintenance phase begins. Software maintenance covers the adaptation to modified requirements, such as new or different user requirements, changes in underlying system software or hardware, and the exchange of internal algorithms. The removal of errors that are detected after the acceptance of the software is also considered as part of the maintenance.

Experience has shown that the costs for software maintenance exceed the costs for software development significantly. Therefore, it would be interesting to find systematic approaches that can help reducing maintenance costs. In the following, we address maintenance in the course of new or modified user requirements. Consider, for instance, a telephone company that provides services to a large number of

---

[1] This research was performed during 1991-92 when R. Gotzhein was with the University of Montreal. The research was supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the Government of Canada.

[+] Department of Computer Science, University of Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern, Germany; email: gotzhein@informatik.uni-kl.de

[*] Département d'IRO, Université de Montréal, C.P. 6128, Succ. Centre Ville, Montréal, Québec, H3C 3J7, Canada; email: bochmann@iro.umontreal.ca

subscribers. The telephone company will certainly be interested in optimizing these services in order to cut costs. Also, it will be interested in adding new services and improving existing ones in order to increase revenues and competitiveness. In a free deregulated market, we can also have a situation where different companies provide different portions of a service, and where new companies enter the market by providing new, better, or cheaper services. In all these cases where the original service is modified, the resulting service has to be validated to ensure that it satisfies the new requirements. In addition, the service has to be validated to ensure that it still satisfies the old requirements as far as they have not been changed.

Since the original services have been validated before, the question arises whether it is possible to modify or extend them without validating the resulting services against the old requirements as far as they have been left unchanged. If so, this could reduce the total effort of system maintenance and thus the maintenance costs significantly. In particular, if the modifications to the system are relatively small, then such an approach would be most beneficial.

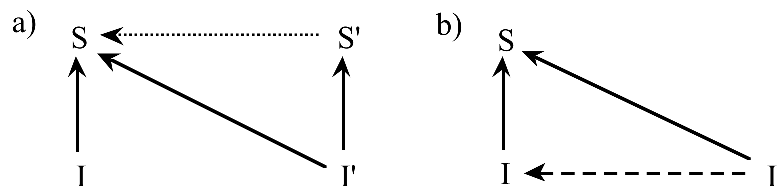

Figure 1: Different approaches to the development of correct software

To make this question a little more precise, consider the situation shown in Figure 1a. Here, we have a requirement specification S that is implemented by I. The arrow from I to S means that I correctly implements S, which has to be validated after the implementation is completed. When the requirements are to be changed, it is good engineering practice to modify S first, yielding S', and then to develop I' that realizes S'. In many cases, I' can be based on I. However, the validation that I' correctly implements S' has to start from scratch. Intuitively, if S' is an extension and/or specialization of S, then I' will also be correct with respect to S, as indicated by the shaded arrow. A different approach is shown in Figure 1b. *Here, the modifications are applied directly to I in a way that preserves the properties of I.* Thus, by construction I' will be correct with respect to S, as indicated by the shaded arrow. In this case, it suffices to validate I' against the new or modified user requirements.

This general approach can be made precise and be applied in different formal frameworks. For instance, we could use trace logic to specify S and S', and CSP to describe I and I' ([Hoa85]). In this case, we can use the relation "sat" to formalize correctness between I and S ($\longrightarrow$), and logical implication "$\supset$" to formalize specialization between S' and S ($\cdots\rightarrow$). Since under these relations, $(I' \text{ sat } S') \wedge (S' \supset S) \supset (I' \text{ sat } S)$ holds, we have exactly the situation shown in Figure 1a. However, there is no suitable relation to formalize specialization between I' and I ($-\rightarrow$), as suggested in Figure 1b. Failure equivalence would certainly be too

strong, because it would leave no room for modifications that are visible to the user.

In this paper, we will focus on the situation shown in Figure 1b and the use of the formal description technique Estelle ([ISO89], [BuDe87]) as a language for the description of I and I'. As a starting point, we consider single Estelle modules without substructure, where the queues of a module are not directly associated with the local module state. To formalize the notion of specialization in Estelle, we apply an abstract behaviour model with formal notions of specialization ([BoGo93]). This model is outlined in Section 1. In Section 2, we define an abstract semantics for Estelle module definitions in terms of the behaviour model. This semantics is related to the standard semantics in [ISO89]. Thus, the notions of specialization become applicable to Estelle module definitions. In Section 3, we examine several *syntactical* modifications of Estelle module definitions (such as adding, removing, and modifying transitions) and show that under certain restrictions they always lead to specialized module definitions (in the previously defined *semantical* sense). This means that in many cases, it can now be decided by a simple inspection of the Estelle module definition whether it is a specialization of another one. In these cases, it is no longer necessary to conduct proofs in the framework of the semantical model. In Section 4, we introduce fusion of Estelle module definitions.

In [Saq91], some initial work on specialization has been reported. However, the treatment there is informal, and the results differ significantly from our results.

1 A general framework for specialization

In our formal framework (for details, see [BoGo93]), the behaviour of an object is defined in terms of external actions that are offered to the environment. In the general case, the sets of offered actions may depend on the object's history, which is modeled as the sequence of previous actions. The basic notions of *action*, *offered actions*, *trace*, *behaviour*, and *specialization* are defined subsequently. More explanations and examples can be found in [BoGo93].

An action is a triplet f(i;o), where f is an operation name, i is the value of the input parameter(s) of the operation, and o is the value of the output parameter(s) returned by the object as result of the operation. This differs from other models (e.g. [Hen85], [Hoa85]), where actions have no structure on the model level.

Definition 1.1: Let F be a set of *operation names*, I be a set of *inputs*, and O be a set of *outputs*. Then $Act = F \times I \times O$ is the set of *actions*.

The sequence of actions that have occurred since the object has been instantiated is called *trace* (or *history*). After each trace, an object allows for a (possibly empty) set of actions, called *chosen set of offered actions*. We assume here that objects execute one action at a time. Therefore, the history of an object can be modeled as a sequence of actions. Nondeterminism is modeled by allowing, for a given trace, several sets of offered actions, and, for a given operation and input, several possible outputs. This leads to the following definition of behaviour in general.

Definition 1.2: Let B be a set of pairs (t,**A**), where t is a sequence, possibly empty (denoted <>), of actions, and $\mathbf{A} = \{A_1,...,A_n\}$ is a set of sets of actions. B is called a *behaviour* if it satisfies the following conditions:

a) if (t,**A**) ∈ B, then **A** ≠ {};
b) if (t,**A**) ∈ B and (t,**A'**) ∈ B, then **A** = **A'**;
c) (<>,**A**) ∈B, for some $\mathbf{A} = \{A_1,...,A_n\}$;
d) if (t,**A**) ∈ B, A ∈ **A**, and a ∈ A, then (t ^<a>,**A'**) ∈ B, for some **A'**;
e) if (t ^<a>,**A'**) ∈ B, for some **A'**, then there is **A** and A ∈ **A** such that (t,**A**) ∈ B and a ∈ A.

Definition 1.3: Let B be a behaviour. Then traces(B) is the set of all t such that there is (t,**A**) ∈ B. The elements of traces(B) are called *traces* of B. ^ denotes concatenation of traces. The elements of **A** are called *sets of offered actions* after t. A *state* of B is a pair (t,A), where A ∈ **A** and (t,**A**) ∈ B.

Definition 1.4: Let (t,A) be the current state of B, f be an operation, and i be an input. We say that B *blocks* for f(i), if there exists no output o such that f(i;o) ∈ A. B *blocks* for f, if for all i, it blocks for f(i). B *accepts* f(i), if there exists an output o such that f(i;o) ∈ A. The *domain* of B in state (t,A) is the set of f(i) accepted by B in this state. We say that B has an *undetermined output* for f(i), if there are at least two actions f(i;o) and f(i;o') in A with o ≠o'.

Undetermined outputs are also considered as nondeterminism.

Definition 1.5:

a) A behaviour B is *constant* iff for all traces of B, the same sets of actions can be offered.
b) A behaviour B is *state deterministic* iff for all traces of B, exactly one set of actions is offered, i.e., iff the state of B is always uniquely determined by the trace.
c) A behaviour B is *deterministic* iff it is state deterministic, and the output is always uniquely determined by operation and input.

A state deterministic behaviour[2] may still have undetermined outputs, thus state determinism is weaker than determinism. A constant, state deterministic behaviour is uniquely characterized by a single set of offered actions. State deterministic behaviours are uniquely characterized by a set of traces, as the following consideration shows: let B be a state deterministic behaviour, T = traces(B), then B = {(t,{A}) | t ∈ T ∧ A = {a | t^<a> ∈ T}} holds. Constant behaviour is independent of the object's history, i.e., for all traces, the object can offer the same sets of actions. Therefore, constant behaviour is uniquely characterized by a set **A** of sets of offered actions. Note that constant behaviours can be deterministic or nondeterministic.

As a starting point for the formal treatment of specialization, we assume that system specifications are given as properties that constrain the sets

---

2 In [Sta72], state deterministic behaviours have been called "observable"; in [Cer92], they have been termed "observably nondeterministic".

of traces and offered actions. Specialization can be understood as the selection of options, as the reduction of nondeterminism, or, in general, as the strengthening of these properties. If the system specification leaves options, then the system's environment must be prepared to deal with any of these possibilities. Thus, a specialized system with less options would not change the resulting behaviour to the worse. We call this form of specialization *reduction*.

Also, we may wish to add properties that express the extension of behaviour, strengthening again the original properties. If the new properties do not touch the original behaviour, then the environment can obtain the same services from the resulting system as from the system before the change was made. It can also obtain the services expressed by the new properties. We call this form of specialization *extension*. Note that extension includes reduction as described before.

On the operational level where a system is modeled as a behaviour, we will capture these forms of specialization by relations between behaviours. Reduction and extension are defined in terms of constrainment, domain coverage, and constrainment on the domain. Informally, a behaviour B' is constrained by a behaviour B, if for all traces of B and B', B' can offer only actions that B can offer, too. B' covers the domain of B, if for all traces of B and B', B' accepts everything that B accepts. B' is constrained by B on its domain, if for all traces of B and B', for the domain of B, B' can offer only actions that B can offer, too; B' may also offer actions f(i;o) that B cannot offer, if f(i) is outside the domain of B. We now define these relations formally, starting with the special cases where behaviours are characterized by sets A of offered actions and sets **A** of sets of offered actions, respectively.

For constant, state deterministic behaviours, we define the following relations:

Definition 1.6: Let B and B' be constant, state deterministic behaviours that are characterized by A and A', respectively.

a) A' *is constrained by* A (written "$A' <_c A$") iff $A' \subseteq A$.

b) A' *covers the domain of* A (written "$A' >_d A$") iff for all actions $f(i;o) \in A$, there is an action $f(i;o') \in A'$. Formally: $A' >_d A \equiv \forall f,i,o.\ (f(i;o) \in A \supset \exists o'.\ f(i;o') \in A')$

c) A' *is constrained by* A *on its domain* (written "$A' <_{cd} A$") iff for all operations f and inputs i, if A accepts f(i), then for all o, if $f(i;o) \in A'$, then $f(i;o) \in A$. Formally: $A' <_{cd} A \equiv \forall f,i.\ (\exists o.\ f(i;o) \in A \supset \forall o.\ (f(i;o) \in A' \supset f(i;o) \in A))$.

d) A' *reduces* A (written "$A' <_r A$") iff $A' <_c A$ and $A' >_d A$.

e) A' *extends* A (written "$A' >_e A$") iff $A' <_{cd} A$ and $A' >_d A$.

For constant behaviours, we define the following relations:

Definition 1.7: Let B and B' be constant behaviours that are characterized by **A** and **A**', respectively. **A**' R **A** iff for all $A' \in$ **A**', there is $A \in$ **A** s.t. A' R A, where R is replaced uniformly by one of the relations $<_c$, $>_d$, $<_{cd}$, $<_r$, or $>_e$, which are interpreted as in Definition 1.6. Formally: $\mathbf{A}'\ R\ \mathbf{A} \equiv \forall A' \in \mathbf{A}'.\ \exists A \in \mathbf{A}.\ A'\ R\ A$.

For arbitrary behaviours, we define the following relations:

Definition 1.8: Let B and B' be behaviours. B' R B iff for all (t,**A**') $\in$ B' and (t,**A**) $\in$ B, **A**' R **A** holds, where R is replaced uniformly by one of the relations $<_c$, $>_d$, $<_{cd}$, $<_r$, or $>_e$, which are interpreted as in Definition 1.7. Formally: B' R B $\equiv \forall$t,**A**,**A**'. ((t,**A**') $\in$ B' $\wedge$ (t,**A**) $\in$ B $\supset$ **A**' R **A**).

This covers the special cases of deterministic and state deterministic behaviours.

Corollary 1.1:

a) Let B, B' be behaviours. Then B' $<_c$ B implies traces(B') $\subseteq$traces(B).
b Let B, B' be behaviours. Then B' $<_c$ B implies B' $<_{cd}$ B.
c) Let B, B' be behaviours. Then B' $<_r$ B implies B' $>_e$ B.
d) $<_c$ and $<_r$ are transitive.
e) $>_e$ is transitive for constant behaviours, state deterministic behaviours, or subsets thereof. It is not transitive in general.
f) Let B, B' be deterministic behaviours. Then B' $<_r$ B iff B = B'.

The proofs for this corollary are listed in [BoGo93]. Part f) shows that the only room left for specializing object behaviour according to $<_r$ is the reduction of nondeterminism. Part c) shows that $<_r$ is stronger than $>_e$. Under $>_e$, it is possible to reduce the number of *unspecified receptions* and to extend the functionality by adding actions, which is prohibited by $<_r$.

2 Semantics of Estelle module instances

In this section, we outline (for details, see [GoBo92]) how an abstract semantics of Estelle module instances in terms of the behaviour model from Section 1 can be derived from the standard Estelle semantics. Once this is done, we can use the notions of reduction, extension, constrainment, domain coverage, and constrainment on the domain for Estelle. In particular, we can then investigate which modifications of Estelle module instances satisfy these relations. There are two main reasons why we do not use the conventional Estelle semantics from [ISO89]. Firstly, this semantics is not sufficiently abstract, because it refers to internal module states. To define formal relations for *external* module behaviour, internal states and structures are not relevant. Secondly, to our knowledge there is no suitable formal relation for specialization which is defined in terms of the conventional Estelle semantics.

In order to define the Estelle semantics in terms of the behaviour model from Section 1, we have to identify *actions*, *traces*, *offered actions*, and *behaviour* in the conventional Estelle model. We will focus on a single module instance without substructure. Also, we assume that transitions are deterministic. Nondeterminism of transitions can only result from the use of the constructs *forone*, *exist*, *all*, and from an undetermined initial state (see [ISO89]). In all other cases, the next state is always uniquely determined by the transition. Therefore, the assumption of deterministic transitions holds if the constructs *forone*, *exist*, *all* are not used in the specification, and the initial state is determined.

A module communicates with its environment by exchanging interactions at its external interaction points. When a transition fires, a single interaction can be accepted, and a (possibly empty) sequence of

interactions is sent. Therefore, we can capture the visible effects of a transition as a pair (i;o), where i is the accepted interaction, and o is the sequence of sent interactions. Note that the pair (i;o) consists of *qualified* interactions, i.e. interactions of the form ip.m(v_1,...,v_k) that include the interaction point. This is directly related to actions f(i;o), if we drop the operation name f, or if we assume that the operation name is the same for all actions.

Actions are defined as pairs (i;o), where i is a single qualified interaction (including "null" for spontaneous transitions, i.e., transitions that do not take an input), and o is a sequence of qualified interactions. Thus, the Estelle semantics characterizes the set of actions associated with a module instance. Different from actions as defined in Section 1, we have no operation, which can be considered as a special case where the operation is always the same and therefore omitted.

In order to define the traces of a module instance, we first extend the state of a module instance by a component *in* that records the interaction accepted by the previous transition and will be reset before the next transition fires. To incorporate this into the Estelle semantics, we slightly modify the auxiliary statement *reception* (see [GoBo92] for details). Since *reception* is only executed for input transitions, *in* will not be modified by the interpretation of spontaneous transitions.

Next, the set of *potential computations of module instance M* is defined (see [GoBo92] for details). We may think of this set as capturing the upper bound on the possible behaviour of M. In a given context as defined by a complete Estelle specification, only a subset of this behaviour can in general be triggered. There are two main differences to the notion of computation from [ISO89]. Firstly, the set of potential computations takes only a single module instance into account, therefore, potential computations are local. Secondly, M may have an environment, consisting of module instances that communicate with M. By defining potential computations, we do not wish to restrict the possible environments of M, therefore, we have made no assumptions about the input environment. Note that the definition of potential computations only takes the transitions of M into account.

With these preparations, we can define the abstract behaviour in terms of the model from Section 1 for a state deterministic Estelle module instance without substructure. For each state, the action (i;o) performed by the previous transition is recorded in the state components *in* and *out*. Different from actions as defined in Section 1, we have no operation, which can be considered as a special case where the operation is always the same and therefore omitted. From the set of computations, we obtain the set of traces by considering only the actions associated with each state. Since we consider only state deterministic module instances, the set of traces uniquely characterizes their behaviour (see Section 1).

It should be noted that a state deterministic module instance may include spontaneous transitions, if they have an output. These transitions have an externally visible effect, therefore, the corresponding action will appear in the trace when the transition is fired. Spontaneous transitions with an empty output are not visible, therefore, the corresponding action would not appear in the trace. This would usually result in a behaviour which is not state deterministic.

From the definition of the set of potential computations, it follows that for any reachable state, a next state only exists if there is an *explicitly* specified, fireable transition. Together with the definition of traces(M) and the fact that for all traces t, the set A of offered actions is defined as A = {a | t^<a> ∈ traces(M)}, it follows that an action can only be offered if a corresponding Estelle transition has been *explicitly* specified. This models the Estelle convention that unspecified receptions are not accepted, which is called "blocking by default". A different approach (which can also be modeled as a behaviour) has been taken in SDL, where unspecified receptions are accepted and discarded, a convention called "ignoring by default".

3 Incremental specialization of Estelle modules

In this section, we will consider changes to the definition of an Estelle module and investigate how the resulting definition is related to the original definition, based on the relations introduced in Section 1. Changes to a module definition apply to all module instances created from this definition. We will consider removal, addition, and modification of transitions, addition of control states, declarations, and interaction points, and the fusion of module definitions.

To illustrate the different forms of specialization, we will modify a customer whose behaviour is given by the Estelle specification in Table 3.1. In this example, the customer who is in the role a subscriber is connected to a tv station through the Estelle channel "cable". The definition of "cable" allows the subscriber to switch his tv set on and off, to switch to another program, and to pay his bill. The tv station may send commercials, movies, news, sports, and bills which show the amount due. The customer, as defined by C, only uses a subset of these features. In control state "idle", he can switch on the tv set. In control state "watching", he is prepared to switch it off again, or to watch a commercial. In the latter case, the customer's reaction (= output) is undetermined. Since he/she does not receive anything worth paying, the customer blocks for bills (in case the tv station wants to charge for commercials).

```
channel cable (tv_station, tv_subscriber);
        by tv_station:          commercial, movie, news, sports, bill
        (amount: integer);
        by tv_subscriber:       on, off, switch, pay (amount: integer);
module C_type activity;
        ip tv: cable (tv_subscriber) individual queue;
        end;
body C_body for C_type;
        state idle, watching;
        initialize to idle begin end;
        trans
                tr1: from idle to watching begin output tv.on end;
                tr2: from watching to same when tv.commercial begin
```

```
end;
        tr3: from watching to idle when tv.commercial begin
tv.off end;
        tr4: from watching to idle begin tv.off end;
end;
```

Table 3.1: Estelle specification of a customer

Some additional notation will be used in this section. M, M' denote module definitions specified by module header, module body, and channel declarations. A module definition M is a structure (states(M), decl(M), ip(M), itrans(M), trans(M)), where states(M), decl(M), ip(M), itrans(M), and trans(M) are the set of control states, the set of declarations, the set of external interaction points, the initialize transition, and the set of transitions of M, respectively. For a declaration d and an interaction point ip, id(d) and id(ip) denote the identifier that is declared; id is extended to sets of declarations and interaction points in the obvious way. By istate(M), we refer to the initial control state of M. We assume that all transitions are expanded. A transition named t has the form[3] "t: from s_t to s'_t when i_t provided p_t begin b_t end". We will not consider the priority- and delay-clauses. "i_t" is called "Estelle input" or "input" for short. Note that an Estelle input i_t corresponds to a set of inputs in the underlying model if i_t has associated parameters. If R is a relation between behaviours, then we write M' R M instead of Behav(M') R Behav(M). Transitions with a when-clause are called "input transitions", transitions without when-clause are called "spontaneous transitions". $t[state]_s$, $t[state']_{s'}$, $t[input]_i$, and $t[provided]_p$ are the transitions that are obtained from t by replacing s_t, s'_t , i_t, and p_t by state, state', input, and provided, respectively. $M[initialize]_{itrans}$ is the the module definition that is obtained from M by replacing itrans(M) by initialize.

If we want to specialize Estelle modules incrementally, it is necessary that specialization is transitive. Otherwise, subsequent specializations of module M_1 could result in a module M_n that does not specialize M_1 although M_i specializes M_{i-1} for $1<i\leq n$. From Corollary 1.1, we know that $<_r$ is transitive in general, but does not leave much room for specializing behaviour. On the other hand, $>_e$ gives more freedom by allowing the extension of behaviour, but is only transitive for state deterministic behaviours, constant behaviours, and subsets thereof. To investigate both notions of specialization, we will therefore focus on state deterministic behaviour. Informally, an Estelle module definition is state deterministic, if its initial state is "sufficiently determined", and if all spontaneous transitions produce an output that uniquely characterizes the next state.

---

[3] To improve the readability of the following examples, we use a syntax for transitions that slightly differs from the Estelle syntax in that the transition name appears in a different position, and without the keyword "name".

3.1 Removing transitions

Removal of transitions will result in a specialization if it reduces the module definition's nondeterminism and maintains its domain coverage. Since we focus on state deterministic behaviours, reduction of nondeterminism means reduction of undetermined outputs.

a) input transitions
Undetermined outputs may exist if the same Estelle input can be accepted by different transitions $t_1,...,t_n$ in the same control state. This is of course not a sufficient condition, since $t_1,...,t_n$ could all produce the same output for the same input. Also, we have to take the provided-clauses into account: a necessary condition for an undetermined output is that more than one provided-clause of $t_1,...,t_n$ can be true for some input.

Instead of giving sufficient conditions for the existence of undetermined outputs, we are interested here in sufficient conditions for the reduction of undetermined outputs while maintaining domain coverage. Intuitively, we may remove a transition t if there are other transitions $t_1,...,t_n$ accepting the same Estelle input in the same control state such that whenever t can fire, there is some transition t_i, $1 \le i \le n$, which can fire, too. To capture this idea formally, we define a function $Remove_t(M,t)$ that removes the transition t from module M iff the above conditions are satisfied.

Definition 3.1: Let M be a module definition, t be a transition, Removal_OK = $\exists t'_1,...,t'_n \in trans(M) - \{t\}. (\forall j. (1 \le j \le n \supset s_{t'_j} = s_t \wedge i_{t'_j} = i_t) \wedge \forall fv. (p_t \supset \bigvee_{1 \le j \le n} p_{t'_j}))$, where "fv" is the list of free variables in p_t and $p_{t'_j}$.

$Remove_t(M,t) =_{Df}$ **if** Removal_OK
then (states(M), decl(M), ip(M), itrans(M), trans(M) - {t}) **else** M

Proposition 3.1: $Remove_t(M,t) <_r M$

Example 3.1:

a) Let C1 $=_{Df}$ $Remove_t$(C,tr3). C1 $<_s$ C, because the number of undetermined outputs in control state "watching" on input "tv.commercial" is reduced by this transformation: C1 no longer switches off the television when a commercial is received.
b) Let C1' $=_{Df}$ $Remove_t$(C,tr1). Since in control state "idle", there is no undetermined output, the conditions for the removal of tr1 are not satisfied. Therefore, C1' = C. Nevertheless, C1' $<_r$ C holds.

The function $Remove_t$ can easily be generalized to define the removal of a list of transitions:

Definition 3.2: Let M be a module definition, T be a list of transitions.

$Remove_T(M,T) =_{Df}$ **if** $\neg$ empty(T) **then** $Remove_T$ ($Remove_t$ (M,head(T)), tail(T)) **else** M

Proposition 3.2: $Remove_T(M,T) <_r M$

Proof: Follows from Proposition 3.1 and the transitivity of $<_r$.

b) spontaneous transitions

Undetermined outputs may also exist if there are several different spontaneous transitions $t_1,...,t_n$ which can fire in the same control state. Recall that a spontaneous transition has no when-clause, therefore, the input is empty. We consider the empty input as a special input that is always available. Since we examine state deterministic behaviours here, we only allow spontaneous transitions that produce a visible, i.e., non-empty output. Thus, we can treat spontaneous transitions in the same way as input transitions.

3.2 Adding transitions

If transitions are added to a module definition, domain coverage is maintained. But the constrainment condition will not be satisfied, because the addition of transitions will extend the set of traces or increase the nondeterminism. Therefore, the addition of transitions cannot lead to a specialization in the sense of $<_r$. Under suitable restrictions, however, it can specialize object behaviour according to $>_e$.

According to $<_{cd}$, we may add actions to any set of offered actions as long as this does not make outputs of already offered actions more undetermined. This condition is satisfied if we add actions extending the domain in a given state. With respect to an Estelle module, this means that we may safely add a transition, if it accepts an input that cannot be accepted by existing transitions in the same control state[4].

At this point, it is important to see that a single Estelle input is a set of inputs of our underlying model, if it has associated parameter values. Also, the control state alone does not always determine whether a given input can be accepted. Further constraints taking the parameter values and the module state into account can be specified in the provided-clause. Thus, the set of inputs that can be accepted by a transition can be reduced. If this is the case, we may complement such a transition by adding another transition that accepts the same Estelle input in the same control state, but different parameter values or in different module states. An appropriate restriction can be stated in terms of the provided-clauses. To capture this formally, we define the function Add_t. We assume that the arguments of Add_t (and of all functions defined in the following) are syntactically correct, and that t is chosen such that its addition to the module instance M will result in a syntactically correct module as defined by Add_t.

Definition 3.3: Let M be a module definition, t be a transition, $Addition_OK = \forall t' \in trans(M).\ (s_{t'} = s_t \wedge i_{t'} = i_t \supset \forall fv.\ \neg(p_{t'} \wedge p_t))$, where "fv" is the list of free variables in $p_{t'}$ and p_t.

---

4 Addition of transitions does not satisfy $<_{cd}$ if we use the "ignoring by default" assumption, as it is, for instance, done in SDL.

$Add_t(M,t) =_{Df}$ **if** Addition_OK
then (states(M), decl(M), ip(M), itrans(M), trans(M) $\cup$ {t}) **else** M

Proposition 3.3: $Add_t(M,t) >_e M$

Example 3.3: Let tr5, tr6, tr7, and tr8 be the following transitions:

tr5: from watching to same when tv.movie begin end;
tr6: from watching to same when tv.news begin end;
tr7: from idle to same when tv.bill provided amount ≤100 begin end;
tr8: from idle to same when tv.bill begin output tv.pay(amount) end;

a) Let C3 $=_{Df}$ Add_t(C1,tr5). tr5 can be added to trans(C1), because the input "tv.movie" is not in the domain of C1 in control state "watching". Apart from commercials, C3 is eager to watch movies. It still refuses to accept anything else, in particular bills.

b) Let C3' $=_{Df}$ Add_t (Add_t (Add_t (Add_t (C2,tr5),tr6),tr7),tr8). As a result, the transitions tr5, tr6, and tr7 will be added to trans(C1). C3' accepts bills with an amount up to 100, but decides not to pay them. The tv station might decide to stop sending movies, but continue to send commercials over the cable. tr8 will not be added, because there exist values for "amount" such that p_{tr7} and p_{tr8} are both satisfied. Thus, addition of tr8 would increase the nondeterminism.

The function Add_t can easily be generalized to the addition of a list of transitions (compare Definition 3.2). But since we have assumed that the transitions of module definitions are deterministic (see also Section 2), undetermined outputs cannot be introduced in this way without violating $<_{cd}$. However, it is possible to introduce undetermined outputs if several transitions are added in one step. This leads us to the function Add_T, which generalizes Add_t.

Definition 3.4: Let M be a module definition, Additions_OK = $\forall t \in T$. Addition_OK.

Add_T(M,T) $=_{Df}$ **if** Additions_OK
then (states(M), decl(M), ip(M), itrans(M), trans(M) $\cup$ T) **else** M

Proposition 3.4: $Add_T(M,T) >_e M$

Example 3.4: Let C6 $=_{Df}$ Add_T (C2,{tr5,tr6,tr7,tr8}). As a result, the transitions tr5, tr6, tr7, and tr8 will be added to trans(C2). Note that this differs from C5 as defined in Example 3.3. Nevertheless, C6 $>_e$ C2 holds. C6 will pay bills that show an amount greater than 100. For amounts up to 100, it is undetermined whether C6 will pay.

3.3 Modifying transitions

In a few cases, a module definition can be specialized by modifying its transitions. This concerns the modification of the from-, to-, when-, and provided-clauses of transitions. We do not consider the modification of transition blocks here. Such modifications would usually require an extensive analysis of the module behaviour and therefore cannot be captured by relatively simple rules.

3.3.1 Modifying the from-, to-, and when-clauses

Modification of the from-clause of a transition t in fact means that some inputs that have been accepted in state s_t will now be accepted in a

different state. This is a specialization if the removal of t is a specialization, and the subsequent addition of the transition t' that differs from t in the from-clause only is a specialization, too. Therefore, we can define the modification of a from-clause in terms of removal and addition.

Modification of a to-clause of a transition means that some traces will lead to different states. It follows that the module instance will offer a different set of actions after each such trace. This is a specialization only if for each of these traces, the new set of offered actions specializes the original set of offered actions. In general, it has to be investigated on a case by case basis whether this condition is satisfied. Also, we can again take the previous approach and define this kind of modification in terms of addition and removal of a transition, which covers some possible cases.

Modification of the when-clause of a transition means that different inputs will be accepted by that transition afterwards. This is only a specialization if the domain coverage is not violated, and the undeterminism does not increase as a result. These conditions are checked by the operations $Remove_t$. and Add_t. Therefore, we can again define this kind of modification in terms of adding and removing a transition.

Definition 3.5: Let M be a module definition, t be a transition, s ∊ states(M), s' ∊ states(M), and i be an input.

a) $Modify_s(M,t,s) =_{Df}$ **if** Removal_OK **then** Add_t ($Remove_t$ (M,t),$t[s]_s$) **else** M

b) $Modify_{s'}(M,t,s') =_{Df}$ Add_t ($Remove_t$ (M,t),$t[s']_{s'}$)

c) $Modify_i(M,t,i) =_{Df}$ **if** Removal_OK **then** Add_t ($Remove_t$ (M,t),$t[i]_i$) **else** M

If t cannot be removed from M, then $t[s']_{s'}$ cannot be added to M. Therefore, in an additional condition as in a) and c) is not necessary in b).

Proposition 3.5:

a) $Modify_s(M,t,s) >_e M$

b) $Modify_{s'}(M,t,s') >_e M$

c) $Modify_i(M,t,i) >_e M$

Example 3.5:

a) Let C5 $=_{Df}$ $Modify_s$(C3',tr7,watching). Since the domain covered by tr7 is also covered by tr8, tr7 can be removed. Also, tr7[watching]$_s$ can be added.

b) Let C6 $=_{Df}$ $Modify_{s'}$(C5,tr7[watching]$_s$,watching). Since removal of tr7[watching]$_s$ would violate domain coverage, C6 = C5.

c) Let C7 $=_{Df}$ $Modify_i$(C3',tr7,tv.sports). Since the domain covered by tr7 is also covered by tr8, tr7 can be removed. Also, tr7[tv.sports]$_i$ can be added after removal of tr7.

3.3.2 Modifying the provided-clause

Modification of a provided-clause p can have the effect that the transition accepts different inputs in different module states with the same control state. We will consider those cases where the different inputs and module states are characterized by a provided-clause p' that is either stronger or weaker than p. In general, p and p' may be unrelated.

a) strengthening the provided-clause

If the provided-clause is strengthened, this can concern both the constraints on the parameter values and the module state. Thus, the modified transition will accept less inputs in less module states. This is a specialization only if the domain coverage is maintained, i.e., if there are other transitions that make up for the reduced domain of the modified transition in all affected module states. Stated otherwise, we have a specialization if the modification of the provided-clause reduces undeterminism. To capture this idea formally, we define a function $Modify_p(M,t,p)$ that changes the provided-clause of transition t of module M into p iff the above conditions are satisfied.

Definition 3.6: Let M be a module instance, t be a transition, p be a provided-clause, $Strengthening_OK = \forall fv.\ (p \supset p_t) \wedge \exists t'_1,\ldots,t'_n \in trans(M)-\{t\} \cup \{t[p]_p\}.\ (\forall j.\ (1 \leq j \leq n \supset s_{t'_j} = s_t \wedge i_{t'_j} = i_t) \wedge \forall fv.\ (p_t \supset \bigvee_{1 \leq j \leq n} p_{t'_j}))$.

$Modify_p(M,t,p) =_{Df}$ **if** Strengthening_OK
then (states(M), decl(M), ip(M), itrans(M), $trans(M)-\{t\} \cup \{t[p]_p\}$)
else M

Strengthening the provided-clause of a transition is very similar to the removal of a transition (compare Removal_OK and Strengthening_OK). In fact, we could comprehend t as two transitions t' and t" which are derived from t by replacing p_t by $p_{t'} = p$ and $p_{t''} = p_t \wedge \neg p$, respectively. Strengthening the provided-clause of t is then equivalent to the removal of t".

Proposition 3.6: $Modify_p(M,t,p) <_r M$

Example 3.6: Let $C8 =_{Df} Modify_p(C3',tr8,amount \geq 50)$. If a provided-clause is not specified for a transition, then according to the Estelle semantics, it is equivalent to "true". Therefore, we have p_{tr8} = true, which is implied by "amount ≥ 50" for all possible values of "amount". Also, "$\forall$amount. (amount ≤ 100 ∨ amount ≥ 50)" is equivalent to "true", which is implied by p_{tr8}. Therefore, the modification of the provided-clause of tr8 will lead to a specialization of C3'.

b) weakening the provided-clause

If the provided-clause is weakened, the modified transition will accept more inputs in more module states. This is a specialization in the sense of $>_e$ only if the undeterminism does not increase. Domain coverage is always maintained by this modification, since all inputs that were accepted before will be accepted afterwards. To capture this idea formally, we define a function $Modify'_p(M,t,p)$ that changes the

provided-clause of transition t of module M into p iff the above conditions are satisfied.

Definition 3.7: Let M be a module instance, t be a transition, p be a provided-clause, Weakening_OK = $\forall fv. (p_t \supset p) \wedge \forall t' \in trans(M)-\{t\}. (s_{t'} = s_t \wedge i_{t'} = i_t \supset \forall fv. \neg(p_{t'} \wedge p))$.

$Modify'_p(M,t,p) =_{Df}$ **if** Weakening_OK
then (states(M), decl(M), ip(M), itrans(M), $trans(M)-\{t\}\cup\{t[p]_p\}$)
else M

Weakening the provided-clause of a transition is very similar to the addition of a transition (compare Adding_OK and Weakening_OK). In fact, weakening of the provided-clause of t is equivalent to the addition of a transition t' that is derived from t by replacing p_t by $p \wedge \neg p_t$, i.e., $Modify'_p(M,t,p) = Add_t(M,t[p \wedge \neg p_t]_p)$.

Proposition 3.7: $Modify'_p(M,t,p) >_e M$

Example 3.7: Let C9 $=_{Df}$ $Modify'_p$(C8,tr7,amount ≤ 200). The constraint ∀amount. (amount ≤ 100 ⊃ amount ≤ 200) is satisfied. But since there are values for which ¬(amount ≥ 50 ∧ amount ≤ 200) does not hold, the modification cannot be applied. Therefore, C9 = C8.

3.4 Adding control states, declarations, and external interaction points

Control states can be added, since they do not affect the module's behaviour. Without the existence of suitable transitions, the new control states will be unreachable. Thus the actual extension of the behaviour is only prepared at this point, but will be effected when transition are added later on. For the addition of control states, we introduce the function Add_s.

Declarations can be added, if they do not interfere with existing declarations, because they do not affect the module instance's behaviour. Note that adding the declaration of a variable which has the same identifier than an already declared variable, but a different type, would result in a syntactically incorrect module definition. Again, the addition of declarations prepares the extension of behaviour, which can be achieved by the addition of transitions.

External interaction points can be added to a module description, because this does not affect the behaviour of the module instance. Since the addition of an external interaction point can be considered as the addition of a declaration, the previous restriction applies here, too. As before, this prepares the extension of behaviour, transitions added subsequently may accept inputs through the new interaction points.

Definition 3.8: Let M be a module definition, s be a state, d be a declaration, ip be an external interaction point, $Addition_d_OK = id(d) \notin id(decl(M))$, $Addition_{ip}_OK = id(ip) \notin id(ip(M))$.

a) $Add_s(M,s) =_{Df}$ (states(M)∪{s}, decl(M), ip(M), itrans(M), trans(M))

b) $Add_d(M,d) =_{Df}$ **if** $Addition_d_OK$
then (states(M), decl(M)∪{d}, ip(M), itrans(M), trans(M)) **else** M

c) $Add_{ip}(M,ip) =_{Df}$ **if** $Addition_{ip}_OK$
then (states(M), decl(M), ip(M)∪{ip}, itrans(M), trans(M)) **else** M

Proposition 3.8:

a) $Add_s(M,s) <_r M$

b) $Add_d(M,d) <_r M$

c) $Add_{ip}(M,ip) <_r M$

Add_s, Add_d, and Add_{ip} can be generalized to functions Add_S, Add_D, and Add_{IP} that define the addition of a set of states, a set of declarations, and a set of external interaction points in the obvious ways.

4 Fusion of module definitions

We have now reached the point where we can define the fusion of module definitions in terms of incremental specialization. We define fusion for a pair of module definitions. The result of the fusion specializes each single module definition, if the conditions for the fusion are satisfied. These conditions concern the applicability of the functions Add_D, Add_{IP}, or form part of the definition of Add_T.

Definition 4.1: Let M, M' be module definitions, Fusion_OK = istate(M) = istate(M') ∧ id(decl(M)) ∩ id(decl(M')) = {} ∧ id(ip(M)) ∩ id(ip(M')) = {}.

Fuse(M,M') $=_{Df}$ **if** Fusion_OK
then Add_T ($Replace_i$ (Add_{IP} (Add_D (Add_S (M,states(M')),
decl(M')),ip(M')), fuse(itrans(M), itrans(M'))),trans(M'))
else M

Informally, fuse(itrans(M),itrans(M')) returns a transition that is obtained by taking the to-clause of itrans(M), and by sequentially composing the transition blocks of itrans(M) and itrans(M'). If Fusion_OK is satisfied, then the to-clauses of itrans(M) and itrans(M') are identical, and the variable identifiers are disjoint. Therefore, fuse(itrans(M),itrans(M')) and fuse(itrans(M'), itrans(M)) are semantically equivalent, because both transitions lead to the same initial state. The function $Replace_i$ replaces the initialize transition of the module definition by the transition given as the second argument. Also, if Fusion_OK is satisfied, then states(M) ∩ states(M') ≠ {} holds.

```
channel home_delivery (customer,restaurant);
        by customer:       order, pay;
        by restaurant:     ring, salad, cheese, chicken;
module C'_type activity;
        ip door (customer) individual queue;
        end;
body C'_body for C'_type;
        state idle, hungry, eating;
        initialize to idle begin end;
        trans
                tr1': from idle to same when door.ring begin end;
                tr2': from idle to hungry when door.ring begin output
        door.order end;
                tr3': from hungry to eating when door.salad begin end;
                tr4': from hungry to eating when door.cheese begin
        end;
                tr5': from eating to idle begin output door.pay end;
        end;
```

Table 4.1: Estelle specification of another customer

Proposition 4.1:

a) Fuse(M,M') $>_e$ M
b) trans(Fuse(M,M')) = trans(M) $\cup$ trans(M') implies
Behav(Fuse(M,M')) = Behav(Fuse(M',M)) and
Fuse(M,M') = Fuse(M',M)[fuse(itrans(M), itrans(M'))]$_{itrans}$

Example 4.1: Let C' be the customer specified in Table 4.1. C' is connected to a restaurant through an Estelle channel "home_delivery". From time to time, the home delivery service rings at the door. The customer may then simply ignore the ringing, or he may order some food and suddenly start feeling hungry. The restaurant then supplies him with food and expects to get paid. From three possible dishes, the customer only accepts salad and cheese. Because he is vegetarian, he will block for chicken and starve. After he has finished eating, the customer will pay for his dish. We can now fuse C8 and C' using the function Fuse. The fusion will result in the Estelle module C" shown in Table 4.2. In this particular case, Fuse(C8,C') = Fuse(C',C8). Therefore, C" $>_e$ C8 and C" $>_e$ C'. Note that C8 and C' have one (initial) control state in common, but are otherwise disjoint behaviours.

5 Conclusions

We have presented a constructive approach for the incremental specialization of Estelle module definitions. The approach is of particular interest in the software maintenance phase, because it can reduce the

total effort of adding or modifying user requirements in some situations. As a starting point, we have assumed that system specifications are given as properties that constrain the set of traces and offered actions. These properties can be interpreted as behaviours, which have provided the formal basis for reduction and extension, two different notions of specialization. Since these notions take only the traces and - for any given trace - the sets of possible next actions into account, the room for specialization in terms of properties is limited. It would be interesting to take other kinds of properties into account. For instance, if new services are added to an existing one, it should always be possible to return to the original service after accessing new services. This case is not covered by our notions of specialization. With respect to Estelle, it would mean that it is always possible to return to the initial state.

Our general approach of specializing object behaviour on the operational level, outlined in the Introduction, can help to reduce software maintenance costs due to new or modified user requirements. Apart from Estelle, it can be considered in other formalisms as well, for instance, LOTOS, CSP, and SDL. In SDL, offered actions not only result from explicitly defined transitions, but also from implicit transitions due to the "ignoring by default" convention. This means that if we remove/add an explicit transition, an implicit transition will be automatically added/removed. This makes the treatment of specialization in SDL using the above notions of reduction and extension more difficult and limited. An option would be to consider weaker notions of specialization, for instance, constrainment, constrainment on the domain, or domain coverage.

```
channel cable (tv_station, tv_subscriber);
        by tv_station:          commercial, movie, news, sports, bill
        (amount: integer);
        by tv_subscriber:       on, off, switch, pay (amount: integer);

channel home_delivery (customer,restaurant);
        by customer:     order, pay;
        by restaurant:   ring, salad, cheese, chicken;

module C"_type activity;
        ip tv: cable (tv_subscriber) individual queue;
        ip door (customer) individual queue;
        end;

body C"_body for C"_type;
        state idle, watching, hungry, eating;
        initialize to idle begin end;
        trans
                tr1: from idle to watching begin output tv.on end;
                tr2: from watching to same when tv.commercial begin
        end;
                tr3: from watching to idle when tv.commercial begin
        tv.off end;
```

```
        tr4: from watching to idle begin tv.off end;
        tr5: from watching to same when tv.movie begin end;
        tr6: from watching to same when tv.news begin end;
        tr7: from idle to same when tv.bill provided amount ≤
100 begin end;
        tr8: from idle to same when tv.bill provided amount ≥
50
                begin output tv.pay(amount) end;
        tr1': from idle to same when door.ring begin end;
        tr2': from idle to hungry when door.ring begin output
door.order end;
        tr3': from hungry to eating when door.salad begin end;
        tr4': from hungry to eating when door.cheese begin
end;
        tr5': from eating to idle begin output door.pay end;
end;
```

Table 4.2: Estelle specification of a composed customer

Acknowledgement. The authors gratefully acknowledge several constructive comments from the referees.

References

[BoGo93] Bochmann, G.v., Gotzhein, R.: Specialization of Object Behaviors and Requirement Specifications, Publication No. 853, Departement d'IRO, Universite de Montreal, January 1993, 25p.

[BuDe87] Budkowski, S., Dembinski, P.: An Introduction to Estelle: A Specification Language for Distributed Systems, Computer Networks and ISDN Systems, Vol. 14, 1987

[Cer92] Cerny, E.: Verification of I/O Trace Set Inclusion for a Class of Non-deterministic Finite State Machines, ICCD'92 Conference, Cambridge, Mass., October 1992

[GoBo92] Gotzhein, R., Bochmann, G.v.: Specialization in Estelle, Publication #835, Département d'IRO, Université de Montréal, Montréal, Canada, September 1992, 23p.

[Hoa85] Hoare, C. A. R.: Communicating Sequential Processes, Englewood Cliffs, Prentice Hall, 1985

[ISO89] Estelle - A Formal Description Technique Based on an Extended State Transition Model, International Standardization Organization, IS 9074, 1989

[Saq91] Saqui-Sannes, P. d.: Comparison of Several Specification Languages with Regards to Stepwise Refinement and Specialization, Working Draft, CITR Project on Development of Prototype Specifications and Implementations, December 1991

[Sta72] Starke, P.H.: Abstract Automata, North-Holland, Amsterdam, 1972