Interner Bericht

A Case Study on Specification, Detection and Resolution of IN Feature Interactions with Estelle

Jan Bredereke, Reinhard Gotzhein

Nr. 245/94

May 1994

Fachbereich Informatik
Universität Kaiserslautern

Postfach 3049
D-67653 Kaiserslautern

E-mail: {bredereke,gotzhein}@informatik.uni-kl.de

Abstract

We present an approach for the treatment of Feature Interactions in Intelligent Networks. The approach is based on the formal description technique Estelle and consists of three steps. For the first step, a specification style supporting the integration of additional features into a basic service is introduced. As a result, feature integration is achieved by adding specification text, i.e. on a purely syntactical level. The second step is the detection of feature interactions resulting from the integration of additional features. A formal criterion is given that can be used for the automatic detection of a particular class of feature interactions. In the third step, previously detected feature interactions are resolved. An algorithm has been devised that allows the automatical incorporation of high-level design decisions into the formal specification. The presented approach is applied to the Basic Call Service and several supplementary interacting features.

1. INTRODUCTION

The evolution of telecommunication systems towards Intelligent Networks (IN) has lead to the development of a large number of features. By features, we refer to "packages of incrementally added functionality providing services to subscribers or the telephone administration" ([Bo+89]). This includes features such as call forwarding, three-way calling, automatic recall and billing. While features can be independent, it is often the case that they interfere or interact[1] in certain situations, i.e. the behaviour of a feature is altered by subscribing to another feature.

To ensure a reliable operation of the network, interference among features has to be detected and resolved before their deployment. However, the complexity of telecommunication systems — the number of deployed features ranging in the order of several hundreds ([Bo+89]) — and the fact that features are provided by multiple vendors have lead to a situation where detection and resolution of interferences is very difficult. This constitutes what has been termed *feature interaction problem*[2] in [Bo+89], and is considered a key obstacle to the development of INs.

The feature interaction problem has received growing attention in the scientific community recently. Several authors have proposed classifications of feature interactions in telecommunication systems, allowing to focus systematically on smaller portions of the problem (e.g. see [CGL+93, CCI92, DaNa93]). In [DaNa93], a distinction is made between *technical interference* and *policy interference*. Technical interference leads to phenomena such as deadlocks, livelocks and congestion. The detection of these phenomena has been studied extensively. Policy interference occurs if features interact such that the policy of one or more of the involved features is violated. Our approach is designed to be general, but we will especially concentrate on policy interference.

To tackle the feature interaction problem, the use of formal approaches during the design of features has been advocated ([CGL+93]). Formal specifications of features can provide a rigorous basis and the necessary insight for the detection and resolution of feature interactions. In our opinion, the following three aspects should be addressed by a formal approach:

1. Specification
   The specification should support the integration of additional features. This requires the selection of a suitable FDT and specification style.

2. Detection
   Formal criteria for the existence of feature interactions should be given that can be applied to the specification of features.

3. Resolution
   There should be a formal way of incorporating high-level design decisions about the resolution of feature interactions into the specification.

We emphasize that these aspects *interfere* and therefore have to be treated together. In

[1] we will use the terms "interfere" and "interact" interchangeably in this context
[2] also termed "service interference problem"

particular, the FDT and the specification style have a strong influence on the detection criteria and on the resolution of feature interactions.

An approach for the specification and detection of feature interactions based on LOTOS has been proposed in [DaNa93]. It consists of "a specification style well-suited for the purpose of detecting and resolving service interference", guidance for the specification of supplementary features, and guidance for the integration of two specifications. This addresses the first aspect of the previous list. The second and third aspect are also briefly addressed, but no formal criteria for the detection of feature interactions — apart from "ad-hoc rules and the skills of the designer/analyser" — are given.

The report is organized as follows. In Section 2, we present a specification style suitable for the detection and resolution of feature interactions based on the FDT Estelle ([ISO89]). We apply this style to the Basic Call Service and several supplementary features. In Section 3, we introduce a formal criterion for the detection of feature interactions, a technique for their resolution, and we outline possible tool support. Finally, we draw conclusions and give an outlook in Section 4. In the appendix, we include our complete Estelle specification of the Basic Call Service.

2. SPECIFICATION OF "IN" FEATURES WITH ESTELLE

Dahl and Najm give in [DaNa93, Sect. 2.2] four requirements for the specification style to support the detection of feature interactions. *Visibility of user/service interactions* is achieved in Estelle through channel definitions and explicit communication statements. *Compactness of constraints* (conditions) pertaining to the firing of user/service interactions is strongly supported in Estelle by transition clauses (`when`, `provided`, ...). *Constuctiveness*, tool support, is given for Estelle as well. *Extensibility* of the specification will be the subject of this section, with additional attention directed to the suitability for our feature interaction detection and resolution procedure discussed in Section 3.

2.1. Specification Style

Adding (further) features to a system means extending its specification. In order to facilitate the detection of feature interactions, the number of spots in the syntax and the semantics which are touched should be as small as possible. Modularity often makes it easier to cope with complexity. Therefore we propose a specific specification style which we will discuss and demonstrate in the remainder of this section. When the behavior of an Estelle specification is extended, the following stylistic rules will allow the use of our detection and resolution procedure in Section 3:

- Extend on a *coarse-grained* level.
- Modify only by *adding* text.

We will discuss these rules in detail now. We try to do extensions on a coarse-grained level. For Estelle modules, the coarsest grain of behavioral description is the transition. So, we work on the level of entire transitions only. Furthermore, we only add transitions. By this every possible execution sequence in the hitherto existing specification remains possible in the extended specification. The existing behaviour is not touched. Of course

there also may be cases where the existing behaviour needs to be changed. Then, we don't change the definition of the corresponding transition, but we disable[3] the old transition and add a new one. It is easier to detect problems resulting from a few such transition replacements than those from many small changes on the level of single Pascal-like statements inside the transition bodies.

Apart from the transitions, there is the state space which may need to be extended, too, to allow for more behaviour. Accordingly, we propose that there should be only modifications by additions. Applying this to the major state of an extended FSA is simple: as a syntactical rule, we only add more state identifiers, but don't remove any. Concerning the extended state space, it is safe to add new variables which have not been defined before. If it really should become necessary to change the meaning of a variable, we propose the use of a new variable instead, so that the change becomes explicit in every access. There is only one notable exception of the rule that the definition of variables and their types should not be altered: the case of record variables. Here, it is safe if we add more components to an already existing record type definition. These components will be accessed only in newly added transitions because of our rule of non-modification of existing transitions.

In the external interface of a module, there also should be only additions, so that existing behaviour will not be touched without adding new transitions. This concerns mainly the channel definitions containing the message[4] types which may be sent and received. There should only be new message types.

The use of the above two stylistic rules has an important consequence for the management of the set of features that may be added to the basic service specification. There is a simple way now to do this management. Mostly, we add large text blocks (several new transitions), sometimes we add text lines (new states, message types, variables). Therefore, it proved to be efficient and elegant to maintain both the basic specification and all additional features in one single text file[5] and to select the desired parts by a preprocessor using `#ifdef` and `#endif` directives, similar to the standard C language preprocessor. Only sometimes this is not sufficient when old transitions have to be disabled. The disabling of transitions is done in a different way, it will be described in Section 3.2.

After we presented our rule that transitions may only be added, we need to mention a special case where we make a single exception: the `initialize` transition which establishes the initial state of a module instance. Naturally, only one such transition may be fired. On the other hand, each feature may need to set up its own part of the extended state or it may need to create additional child modules and the corresponding communication structure. We could devise some mechanism to fire each element of a set of special transitions before the "normal" transitions become enabled. But this would not be worth the pain. It is possible that these initialization actions are related, so all of them have to be checked for interactions anyway. Our detection approach from Section 3.1 would not give helpful information in this special case, it would mark each of the special tran-

[3] To be exact, our method will leave the transition *enabled*, but not *fireable*.

[4] We denote Estelle interactions by "message" here to avoid confusion with the notion of feature interactions.

[5] Maybe partitioned into several text files and merged by an "include" directive to facilitate the handling and maintenance of a large specification, as usual.

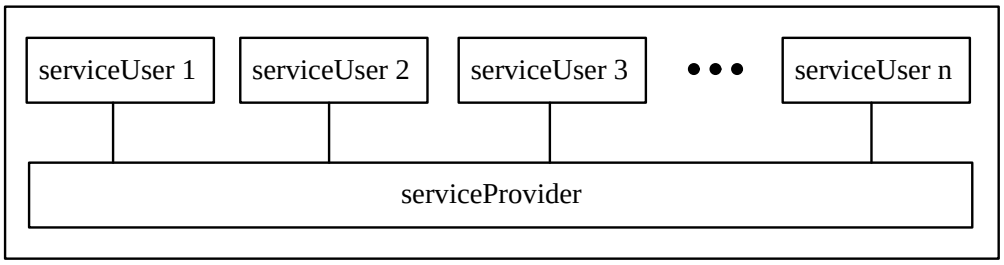

Figure 1. The call service architecture.

sitions as possibly interacting. Therefore, we just concatenate all of the actions into one single initialization transition and then we check this one transition manually for possible interactions.

Besides the rules of specification style discussed above, there are of course more general rules of style which facilitate the detection and resolution of feature interactions. It is important to use them, too, but they are no prerequisite for the application of our detection and resolution procedure in Section 3. For example it is advisable to use naming conventions that provide information on the meaning of an identifier that is as specific as possible. If the name is too general, it will change its intuitive meaning when put into a different context, e.g. is referred to in a new feature. Sometimes, it may be a good idea to pre- or postfix an identifier with the related feature. This prevents name clashes when features are added.

2.2. The Basic Call Service

The Basic Call Service itself is well known. Each end user may access it through his end user equipment, which is identified by a unique number. He may take the receiver off the hook or put it on the hook again; he may dial and send a number of a callee to the provider, for to be connected; he may hear different signal tones or, if everything went well, the words which his interlocutor speaks. The service architecture can be found in Figure 1.

To keep the example simple, we remain on the service level, i.e., the service provider shows no internal structure, just like in the example in [DaNa93]. A deeper study will include the distribution aspect, too, so that feature interactions depending on the distribution can be investigated. Nevertheless, even this simple example demonstrates several feature interactions. As we will see, our approach for detection and resolution scales well to larger systems, so it may be used afterwards with them as well.

The Estelle module structure in our example specification is exactly like in Figure 1. There is only one kind of communication channel, which is defined as follows:

```
channel basicCallService(user,provider);                              (1)
by user:
  putOnHook; takeOffHook; call(calledUser: serviceUserNumType);
  {putOn}    {off}
by provider:
```

```
  freeTone; tryConnectTone; busyTone; ringTone;
            {tryConTone}
by user,provider:
  blaBla;
```

At a specific moment of time, each end user equipment is in a certain state. The BCS provider has to keep track of these states since its actions may depend upon it (e.g. sending a free tone to some of these end user equipments). In a later refinement step there will be substructure in the service provider, e.g. one child module instance per end user, and it will be most appropriate to use an Estelle major `state`. But since we chose a higher abstraction level in this example, we define a homogenous state space based on an array of records:

```
type                                                                    (2)
  userInfoType = record
    userState: userStateType;
    otherParty: serviceUserNumTypeWithNull;
  end;
  userInfoArrayType = array[serviceUserNumType] of userInfoType;
var
  userInfo: userInfoArrayType;
```

In this, the `userStateType` is defined as follows:

```
type                                                                    (3)
  userStateType =
      (onHook,
       offHook,
       busyNotification,              {busy}
       tryConnectNotification,        {tryCon}
       outgoingConnection,            {oCon}
       incomingConnection,            {iCon}
       onHookStillConnection,         {onHkCon}
       partnerOnHookStillConnection, {pOnHkCon}
       busyConnectionEnded,           {conEnd}
       ringNotification,              {ringing}
       waitingForProvider             {w4prov} );
```

Based on these messages and states, the Estelle transitions define the behaviour of the BCS provider. A finite state automaton derived from these transitions for one end user is presented in Figure 2, which shows only the received and sent Estelle messages[6]. In the Estelle specification, the transition corresponding to the arc leading from `offHook` to `tryCon` by receiving `call` looks like:

[6]We describe the European BCS, which is slightly different from the American BCS.

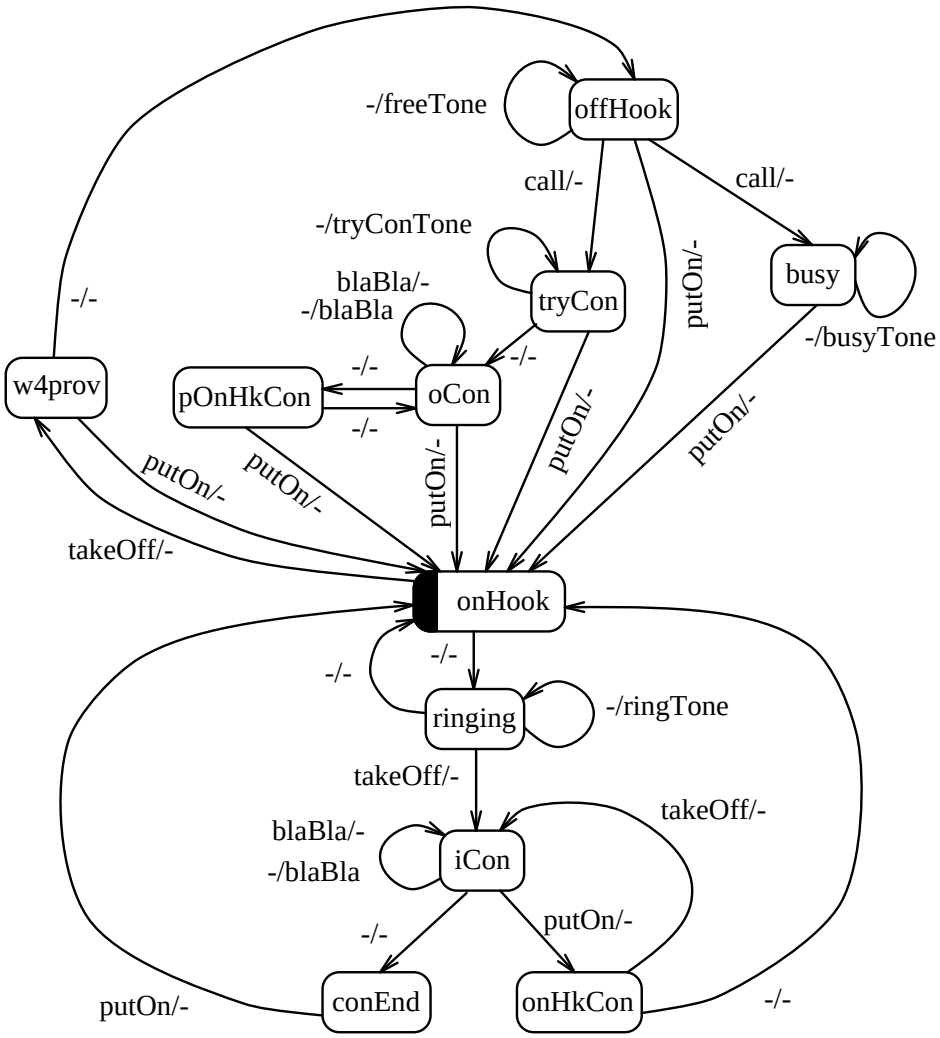

Figure 2. A simplified finite state automaton of the BCS provider.

```
trans                                                                    (4)
  any userIdx: serviceUserNumType do
    when client[userIdx].call(calledUser)
      provided (userInfo[userIdx].userState = offHook)
           and (userInfo[calledUser].userState = onHook)
        begin
          userInfo[userIdx].userState := tryConnectNotification;
          userInfo[userIdx].otherParty := calledUser;
          userInfo[calledUser].userState := ringNotification;
          userInfo[calledUser].otherParty := userIdx;
        end;
```

As can be seen, this Estelle transition also covers the arc from `onHook` to `ringing` (`ringNotification`) in Figure 2. (In this monolithic service specification, there are no

independent FSAs, and the "FSA" in Figure 2 is just an abstraction for a single end user which was derived to support the understanding.)

The complete Estelle specification of the Basic Call Service may be found in the appendix.

2.3. Adding Selective Call Rejection — Terminating Side

A feature that can be added to the Basic Call Service is *Selective Call Rejection — Terminating Side* (SCRT). It's policy is as follows: a user who has subscribed to this feature maintains a black list of originating users from whom he never wishes to be called. If they should try, they will receive a special rejection tone.

We add this feature in the way as discussed in Section 2.1. The state space has to be extended to include one black list per user. This is done by adding the following component to the record definition in text block (2):

```
#ifdef SCRT                                                    (5)
scrtBlackList: set of serviceUserNumType;
#endif SCRT
```

Also, we need one more major state `rej` (`rejectNotification`) to model the case when a call is being rejected. To reduce the complexity of the change, we only add (or disable) entire transitions. We start adding a transition from the state `offHook` to the new state `rej` which receives a `call` message (see text block 6 below). In those cases when it is enabled, the old BCS transitions from `offHook` to `tryCon` and to `busy` have to be disabled. These are now only activated for the cases when the caller is not on the black list. (How the firing conditions of the old transitions are reduced to fewer cases will be discussed in Section 3.2.)

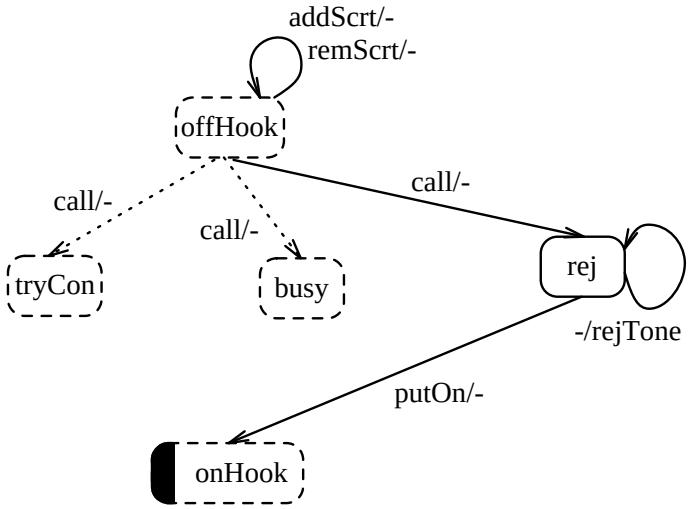

Figure 3. Adding Selective Call Rejection — Terminating Side.

```
#ifdef SCRT                                                          (6)
trans
  any userIdx: serviceUserNumType do
      when client[userIdx].call(calledUser)
        provided (userInfo[userIdx].userState = offHook)
             and (userIdx in userInfo[calledUser].scrtBlackList)
          begin
            userInfo[userIdx].userState := rejectNotification;
            userInfo[userIdx].otherParty := calledUser;
          end;
....
#endif SCRT
```

Next, we have to cover the case when a caller has been rejected and receives the corresponding message. For this, we add one more message and two more transitions. (Figure 3 shows the arcs and the state `rej` that have to be added to Figure 2.) Finally, two further messages and two transitions have to be added in order to allow the user to maintain his black list.

Note that all extensions are done on a coarse scale, adding chunks of specification text most of the time, and, on the syntactical level, only adding more items to pre-existing lists. As has been said in the beginning of Section 2, the management of the specification text may be done elegantly by a preprocessor which selects the appropriate parts from a single text file using `#ifdef` directives.

2.4. Adding Further Selected Features

There are more features which may be added to the Basic Call Service. The second feature is *Call Forwarding* (CF). It's policy is as follows: it allows a user to redirect any call directed at his own end user equipment to any other end user equipment, e.g. because he leaves his home or bureau to visit the user of this other equipment. In the specification, similar extensions have to be made like in the case of SCRT. The state record has to be extended by a further component to hold the number to forward to. Two new messages and two transitions have to be added to allow the user to set a call forward or to reset it. And, of course, we need a transition to do the actual call forwarding. It will disable and thereby replace the BCS call transition in text block (4):

```
#ifdef CF                                                            (7)
trans
  any userIdx: serviceUserNumType do
    when client[userIdx].call(calledUser)
      provided (userInfo[userIdx].userState = offHook)
           and totalFnctCheckState(
                    callForwardDestination(calledUser), onHook)
        begin
          userInfo[userIdx].userState := tryConnectNotification;
          userInfo[userIdx].otherParty :=
              callForwardDestination(calledUser);
```

```
            userInfo[callForwardDestination(calledUser)
                    ].userState := ringNotification;
            userInfo[callForwardDestination(calledUser)
                    ].otherParty := userIdx;
          end;
....
#endif CF
```

A similar adjustment is necessary for the situation when the callee is busy. We introduced two auxiliary functions: `callForwardDestination` was introduced because we wanted the CF feature to follow possible chains of call forwardings, which is done by this function. `totalFnctCheckState` was needed because there may be a loop in a forwarding chain. The first function signals this by a special error value[7], which has been added to the range of the function. Because of this error value, accessing the `userInfo` array in the `provided` clause (for comparison of the state component) becomes a partial function. This access function is made total again by the second function, it returns `false` in the error case. To handle such error conditions, we have to add another error state and the corresponding transitions, similar to the "rejected" state `rej` of the SCRT feature (c.f. Figure 3).

The third feature which we present should be added to any practically usable specification. It takes care that the provider ignores any inappropriate user input (SAFE feature). This can be done by adding one transition for each possible message, and they should be enabled if no other transitions are enabled[8]. E.g. dialing a number while the receiver is on hook should not lead to a deadlock resulting from an unspecified reception. Such deadlocks are very useful in the testing phase to detect any unspecified reception which should not be there, but they cannot be called userfriendly when the system is deployed. Notice that these explicit ignore-transitions, which may be turned on and off, are more flexible than "ignoring by default".

Fourth, as a refinement of the SAFE feature, one can also add a runtime error detection feature (Log end User eQuipment Errors — LUQE). It will detect errors which are not due to a sloppy user but may indicate hardware problems in the end user equipment. An example is the situation when the message is received twice that the receiver is hung on the hook, without that it is taken off inbetween.

The complete Estelle specification of all additional features discussed in this report may be found in the appendix.

3. DETECTION AND RESOLUTION OF FEATURE INTERACTIONS WITH ESTELLE

An obvious approach would be to give criteria for avoiding feature interactions. If, for instance, features can only be used exclusively, i.e., if one feature is in operation, then all other features are disabled, then no interactions can occur[9]. However, in many

[7] Loop detection may be done, e.g., by setting a limit on the number of hops.
[8] A simple way to achieve this is described in Section 3.2.
[9] if "disabling" is not considered a feature interaction

cases, feature interactions are wanted. For instance, if Call Forwarding (CF) and Selective Call Rejection — Terminating Side (SCRT) are among the installed features, then feature interactions are unavoidable. The policy of CF demands a call to be delivered and the policy of SCRT excludes certain calls from delivery. What can be avoided is that the subscriber will be surprised every time he uses these features simultaneously by defining which feature will take precedence over the other. Determining a precedence between different interacting features thus will be a typical technique for resolving feature interactions, making the behaviour more predictable for the subscriber.

In Section 3.1 we will present a technique to detect in advance possible feature interactions automatically; and in Section 3.2 we will describe how a resolution step may be supported so that essentially only the still missing high level design decisions are added manually. In Section 3.3 we will discuss tool support.

3.1. Detection of Feature Interactions

As a consequence of using the specification style presented in Section 2, we can give a necessary condition for feature interactions to occur:

> Rule:
> A feature interaction may occur only if there is a state in which at least two transitions belonging to different features can be fireable[10] simultaneously, i.e. if there is non-determinism among different features.

This non-determinism results from adding transitions of different features independently. For orthogonality, we also denote the basic service as a feature. Obviously, if we add another feature, whose transitions are not enabled in any state which the basic service can reach, there will be no feature interaction (because the second feature will do nothing at all). Similarly, the above rule applies also between further features.

It is interesting to have a criterion for the cases where feature interactions cannot occur. But even more interesting are the cases where they can. And the above rule may help us then, too: it provides us with a relatively short list of critical spots. If we investigate each of them and resolve the non-determinism (as described in Section 3.2), then no unwanted feature interaction will be left.

It has to be stressed that only non-determinism between the transitions of *different* features is an indication of possible feature interactions. Non-determinism among the transitions of a single feature is a common mean to express abstraction and has nothing to do with our topic. Therefore, we have to differentiate between these two kinds of non-determinism. This is done by

> an unambiguous association of each transition to one feature.

On the syntactical level, we do this by a provision which we will present in Section 3.2.

The attentive reader will already have recognised several possibly interacting transitions in our example in Section 2. The new transition of the SCRT feature which processes the `call` message in the `offHook` state (see Figure 3) competes non-deterministically with

[10] "Enabled" would not be sufficient because it does not take into account priorities and delays.

the three transitions from the CF feature to process the same message in the same sate. And all of them compete with the two of the Basic Call Service[11]. On the other hand, all other transitions of the SCRT and CF features have no chance to interact with each other or with the Basic Call Service because they either receive messages which are not known to the other features or origin from states which uniquely belong to their feature. So, the resolution step can concentrate on these six transitions only (from a total of 40 transitions for all features of our example).

As an exception, we get a long interaction list if we specify the SAFE feature from Section 2.4 simply by letting it discard *any* message from the user. The resolution of these interactions, of course, is absolutely trivial: in any collision case, the SAFE transition has to be disabled (for how this is done see Section 3.2).

If we strive to *avoid* feature interactions as far as possible in the first place, the above rule gives a helpful hint, too. We have to separate the messages and states of the different features as far as possible. This corresponds to the well known rule that the concept of modularity reduces side effects and isolates the spots where they still may occur.

3.2. Resolution of Feature Interactions

One technique to resolve feature interactions is to define, for each pair of features, which one will take precedence over the other. This step can be seen as a high-level design decision that requires background knowledge about policies of the telephone administration, and therefore can not be automated. The results of this step can be represented in the form of a precedence matrix, with an entry for each pair of features. An example for such a precedence is that the SCRT transitions override any BCS transitions. These, in turn, override the SAFE transitions. Sometimes, parts of two colliding features have to be combined, as in the case of SCRT and CF. Screening should work also for forwarded calls. Such combination of feature parts can be done by adding another "feature" (named SCRT&CF in our example) with transitions that just cover these cases and which have a higher precedence than both of the colliding features. In the SCRT and CF example, the SCRT `call` processing transition should be disabled because it doesn't follow the call forwarding chain when it checks the callee's black list. So we need all the CF transitions plus one transition (SCRT&CF) which does the same job as the old screening transition, but on forwarding chains:

```
#ifdef CF                                                          (8)
....
#ifdef SCRT
trans
  any userIdx: serviceUserNumType do
    when client[userIdx].call(calledUser)
      provided (userInfo[userIdx].userState = offHook)
           and totalFnctInScrtBlackList(
                   callForwardDestination(calledUser), userIdx)
        begin
          userInfo[userIdx].userState := rejectNotification;
```

[11] If they are not overridden by the CF transitions as suggested in Section 2.4.

```
            userInfo[userIdx].otherParty :=
              callForwardDestination(calledUser);
          end;
#endif SCRT
#endif CF
```

Table 1 shows a precedence matrix for all the features discussed so far. If there is an entry for (f, f'), then f takes precedence over f'. Entries (f, f') are omitted if no precedence is defined between f and f', or if f' takes precedence over f. Entries may be omitted if they can be derived from the transitive closure (note that $\succ$ is transitive). This reduces the number of necessary entries in case of a total ordering from $\frac{n^2-n}{2}$ down to $n-1$ (less in all other cases).

Table 1
Precedence matrix

	BCS	SCRT	CF	SCRT&CF	SAFE	LUQE
BCS					$\succ$	
SCRT	$\succ$				$\succ$	
CF	$\succ$	$\succ$			$\succ$	
SCRT&CF	$\succ$	$\succ$	$\succ$		$\succ$	
SAFE						
LUQE					$\succ$	

The resolution of feature interactions as defined by a precedence matrix of dimension n (like the one in Table 1) can be incorporated into the Estelle specification in three steps:

Step 1. From the precedence matrix, derive a precedence function prec associating a precedence value with each feature such that $f \succ f'$ implies $\text{prec}(f) < \text{prec}(f')$[12].

Step 2. Associate each transition of the Estelle specification with one feature.

Step 3. For all transitions t: if t is associated with feature f, then add a priority clause "`priority` f_p", where $f_p = \text{prec}(f)$.

The association between transitions and features from Step 2 should also be done syntactically, according to the following provision: for each feature, one constant identifier is defined. In Step 3 we will need a priority constant for each feature anyway, so we will define one[13] priority constant per feature (e.g. `bcsPriority`, `scrtPriority`, `cfPriority`, ...). This way, the priority clause from Step 3 will provide us with the syntactical association between transitions and features. An example will follow below.

[12] In Estelle, smaller values denote higher priorities. We will use the value of the precedence function as the Estelle priority below.

[13] If more than one priority value per feature should be needed, the method can be extended straightforwardly.

Based on a precedence matrix with dimension n, a precedence function prec can be determined as follows:

1. For all features f: $\mathrm{prec}(f) = 0$ is the initial value of the precedence function.
2. Check every pair (f, f') of features whether the required relation between f and f' according to the precedence matrix is satisfied, i.e. whether $f \succ f'$ implies $\mathrm{prec}(f) < \mathrm{prec}(f')$ or whether there is no entry for (f, f').

   Yes: Stop; the precedence function has been successfully determined.

   No: select one such pair and increase the value of $\mathrm{prec}(f')$ to the smallest natural number such that the relation is satisfied.
3. If for all features f, the current value of $\mathrm{prec}(f)$ is less than n, continue with 2; otherwise, stop, there is a cycle, and no precedence function exists.

Calculation of the precedence function for the precedence matrix in Table 1 yields: prec(BCS) = 3, prec(SCRT) = 2, prec(CF) = 1, prec(SCRT&CF) = 0, prec(SAFE) = 4, prec(LUQE) = 0. In Estelle, it will look as follows:

```
const                                                              (9)
  bcsPriority       = 3;
  scrtPriority      = 2;
  cfPriority        = 1;
  scrtAndCfPriority = 0;
  luqePriority      = 0;
  safePriority      = 4;
....
trans
  priority bcsPriority  { <-- new ************************* }
    any userIdx: serviceUserNumType do
      when client[userIdx].call(calledUser)
        provided (userInfo[userIdx].userState = offHook)
             and (userInfo[calledUser].userState = onHook)
          begin .... end;
#ifdef SCRT
trans
  priority scrtPriority  { <-- new ************************** }
    any userIdx: serviceUserNumType do
        when client[userIdx].call(calledUser)
          provided (userInfo[userIdx].userState = offHook)
               and (userIdx in userInfo[calledUser].scrtBlackList)
            begin .... end;
#endif SCRT
```

The incorporation of the precedence function by the definition text for the priority constant values gives us an advantage. In this place, we may easily include the output of a tool which computes the precedence function automatically from the precedence matrix.

The consequence of the modification according to Steps 1 to 3 is that non-determinism due to feature interaction is removed, i.e., there is no uncertainty about which behaviour can be expected by the service user. This may lead to transitions that are no longer executable, because their firing condition is completely overlapped by the firing condition of other transitions. In a subsequent optimization step, this "dead code" may be removed without altering the specified behaviour.

In many cases, it will not be necessary to define the precedence matrix completely. Based on the results of the feature interaction detection step, it is sufficient to define a precedence relation between interacting features, i.e. between features with transitions that can be enabled simultaneously. In the examples in Section 2 this is the case for transitions accepting a call in state `offHook`, i.e. for features BCS, CF, SCRT, and for all transitions where unexpected input can be discarded, i.e. for all combinations of features BCS, CF, SCRT with SAFE.

It may be observed in our example that it is possible that several features are assigned the same priority (SCRT&CF, LUQE). This may happen if, according to our rule in Section 3.1, it is impossible that these features may interact, because there is no chance for a nondeterministic choice. In our example, the error conditions which the LUQE transitions look for are completely disjoint from the state/message condition under which the SCRT&CF transition processes a `call` message.

Note that if it is necessary that we add transitions to resolve a detected feature interaction (like in the SCRT&CF case), this is done formally like the addition of any other feature. This implies that the new "feature" may interact with any other feature, too. Therefore, the (automated) detection procedure from Section 3.1 has to be repeated. But this does *not* mean that the resolution step has to be redone from the beginning, too. On the contrary, we keep everything achieved up to now. E.g. if we have introduced additional precedences between features, they will not be lost.

3.3. Tool support

As has been discussed in Section 3.2, it is not feasible to automate the entire process of detection and resolution. But nevertheless these steps may be supported by tools for an interactive treatment. The following tools are already in use:

- A *preprocessor* to extract the selected features from the possible set of features as discussed in Section 2.1 and 2.3. We implemented a filter similar to the standard C language preprocessor (but dropping, e.g., macro preprocessing).

- An *animation* tool for simulation purposes (Pet/Dingo, see [SiSt93]), including a graphical user interface. We have extended this interface by provisions to directly send and receive all messages defined for an end user equipment. Figure 4 shows the window for end user equipment number 2. The boxes in the last line are highlighted each time the corresponding message is received, and the buttons and pull-down menus in the last but one line may be used to send arbitrary messages.

- A tool to *compute priorities* from a precedence matrix as discussed in Section 3.2.

The following tools are under development:

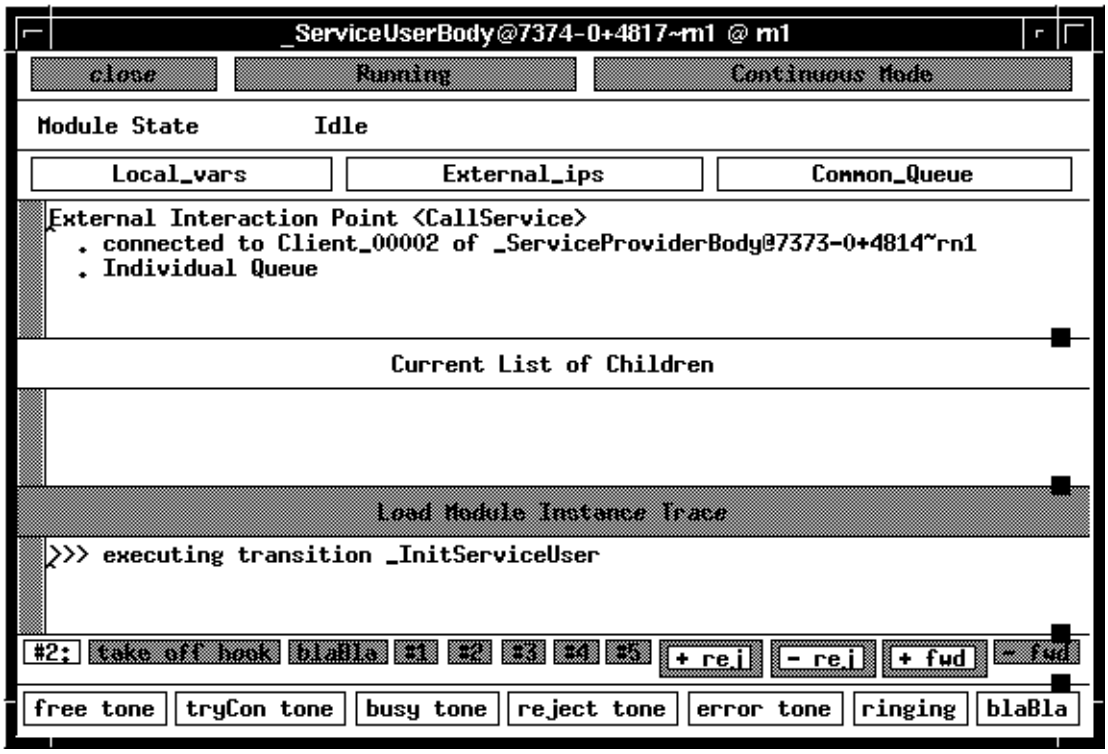

Figure 4. The graphical interface for one end user generated by the Pet/Dingo toolset, and our extensions.

- A tool to *detect non-determinism* among features (c.f. Section 3.1).
- A tool to *detect non-executable transitions*. It will allow to optimize the result of the resolution step (c.f. Section 3.2).
- A tool for the *automatic derivation of the SAFE feature*. The addition of a feature introducing a new message requires the addition of a new ignore transition to the SAFE feature. This can be automated easily[14] (c.f. Section 2.4 and 3.1).

We expect that we can present first versions of these tools by the end of 1994. We expect that they will support the development of modular IN systems substantially and will reduce the manual share in treating the feature interaction problem to high-level design decisions; they will have to be made for the cases which the detection tool indicated.

4. CONCLUSIONS

We have presented an approach to specifying, detecting and resolving IN Feature Interactions with Estelle. A specification style supporting the integration of additional features into a basic service is introduced. Based on that specification style, we have given a formal

[14]The SAFE feature could even be reduced to a single transition based on the Estelle language extension of `interactionsets` proposed in [Ten90].

criterion to detect especially those feature interactions classified as policy interference. A specification can be checked automatically against the criterion. This detection serves as a starting point for subsequent high-level design decisions about the resolution of policy interference. The incorporation of these decisions into the Estelle specification and thus the resolution of policy interference can be automated, too. We have devised an algorithm for this design step. We applied our approach to the Basic Call Service and several supplementary features.

We point out that our detection criterion is simple enough to be calculated by tools quickly even for large specifications; the same applies to the algorithm we use in the resolution step. Therefore, our approach scales well to the complexity of practical systems.

Some future work remains to be done. The incorporation of further features into the Basic Call Service is currently under investigation. The refinement from the service to a protocol is an obviously promising step to go, introducing more structure into the system architecture (c.f. Figure 1). Besides the very helpful tools which already exist, as discussed in Section 3.3, some important tools (e.g. for the automation of the detection step) are still under construction. Also, it should be investigated how our approach marks with respect to the classes of further feature interaction classifications (c.f. Section 1). Since the Estelle language extension discussed in [Ten90] seems to prove valuable, it may be promising to investigate some more Estelle language extensions.

Addition of features and thus extension of the Basic Call Service (BCS) has been achieved by adding specification text, i.e., on a purely syntactical level. This has lead to potential non-determinism, which has been removed in a subsequent resolution step. We expect that it will be possible to develop a theoretical foundation together with suitable (syntactical) rules for incremental specialization such that the following holds: if the specification of the BCS augmented by additional features is an incremental specialization of the BCS, then it also extends the BCS semantically. Some work in this direction has been reported in [GoBo94], where semantical relations for specializing Estelle module definitions and a notion of incremental specialization based on these relations have been introduced.

REFERENCES

[Bo+89] Bowen, T. F. et al.. *The feature interaction problem in telecommunication systems*. In "Seventh IEEE International Conference on Software Engineering for Telecommunication Systems" (July 1989).

[CCI92] CCITT. *Principles of Intelligent Network Architecture, Draft Recommendation Q.1201* (1992).

[CGL+93] Cameron, E. J., Griffeth, N., Lin, Y.-J., Nilson, M. E., Schnure, W. K., and Velthuijsen, H. *A feature interaction benchmark in IN and beyond*. IEEE Communications Magazine pp. 64–69 (Mar. 1993).

[DaNa93] Dahl, O. C. and Najm, E. *Specification & detection of IN service interference using LOTOS*. In Tenney, R. L., Amer, P. D., and Uyar, M. Ü., editors, "Sixth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols — FORTE '93", Boston, Mass. (26–29 Oct. 1993). North-Holland.

[GoBo94] Gotzhein, R. and von Bochmann, G. *Specialization in Estelle.* In Vuong, S. T. and Chanson, S. T., editors, "Proceedings of the 14. International IFIP Symposium on Protocol Specification, Testing and Verification — PSTV '94", Vancouver, Canada (7–10 June 1994). To appear.

[ISO89] ISO/TC 97/SC 21, ISO 9074. *Information Processing Systems — Open Systems Interconnection — Estelle: A Formal Description Technique Based on an Extended State Transition Model* (1989).

[SiSt93] Sijelmassi, R. and Strausser, B. *The PET and DINGO tools for deriving distributed implementations from Estelle.* Comp. Networks and ISDN Systems **25**(7), 841–851 (Feb. 1993).

[Ten90] Tenney, R. L. *Adding interaction sets to Estelle.* In Quemada, J., Maños, J., and Vazquez, E., editors, "Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols — FORTE '90", pp. 477–483, Madrid, Spain (5–8 Nov. 1990). North-Holland.

Appendix

```
{
  Specification of the Basic (Telephone) Call Service in Estelle

#ifdef SCRT
  ===== New feature: "scrt" =====
  This service is extended in that it allows a (s)elective (c)all (r)ejection
  at the (t)erminating side. A user has a "black list" of other users
  from which he never wishes to be called.
  Modifications of the black list may be done by special feature keys
  on the phone, but only in the off hook state when the free tone can
  be heard.
  ==============================

#endif SCRT
#ifdef CF
  ===== New feature: "cf" =======
  This service is extended in that it allows a (c)all (f)orwarding.
  A user may specify another user to whom any call will be forwarded.
  Setting and Resetting this forwarding number may be done by
  special feature keys on the phone, but only in the off hook state
  when the free tone can be heard.

  Possibility of interaction whith other features:
  The notion of "originating/calling user" is messed up somehow. Now, you
  have to differentiate between the user who forwarded to you
  and the user who is making a call (and all the other intermediate
  users).
  ==============================

#endif CF
#ifdef LUQE
  ===== New feature: "luqe" =====
  This service is extended in that it detects and ignores any messages
  from the end user equipment which are "impossible", e.g., if the user
  seems to on-hook his phone twice without off-hooking it inbetween.
  (L)og (u)ser e(q)uipment (e)rrors.
  If such things are detected, a notice is written to a maintenance log
  which could be checked regularly by a maintenance person.
  ==============================

#endif LUQE
#ifdef SAFE
  ===== New feature: "safe" =====
  This service is extended in that it ignores safely any unexpected
  user input, e.g., talking while no line is established, dialing
  while line is established, etc.
  Also ignored is dialing after taking the hook off, before you have got
  the free tone.
  (Warning: Your specification won't complain anymore if you just
  forgot to specify a transition!)
  If this feature is used, it is recommended that the "luqe" feature
  to handle unexpected input resulting from user equipment failures
```

```
  should be used, too. Otherwise, they would be ignored too and could go
  unnoticed, since there is no "detection by deadlock" anymore.
  ==============================

#endif SAFE
  File: basiccall.e

  Author: Jan Bredereke

  Date: 15.2.1994

  Last modified: 25.4.1994

}
specification bcsEnvironment;
  default individual queue;
  timescale seconds;

  { Relative priorities of the bcs transitions and those of added features: }
  { (smaller values mean higher priorities, "0" ist highest) }
  const
    bcsPriority = 100;                { Priority of the bcs transitions. }
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
    scrtPriority = 90;                { Priority of the "service call
                                        rejection - terminating side"-
                                        transitions. }
{ ============================== }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
    cfPriority = 80;                  { Priority of the "call forwarding"-
                                        transitions. }
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
    scrtAndCfPriority = 75;           { Priority of the transitions which
                                        resolve a feature interaction
                                        between "service call rejection -
                                        terminating side" and "call
                                        forwarding". }
{ ============================== }
#endif SCRT
{ ============================== }
#endif CF
#ifdef LUQE
{ ===== New feature: "luqe" ===== }
    luqePriority = bcsPriority;       { Priority for detecting, logging and
                                        ignoring user equipment errors. }
{ ============================== }
#endif LUQE
#ifdef SAFE
{ ===== New feature: "safe" ===== }
    safePriority = 1000;              { (Very low) priority for ignoring
                                        unexpected interactions from user. }
```

```
{ ============================== }
#endif SAFE
    { (Please add priority values for new features before this line.) }

  { Other constants: }
  const
    maxServiceUserNum = any integer;  { Max. possible number of subscribed
                                        users, may be as high as desired, but
                                        finite.}
    maxResourceNum = any integer;     { Max. number of lines that may be served
                                        simultaneously by the resources of the
                                        network. }
    nullServiceUser = 0;              { Null value for service user number. }
    ringToneRepeat  = 4;              { Delay time between repeated ring
                                        tones. }
    tryConToneRepeat = ringToneRepeat; { Delay time between repeated
                                        notifications that other party is
                                        ringing. }
    busyToneRepeat  = 3;              { Delay time between repeated busy
                                        tones. }
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
    rejectToneRepeat = 3;             { Delay time between repeated reject
                                        tones. }
{ ============================== }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
    errorToneRepeat = 3;              { Delay time between repeated error
                                        tones. }
{ ============================== }
#endif CF
    freeToneRepeat  = 2;              { Delay time between repeating the
                                        prompt to the end user equipment
                                        for generating another period of
                                        "continuous" free tone. }
#ifdef CF
{ ===== New feature: "cf" ======= }
    cfMaxHops = 5;                    { Max. forwarding hops allowed.
                                        This is to prevent infinite loops,
                                        which are an interaction of the
                                        "cf" feature with itself. }
{ ============================== }
#endif CF
  type
    serviceUserNumType         = 1..maxServiceUserNum;
                                             { Range type for user array. }
    serviceUserNumTypeWithNull = nullServiceUser..maxServiceUserNum;
                                             { Same, with null value. }

  channel basicCallService(user,provider);
  by user:
    {putOn}   putOnHook;              { user hangs phone on hook }
    {off}     takeOffHook;            { user picks phone off hook }
```

```
    call(calledUser: serviceUserNumType);
                                    { user dials a number }
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
    {addScrt} addToScrtBlackList(refusedUser: serviceUserNumType);
                                    { user doesn't want to be called anymore
                                      from refusedUser }
    {remSrct} removeFromScrtBlackList(refusedUser: serviceUserNumType);
                                    { user cancels scrt black list entry }
{ ============================== }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
    {setCf}   setCallForward(forwardUser: serviceUserNumType);
                                    { user wants to forward any call
                                      to forwardUser }
    {resetCf} resetCallForward;
                                    { user cancels call forwarding }
{ ============================== }
#endif CF
  by provider:
    freeTone;                       { provider sends free tone }
    {tryConTone} tryConnectTone;    { provider sends notification tone that
                                      other phone is ringing }
    busyTone;                       { provider sends busy tone }
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
    {rejTone} rejectTone;           { provider sends tone that call is
                                      rejected }
{ ============================== }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
    {errTone} errorTone;            { provider sends tone that some error was
                                      detected, e.g., a call forwarding loop }
{ ============================== }
#endif CF
    ringTone;                       { phone rings }
  by user,provider:
    blaBla;                         { user speaks into phone or
                                      phone replays the words }

  module serviceUserType systemprocess(userIdx: serviceUserNumType);
  { (The "userIdx" is provided mainly for animation purposes.) }
  ip callService: basicCallService(user);
  end; { of module serviceUserType }

  body serviceUserBody for serviceUserType;
#include 'serviceUserBody.e'
    { external; }

  module serviceProviderType systemprocess;
  ip client: array[serviceUserNumType] of basicCallService(provider);
  end; { of module serviceProviderType }
```

```
  body serviceProviderBody for serviceProviderType;
    type
      userStateType =  { States which an end-user may be in (from the
                         provider's point of view): }
          (onHook,                          { Phone hangs on hook, no
                                              association with other user. }
           offHook,                         { Phone is picked up, free tone
                                              is received, user may dial a
                                              number. No association with
                                              other user. }
           {busy}    busyNotification,      { Caller receives busy notification
                                              because callee is not in state
                                              onHook. }
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
           {rej}     rejectNotification,    { Caller receives reject
                                              notification because callee has
                                              put him on his scrt black list. }
{ ============================== }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
           {error}   errorNotification,     { User receives error notification
                                              because some error was detected,
                                              e.g., a call forwarding loop }
{ ============================== }
#endif CF
           {tryCon}  tryConnectNotification, { User has put a call and receives
                                              a notification tone that callee
                                              receives ring tone.
                                              Field "otherParty" now contains
                                              number of callee. }
           {oCon}    outgoingConnection,    { A caller who has an established
                                              voice-band connection with a
                                              callee. }
           {iCon}    incomingConnection,    { A callee who has an established
                                              voice-band connection with a
                                              caller. }
           {onHkCon} onHookStillConnection, { A callee has hung phone following
                                              an incomingConnection,
                                              conversation may be resumed if
                                              he picks up the phone again. }
           {pOnHkCon} partnerOnHookStillConnection, { Same situation, state
                                              for corresponding calling
                                              partner.
                                              The definition of this state is
                                              somewhat redundant, it could be
                                              merged with "oCon".  But this
                                              would imply that some
                                              transitions not only check the
                                              state of the local phone but
                                              also of the remote partner.  In
                                              this global state oriented
```

```
                                          specification style, this would
                                          be no problem, but a later
                                          refinement into a set of local
                                          FSA would require the splitting
                                          of "oCon" into "oCon" and
                                          "pOnHkCon" again.
                                          And, finally, from a conceptual
                                          point of view, we are NOT in
                                          an "outgoing connection" state
                                          when the callee has hung up
                                          the phone, since we cannot
                                          talk to anybody. }
          {conEnd}  busyConnectionEnded,  { Phone is still off hook after
                                          a connection, but there is no
                                          association. Going onHook is
                                          necessary before further calling
                                          or receiving may take place. }
          {ringing} ringNotification,     { A Callee receives indication that
                                          a caller wants a conversation. }
          {w4prov}  waitingForProvider);  { User is off hook but receives
                                          no tone until the network has
                                          sufficient resources to treat
                                          his requests. }
      userInfoType = record
        userState: userStateType;
        otherParty: serviceUserNumTypeWithNull;
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
        scrtBlackList: set of serviceUserNumType;
{ ============================== }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
        forwardUser: serviceUserNumTypeWithNull;
{ ============================== }
#endif CF
      end;
      userInfoArrayType = array[serviceUserNumType] of userInfoType;
    var
      userInfo: userInfoArrayType;
      currentServiceLoad: integer; { How many lines are currently served. }

#ifdef CF
{ ===== New feature: "cf" ======= }
    { Auxiliary functions:

    { Simply check if a user is in a certain state.
      But this is a TOTAL FUNCTION on serviceUserNumTypeWithNull, that
      means it doesn't crash if the userIdx is null.
      In this case, it just returns false. }
    function totalFnctCheckState(userIdx: serviceUserNumTypeWithNull;
                                 checkState: userStateType;
                                 { Sorry, only necessary because of a
                                   Pet/Dingo compiler bug: }
```

```
                                userInfo: userInfoArrayType):
        boolean;
    begin
      if userIdx = nullServiceUser then
        totalFnctCheckState := false
      else
        totalFnctCheckState := (userInfo[userIdx].userState = checkState);
    end;

#ifdef SCRT
{ ===== New feature: "scrt" ===== }

    { Simply check if a user is in a certain state.
      But this is a TOTAL FUNCTION on serviceUserNumTypeWithNull, that
      means it doesn't crash if the userIdx is null.
      In this case, it just returns false. }
    function totalFnctInScrtBlackList(userIdx: serviceUserNumTypeWithNull;
                                checkIdx: serviceUserNumType;
                                { Sorry, only necessary because of a
                                  Pet/Dingo compiler bug: }
                                userInfo: userInfoArrayType):
        boolean;
    begin
      if userIdx = nullServiceUser then
        totalFnctInScrtBlackList := false
      else
        totalFnctInScrtBlackList := checkIdx in userInfo[userIdx].scrtBlackList;
    end;

{ ============================== }
#endif SCRT
    { Follow the call forwarding pointer chain starting at calledUser
      and return the final destination.
      If no forwarding is set, calledUser itself is returned.
      If the forwarding chain is too long (or infinite),
      return nullServiceUser. }
    function callForwardDestination(calledUser: serviceUserNumType;
                                    { Sorry, only necessary because of a
                                      Pet/Dingo compiler bug: }
                                    userInfo: userInfoArrayType):
        serviceUserNumTypeWithNull;
    var
      fwdCnt: integer; { Counter for call forwarding hops. }
      tmpUser: serviceUserNumTypeWithNull; { temporary pointer }
    begin
      { Search iteratively for user to forward to. }
      fwdCnt := 0;
      tmpUser := calledUser;
      while (userInfo[tmpUser].forwardUser <> nullServiceUser)
            and (fwdCnt < cfMaxHops) do
      begin
        tmpUser := userInfo[tmpUser].forwardUser;
        fwdCnt := succ(fwdCnt);
      end;
```

```
      if fwdCnt > cfMaxHops then
        callForwardDestination := nullServiceUser
      else
        callForwardDestination := tmpUser;
    end;
{ ================================ }

#endif CF
#ifdef LUQE
{ ===== New feature: "luqe" ===== }
    { Auxiliary function: }

    type
      string35 = packed array[1..35] of char;

    { Log a user equipment error for the maintenance person. }
    procedure logUserEquipmentError(errText: string35);
      primitive;
{ ================================ }

#endif LUQE
    {********** Transition part **********}
    initialize
name trServiceProviderInit:
      begin
        all userIdx: serviceUserNumType do
        begin
          userInfo[userIdx].otherParty:=nullServiceUser;
          userInfo[userIdx].userState:=onHook;
#ifdef SCRT
{ ===== New feature: "scrt" ===== }
          userInfo[userIdx].scrtBlackList:=[];
{ ================================ }
#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
          userInfo[userIdx].forwardUser:=nullServiceUser;
{ ================================ }
#endif CF
        end;
        currentServiceLoad := 0;
      end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].takeOffHook
            provided userInfo[userIdx].userState = onHook
name trTakeOff_onHook:
              begin
                userInfo[userIdx].userState  := waitingForProvider;
                userInfo[userIdx].otherParty := nullServiceUser;
              end;
```

```
    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          provided (userInfo[userIdx].userState = waitingForProvider)
                   and (currentServiceLoad < maxResourceNum)
name trResource_waitingForProvider:
            begin
              userInfo[userIdx].userState  := offHook;
              userInfo[userIdx].otherParty := nullServiceUser;
              currentServiceLoad := succ(currentServiceLoad);
            end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].takeOffHook
            provided userInfo[userIdx].userState = ringNotification
name trTakeOff_ringNotification:
              begin
                userInfo[userIdx].userState := incomingConnection;
                { (Variable "userInfo[userIdx].otherParty" was set by
                  transition "trCall_offHook_OnHook".) }
                userInfo[userInfo[userIdx].otherParty].userState :=
                    outgoingConnection;
                userInfo[userInfo[userIdx].otherParty].otherparty := userIdx;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].takeOffHook
            provided userInfo[userIdx].userState = onHookStillConnection
name trTakeOff_onHookStillConnection:
              begin
                userInfo[userIdx].userState := incomingConnection;
                userInfo[userInfo[userIdx].otherParty].userState :=
                    outgoingConnection;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and (userInfo[calledUser].userState = onHook)
name trCall_offHook_OnHook:
              begin
                userInfo[userIdx].userState := tryConnectNotification;
                userInfo[userIdx].otherParty := calledUser;
                userInfo[calledUser].userState := ringNotification;
                userInfo[calledUser].otherParty := userIdx;
              end;

    trans
```

```
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and (userInfo[calledUser].userState <> onHook)
name trCall_offHook_NotOnHook:
              begin
                userInfo[userIdx].userState := busyNotification;
                userInfo[userIdx].otherParty := calledUser;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = incomingConnection
name trPutOn_incomingConnection:
              begin
                userInfo[userIdx].userState := onHookStillConnection;
                userInfo[userInfo[userIdx].otherParty].userState :=
                    partnerOnHookStillConnection;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = outgoingConnection
name trPutOn_outgoingConnection:
              begin
                userInfo[userInfo[userIdx].otherParty].userState :=
                    busyConnectionEnded;
                userInfo[userInfo[userIdx].otherParty].otherparty := userIdx;
                userInfo[userIdx].userState := onHook;
                userInfo[userIdx].otherParty := nullServiceUser;
              end;


    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = partnerOnHookStillConnection
name trPutOn_partnerOnHookStillConnection:
              begin
                userInfo[userIdx].userState := onHook;
                userInfo[userInfo[userIdx].otherParty].userState :=
                    onHook;
                userInfo[userInfo[userIdx].otherParty].otherparty :=
                    nullServiceUser;
                userInfo[userIdx].otherParty := nullServiceUser;
                currentServiceLoad := pred(currentServiceLoad);
              end;

    trans
```

```
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided (userInfo[userIdx].userState = busyNotification)
                  or (userInfo[userIdx].userState = busyConnectionEnded)
                  or (userInfo[userIdx].userState = offHook)
name trPutOn_busyOrOffHook:
              begin
                userInfo[userIdx].userState := onHook;
                userInfo[userIdx].otherParty := nullServiceUser;
                currentServiceLoad := pred(currentServiceLoad);
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = waitingForProvider
name trPutOn_waitingForProvider:
              begin
                userInfo[userIdx].userState := onHook;
                userInfo[userIdx].otherParty := nullServiceUser;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = tryConnectNotification
name trPutOn_tryConNotification:
              begin
                userInfo[userIdx].userState := onHook;
                userInfo[userInfo[userIdx].otherParty].userState := onHook;
                userInfo[userInfo[userIdx].otherParty].otherparty :=
                    nullServiceUser;
                userInfo[userIdx].otherParty := nullServiceUser;
                currentServiceLoad := pred(currentServiceLoad);
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          provided userInfo[userIdx].userState = offHook
            delay(freeToneRepeat)
name trDelay_offHook:
              begin
                output client[userIdx].freeTone;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          provided userInfo[userIdx].userState = ringNotification
            delay(ringToneRepeat)
```

```
name trDelay_ringNotification:
              begin
                output client[userIdx].ringTone;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          provided userInfo[userIdx].userState = tryConnectNotification
            delay(tryConToneRepeat)
name trDelay_tryConNotification:
              begin
                output client[userIdx].tryConnectTone;
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].blaBla
          provided (userInfo[userIdx].userState = outgoingConnection)
                   or (userInfo[userIdx].userState = incomingConnection)
name trBlaBla_outgoingOrIncoming:
              begin
                output client[userInfo[userIdx].otherParty].blaBla;
                { In the LOTOS spec., there is no correlation
                  between spoken and heard "blaBla". }
              end;

    trans
      priority bcsPriority
        any userIdx: serviceUserNumType do
          provided userInfo[userIdx].userState = busyNotification
            delay(busyToneRepeat)
name trDelay_busyNotification:
              begin
                output client[userIdx].busyTone;
              end;
#ifdef SCRT

{ ===== New feature: "scrt" ===== }

    { Entering the parameters of the new feature:
      adding to and removing from the black list. }

    { Adding to the black list. }
    trans
      priority scrtPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].addToScrtBlackList(refusedUser)
            provided userInfo[userIdx].userState = offHook
name trAddScrt_offHook:
              begin
                userInfo[userIdx].scrtBlackList :=
                    userInfo[userIdx].scrtBlackList + [refusedUser];
```

```
              end;

    { Removing from the black list. }
    trans
      priority scrtPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].removeFromScrtBlackList(refusedUser)
            provided userInfo[userIdx].userState = offHook
name trRemScrt_offHook:
              begin
                userInfo[userIdx].scrtBlackList :=
                    userInfo[userIdx].scrtBlackList - [refusedUser];
              end;

    { Actual new behaviour. }

    { Entry point into new behaviour: }
    { This transition overrides the transitions trCall_offHook_OnHook
      and trCall_offHook_NotOnHook, if it is enabled. }
    trans
      priority scrtPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and (userIdx in userInfo[calledUser].scrtBlackList)
name trCall_offHook_scrtRejected:
              begin
                userInfo[userIdx].userState := rejectNotification;
                userInfo[userIdx].otherParty := calledUser;
              end;

    trans
      priority scrtPriority
        any userIdx: serviceUserNumType do
          provided userInfo[userIdx].userState = rejectNotification
            delay(rejectToneRepeat)
name trDelay_rejectNotification:
              begin
                output client[userIdx].rejectTone;
              end;

    { This transition is a direct extension of the transition
      trPutOn_busyOrOffHook to the new interaction.
      It serves to exit from the new behaviour. }
    trans
      priority scrtPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = rejectNotification
name trPutOn_rejectNotification:
              begin
                userInfo[userIdx].userState := onHook;
                userInfo[userIdx].otherParty := nullServiceUser;
                currentServiceLoad := pred(currentServiceLoad);
```

```
              end;
{ ============================== }
#endif SCRT
#ifdef CF

{ ===== New feature: "cf" ======= }

    { Entering the parameters of the new feature:
      setting and resetting the forwarding number. }

    { Setting the forwarding number. }
    trans
      priority cfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].setCallForward(forwardUser)
            provided userInfo[userIdx].userState = offHook
name trSetCf_offHook:
              begin
                userInfo[userIdx].forwardUser := forwardUser;
              end;

    { Resetting the forwarding number. }
    trans
      priority cfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].resetCallForward
            provided userInfo[userIdx].userState = offHook
name trResetCf_offHook:
              begin
                userInfo[userIdx].forwardUser := nullServiceUser;
              end;

    { Actual new behaviour. }

    { Entry points into new behaviour: }
    { This transition and the transition trCall_offHook_OnHook_callForward_fail
      override the transition trCall_offHook_OnHook }
    trans
      priority cfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and totalFnctCheckState(
                         callForwardDestination(calledUser, userInfo),
                         onHook,
                         userInfo)
name trCall_offHook_OnHook_callForward_success:
              begin
                userInfo[userIdx].userState := tryConnectNotification;
                userInfo[userIdx].otherParty :=
                    callForwardDestination(calledUser, userInfo);
                userInfo[callForwardDestination(calledUser,
                                                userInfo)].userState :=
                    ringNotification;
```

```
              userInfo[callForwardDestination(calledUser,
                                              userInfo)].otherParty :=
                  userIdx;
            end;
#ifdef SCRT

{ ===== New feature: "scrt" ===== }

    { Resolution of a feature interaction between "cf" and "scrt". }
    { First, the call forward destination is determined. Then,
      the black list of this destination is applied to screen out
      the original caller, if he is in the list.
      This prevents you from being called by a person on the black
      list because he uses a detour through another phone.
      Using the opposite order of rules would do a weaker thing,
      it would just protect you from forwarding the unwanted calls.
      But we could make the screening still stronger: We could use
      both rules simultaneously and enable each intermediate phone
      plus the receiving phone to screen out the original caller.
      This could be done by passing on the number of the original
      caller in each step.
      If we would take the pain of passing on ALL numbers on the path,
      we could screen them also.
      But anyway, the formerly strong call forwarding feature gets weaker
      and weaker. }

    { This transition overrides the transitions
      trCall_offHook_scrtRejected, trCall_offHook_OnHook_callForward_success
      trCall_offHook_callForward_fail and trCall_offHook_notOnHook_callForward,
      if it is enabled. }
    trans
      priority scrtAndCfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and totalFnctInScrtBlackList(
                         callForwardDestination(calledUser, userInfo),
                         userIdx,
                         userInfo)
name trCall_offHook_scrtRejected_withCf:
            begin
              userInfo[userIdx].userState := rejectNotification;
              userInfo[userIdx].otherParty :=
                callForwardDestination(calledUser, userInfo);
            end;

{ ============================== }
#endif SCRT

    { This transition partially overrides the transition
      trCall_offHook_OnHook. If it is enabled, it also overrides
      the transition trCall_offHook_NotOnHook, since forwarding shall work
      even if the forwarding user is using his phone otherwise. }
    trans
```

```
      priority cfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and (callForwardDestination(calledUser,
                                             userInfo) = nullServiceUser)
name trCall_offHook_callForward_fail:
              begin
                { Too many call forwarding hops! }
                userInfo[userIdx].userState := errorNotification;
                userInfo[userIdx].otherParty := calledUser;
              end;

    { This transition and the previous one override the transition
      trCall_offHook_NotOnHook }
    trans
      priority cfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
            provided (userInfo[userIdx].userState = offHook)
                 and (callForwardDestination(calledUser,
                                             userInfo) <> nullServiceUser)
                 and not totalFnctCheckState(
                             callForwardDestination(calledUser, userInfo),
                             onHook,
                             userInfo)
name trCall_offHook_notOnHook_callForward:
              begin
                userInfo[userIdx].userState := busyNotification;
                userInfo[userIdx].otherParty := calledUser;
              end;

    { Entirely new behaviour: }

    trans
      priority cfPriority
        any userIdx: serviceUserNumType do
          provided userInfo[userIdx].userState = errorNotification
            delay(errorToneRepeat)
name trDelay_errorNotification:
              begin
                output client[userIdx].errorTone;
              end;

    { This transition is a direct extension of the transition
      trPutOn_busyOrOffHook to the new interaction.
      It serves to exit from the new behaviour. }
    trans
      priority cfPriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = errorNotification
name trPutOn_errorNotification:
              begin
```

```
                userInfo[userIdx].userState := onHook;
                userInfo[userIdx].otherParty := nullServiceUser;
                currentServiceLoad := pred(currentServiceLoad);
              end;
{ ============================== }
#endif CF
#ifdef LUQE
{ ===== New feature: "luqe" ===== }
    { Detect and ignore any messages from the end user equipment
      which are "impossible". }

    trans
      priority luqePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
            provided userInfo[userIdx].userState = onHook
name trPutOn_logUserEquipmentError:
              begin
                logUserEquipmentError('<putOnHook> while <onHook>         ');
              end;

    trans
      priority luqePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].takeOffHook
            provided not (userInfo[userIdx].userState
                          in [onHook, ringNotification, onHookStillConnection])
name trTakeOff_logUserEquipmentError:
              begin
                logUserEquipmentError('<takeOffHook> while not on hook    ');
              end;
{ ============================== }

#endif LUQE
#ifdef SAFE

{ ===== New feature: "safe" ===== }
    { Ignore any unexpected reception of interactions from user. }
    { (Warning: Your specification won't complain anymore if you just
      forgot to specify a transition!) }

    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].putOnHook
name trPutOn_ignore:
            begin
            end;

    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].takeOffHook
name trTakeOff_ignore:
```

```
            begin
            end;

    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].call(calledUser)
name trCall_ignore:
            begin
            end;

    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].blaBla
name trBlaBla_ignore:
            begin
            end;

#ifdef SCRT
{ ===== New feature: "scrt" ===== }
    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].addToScrtBlackList(refusedUser)
name trAddScrt_ignore:
            begin
            end;

    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].removeFromScrtBlackList(refusedUser)
name trRemScrt_ignore:
            begin
            end;
{ ============================== }

#endif SCRT
#ifdef CF
{ ===== New feature: "cf" ======= }
    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].setCallForward(forwardUser)
name trSetCf_ignore:
            begin
            end;

    trans
      priority safePriority
        any userIdx: serviceUserNumType do
          when client[userIdx].resetCallForward
name trResetCf_ignore:
```

```
            begin
            end;
{ ============================== }

#endif CF
{ Insert more transitions here if you add more interactions from user! }

{ Comment: this really should be done by some "interactionset" construct
  and a SINGLE transition. Then, there would be no danger of missing
  a channel extension. }
{ ============================== }
#endif SAFE
  end; { of body serviceProviderBody }

  modvar { of specification }
    serviceUser: array[serviceUserNumType] of serviceUserType;
    serviceProvider: serviceProviderType;

  initialize { specification }
name trBcsEnvironmentInit:
    begin
      init serviceProvider with serviceProviderBody {@local};
      all userIdx: serviceUserNumType do
      begin
        init serviceUser[userIdx] with serviceUserBody(userIdx) {@local};
        connect serviceUser[userIdx].callService to
                serviceProvider.client[userIdx];
      end;
    end;

end. { of specification bcsEnvironment }
```