Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

Daniel Amyot
High Level Design and Prototyping of Agent Systems
Department of Systems and Computer Engineering, Carleton University
email: damyot@csi.uottawa.ca

Abstract. *Functional scenarios describing system views, uses, or services are a common way of capturing requirements of telecommunication systems. However, integrating individual descriptions of telephony features in different ways may result in different kinds of unexpected or undesirable interactions. Appropriate integration techniques can hopefully lead to fewer such interactions. In this report, we first present how a collection of features integrated visually through causal scenarios called Use Case Maps (UCMs) may help generating high-level LOTOS specifications. Integrating UCMs together helps avoiding trivial and artificial interactions before any prototype is generated. Then, we use the powerful testing concepts and tools of LOTOS to detect remaining undesirable interactions. To illustrate these concepts, we capture and validate a subset of the telephony features from the First Feature Interaction Contest. We discuss the results of this experiment, as well as strengths and weaknesses of our methodology.*

Key words. *Causal scenario, feature interaction, integration, LOTOS, specification-level validation, testing, Use Case Maps.*

1 INTRODUCTION

A *feature* is a collection of services packaged together that can be commercialized. Undesirable interactions between features still represent nowadays a complex problem that telecommunication systems designers must face [18][31], and this situation is likely to remain challenging in the future. By definition, features interact with each other and with the basic system services, the so-called *Plain Old Telephone System* (POTS). However, a feature might be prevented from working properly according to its intent because of some unexpected interactions with other features in the system. This is at the heart of the feature interaction (FI) problem. Similar challenges can be found in the agent community where agent goals might be conflicting and impossible to fulfil simultaneously [16][24]. For the last decade, many partial solutions have been suggested to avoid, detect, analyze, and solve feature interactions at design time and run time. Our proposal is one of avoidance at design time, and one of detection at design time with the help of an executable prototype. Avoidance of trivial interactions is achieved through the visual integration of scenarios expressed with the *Use Case Map* (UCM) notation [10][17]. Detection is done by using a process algebra, the *Language Of Temporal Ordering Specification* (LOTOS) [26] in our case, and formal V&V techniques.

LOTOS has been used for years for the specification and validation of telephony systems ([7][19][20][23]) and for the detection of interactions between telephony features ([21][22] [31][32][38][39][40][41]). Research is still ongoing as to its application to real-size problems. Use cases were utilized for the analysis of interactions in [33]. More recently, UCMs have also been used to tackle the problems of feature interactions and resolution of conflicts in multi-agent systems ([11][12][13][14][15][16]). The UCM notation helps designers with the visualization of problematic situations and their avoidance at a high level of abstraction. An approach where UCMs are transformed into LOTOS specifications and test cases has been applied to a number of examples in the areas of distributed systems and telephony ([2][3][4][5][6]).

With such knowledge and experience available, a methodology that would make use of the best features of UCMs (e.g., visual description and integration of features) and LOTOS (e.g., powerful theory and tools for validation and verification) for the avoidance and detection of feature interactions in telephony systems seems a natural evolution. Herein, we use such an approach (Section 2), and we illustrate it using some of the features described in the first feature interaction contest [25]. We present UCMs for selected features in Section 3. These UCMs were captured and integrated by Petriu in [35]. We discuss the synthesis and the validation of the LOTOS specification in the following section (Section 4). When integrating UCM scenarios (features) together, some trivial interactions can be avoided. However, for the remaining undesirable interactions, we use traditional LOTOS techniques and tools (Section 5). We discuss this methodology with three other approaches in Section 6 and then provide general conclusions.

2 METHODOLOGY

2.1 Rigorous Approach Based on Scenarios

We believe that the usage of UCMs in a scenario-based approach represents a judicious choice for the description and the design of reactive and distributed systems. Scenarios fit well in approaches that intend to bridge the gap between (informal) requirements and the first system design.

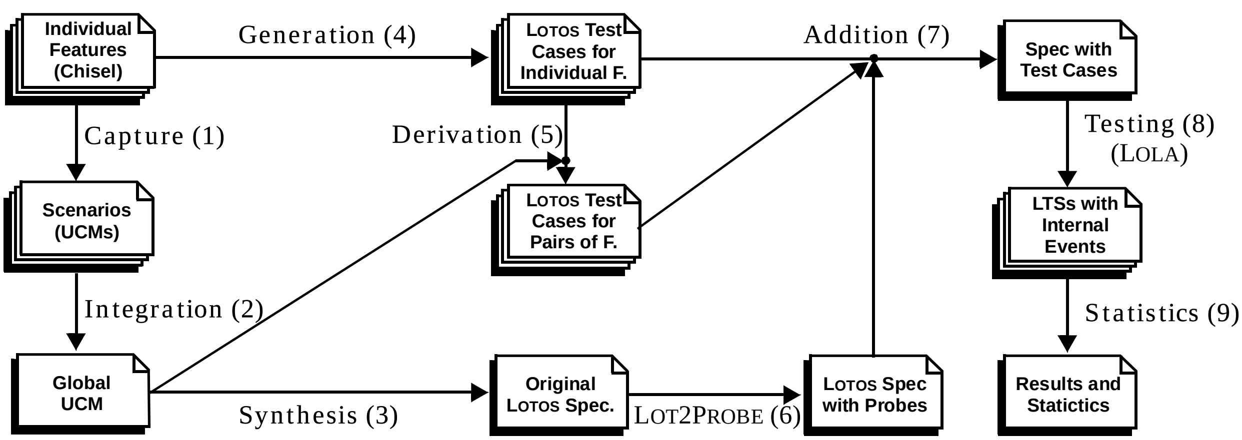


Figure 1 Scenario-Based Approach Used in this Experiment

Figure 1 introduces a scenario-based approach for designing telephony systems that are free of undesirable feature interactions. It is adapted from a more generic and rigorous approach discussed in [4][6]. We observed several advantages to this rigorous approach, the most important being related to the separation of the functionalities from the underlying structure, fast prototyping, test cases generation, and documentation of the requirements and of the high-level design.

In our case study, the start point is a collection of individual features described as Chisel diagrams [1]. Each feature is then captured as a Use Case Map (1). In the literature, this phase is often referred to as *scenario elicitation*, although in our case the requirements were already in the form of operational scenarios. The responsibilities in the UCMs are bound to components in the underlying structure, which is common for all scenarios in this specific example. UCMs can then be integrated together to produce a global UCM that covers all cases (2). Sequential, alternative, and parallel composition[1] are used as integration operators, as well as more subtle abstraction and composition mechanisms that make use of stubs and plug-ins. Once the global UCM is available, it can be used to synthesize a LOTOS specification, which becomes the executable prototype (3).

Concurrently with these steps, validation test cases can be generated from the Chisel diagrams (4) to ensure that the specification conforms to POTS and to each individual feature, when only one is active at a time. We can create further test cases, built on top of the test cases for individual features, in order to detect undesirable interactions between pairs of features integrated according to the global UCM (5). All test cases are described in the same language as the specification, i.e., LOTOS.

Probes can be inserted in the specification in order to measure how much of the structure of the specification is covered by the test suite and to ensure that the whole specification has been exercised by at least one test case (6). The new specification then contains the probes, to which we add the test cases for individual features and those for pairs of features (7).

Once the specification has been tested against all the test cases (8), results and statistics (9) can be obtained from the resulting trees (*Labeled Transition Systems* — LTSs). One of the following verdicts will occur:

- At least one test case from the individual feature set has failed. Since it does not work properly on its own, the specification of this feature has been incorrectly synthesized from the global UCM, or this UCM does not conform to the Chisel diagram. In the latter case, the capture or the integration of this scenario might be the cause.
- At least one test case from the feature interaction set has failed. The specification of the two features involved is incorrect w.r.t. their integration in the global UCM, or there is a *feature interaction*, i.e., an unforeseen and undesirable result.
- At least one probe has not been visited by the entire test suite. Some part of the specification is unreachable, or the test suite is incomplete and does not cover a case that the specification considers, or the specification covers a case that should not be considered.
- The test suite has passed successfully, and all probes have been covered. The specification conforms to the requirements (Chisel diagrams), and no feature interaction was detected. We then have a good level of confidence in the global UCM, in the LOTOS specification, and in the test suite.

Following the verdict, modifications may be required to the UCMs, to the test cases, and/or to the specification. In fact, the approach of Figure 1 is iterative. It is also incremental as new features may be integrated at a later time.

1. Composition is a much overloaded term. In this report, we use *integration* when we refer to the process of merging several UCM scenarios, while we use *composition* to represent the different constructs used in such integration. Composition refers also to the way plugins are linked together in a stub, and to the way LOTOS concurrent processes interact with each other.

3 USE CASE MAPS FOR FEATURES

3.1 Use Case Maps in a Nutshell

UCMs are a visual notation we utilize for capturing the requirements of reactive and distributed systems. They describe scenarios in terms of *causal relationships* between *responsibilities*. UCMs put emphasis on the most relevant, interesting, and critical functionalities of the system. They can have internal activities as well as external ones. Usually, UCMs are abstract (generic), and could include multiple traces (called *routes*). With UCMs, scenarios are expressed above the level of messages exchanged between components, hence they are not necessarily bound to a specific underlying structure. They provide a path-centric view of system functionalities and improve the level of reusability of scenarios.

Figure 2 shows a simple UCM where a user (U_1) tries to establish a connection with another user (U_2) through some network. U_1 first sends a connection request (**R**) to the network. The latter verifies (**V**) whether or not the called party is free. If she is, then there will be some status update (**F**) and a ring signal (**S**) will be activated on U_2's side. Otherwise, the network status will be updated differently (**O**) and a message stating that U_2 is not available (**M**) will be sent back to U_1.

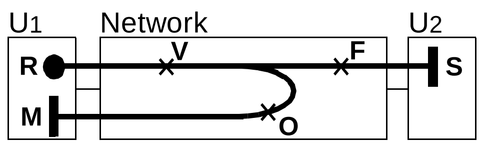


Figure 2 Simple Connection UCM

A scenario starts with a triggering event or a precondition (filled circle labeled **R**) and ends with one or more resulting events or postconditions (bars), in our case **S** and **M**. Intermediate responsibilities (**V**, **F**, **O**) have been activated along the way. In this picture, the activities are allocated to abstract components (U_1, U_2, **Network**). The notation allows for alternative paths (the fork in the figure), concurrent paths, and for explicit synchronous/asynchronous interactions between paths. For a detailed description of the notation, refer to [17].

The construction of a UCM can be done in many ways. Usually, one starts by identifying the activities that are to be performed by the system. They can then be allocated to scenarios and/or to components. Components can be discovered along the way. Eventually, the two views are merged to form a *bound UCM*, like the one in Figure 2.

3.2 Overview of the FI Contest Content

In the FI contest description (see [25]), a network was modeled as a collection of black boxes communicating with each other via defined interfaces. Definitions for the POTS service and the twelve features were given as sequences of (synchronous) events taking place on these interfaces. Interactions were to be detected between pairs of features.

Network Structure

The left half of Figure 3 shows that the network consists of end-user equipment (telephones A, B, and C), a switch, a Service Control Point (SCP) that processes IN features [28], an Operations System (OS) that does billing, and a global clock (not on the figure). The network interfaces are the interface between a user and the switch (on which the telephone is used for signaling); the interface between the switch and the SCP (on which IN messages are used); and the interface to the billing system (for tracking the beginning and end of each call).

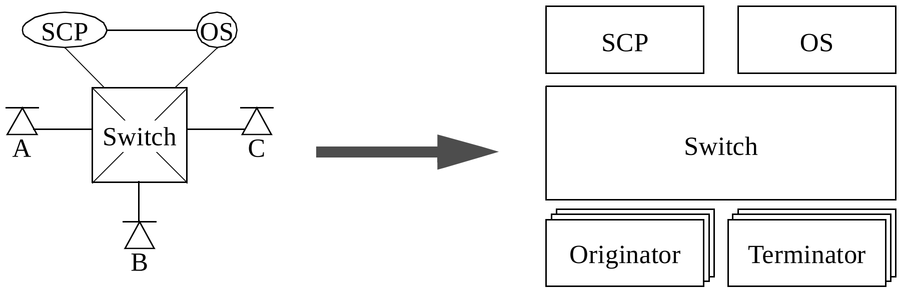


Figure 3 The Network and the UCM Structure

This network was transformed in an abstract structure (right half of Figure 3) on which UCMs that capture the Chisel diagrams are to be drawn. The switch, the SCP, and the OS were mapped onto abstract components. The phones were split into two sets of (replicated) components based on the user's role in a call, i.e., Originator or Terminator. The interfaces were left out as they are usually part of a more refined level of abstraction than the one addressed by UCM structures.

Features

On top of POTS, the first phase of the contest described ten features, but this report mainly focuses on four of them:

- *Calling Number Delivery* (CND): allows the called telephone to receive a calling party's Directory Number and the date and time. The number is delivered whenever an idle called party receives the Ringing event.
- *IN Freephone Billing* (INFB): allows the subscriber to pay for incoming calls. Call routing, although normally part of this feature, has been dissociated into another feature.
- *IN Teen Line* (INTL): restricts outgoing calls based on the time of day (i.e., hours when homework should be the primary activity). This can be overridden on a per-call basis by anyone with the proper identity code.
- *Terminating Call Screening* (TCS): restricts incoming calls. Calls from lines that appear on a screening list are redirected to a vague but polite message.

The six remaining features were IN Freephone Routing (INFR), Call Forwarding Busy Line (CFBL), Three-way Calling (3WC), IN Call Forwarding (INCF), Call Waiting (CW), and Charge Call (CC). The second phase contained two additional features, namely Cellular pays (Cell) and Return Call (RC). The UCMs developed in this phase considered a third additional feature as well, namely Automatic Call Back (ACB) [35].

3.3 UCM Capture from Chisel Diagrams

This section discusses the step (1) in Figure 1. Chisel diagrams are used to define requirements for communications services and service features. Since its design originated at a usability workshop involving practitioners, the language Chisel is intended to reflect current practice for writing these requirements [1]. The authors of this language claim that it is unambiguous, that it applies to a variety of network technologies, and that it has a sound basis for translation to commonly used formal software specification languages. The purpose of Chisel diagrams is to improve communication between the diverse people and organizations involved in the telecommunications service creation process.

The Chisel diagram for INTL is given in Figure 4. Sequences and alternatives of events on the network interfaces are supplemented by variables and conditions. Each node in the tree has a unique identifier, an events name, and a list of parameters. Nodes are also allowed to have multiple events, separated by |||, that can occur in any order. Some leaves are followed by references to a specific node in the POTS Chisel diagram, and variables in that diagram are assigned values from the INTL diagram. We will not dwell further in the explanation of Chisel diagrams, nor will discuss the meaning and the correctness of the INTL feature in Figure 4.

Usually, requirements do not come in such formal and operational form. In our case, since these Chisel diagrams are at a somewhat lower level of abstraction than UCMs, the scenarios to be captured will be more abstract. This is generally not the case because requirements are often described in prose form, i.e., in an informal and non-operational form.

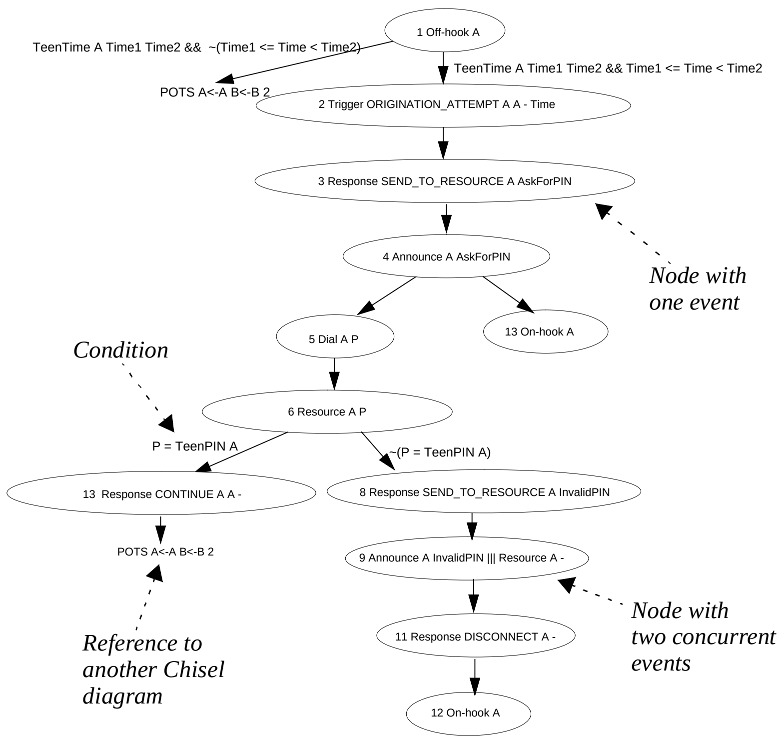


Figure 4 Chisel Diagram for IN Teen Line (INTL)

The Chisel diagrams are based on events that are shared between entities (the network components), whereas UCMs are described in terms of responsibilities performed by components. This first issue has been resolved by assigning these events to the component in which they will most likely be observed. Hence, events that are unobservable by the user become local to the system components (Switch, SCP, OS). Figure 5 shows a partial UCM for the INTL feature of Figure 4. Some events become responsibilities local to the switch (like setting the busy status of the originator), others become responsibilities that the user can observe (like getting an announcement "Ask for PIN"), and others remain events that the user can trigger (like off-hook). Responsibilities are marked with a cross, and event names are associated to start and end points. They are bound to their respective network components (see Figure 3). Obviously, some responsibilities will be refined as events or messages between components at a lower level of abstraction (like Chisel diagrams, Message Sequence Charts (MSCs) [29], ObjecTime design, or LOTOS specifications), but UCMs delay this kind of decision to a next refinement stage, possibly with another and more appropriate notation.

Resources and responses (on the SCP-Switch interface) were not put on the UCM because they are basically messages and they are hidden from the user's point of view (from which we describe the scenarios). However, their existence is somewhat implied by the path crossing the SCP-Switch boundary on several occasions. Resource and response messages represent only one way to implement the causal relationship shown in the UCM and the checking of the conditions in the SCP. This is in fact the refinement chosen by the producers of the original Chisel diagrams, which are more detailed (and hence less loose) than UCMs. UCMs provide a description similar to a *service specification* in the OSI model, where we can specify abstract actions and components that are not always visible to the user, but without committing too soon to an implement-oriented solution.

Note also that the conditions are simplified to the point where they become simple (italicized) labels on the paths. The conditions themselves should be expressed with another notation, more suitable for dealing with data.

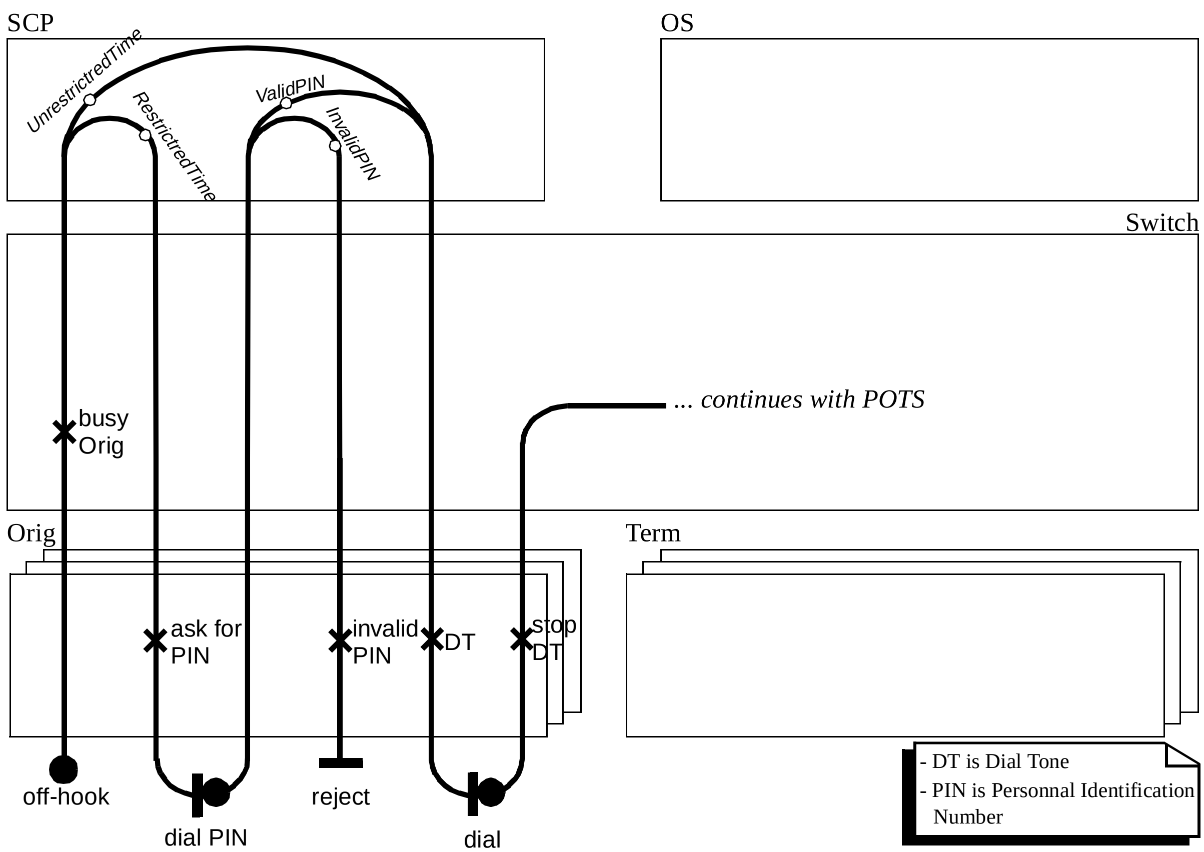


Figure 5 Partial UCM for INTL

This UCM is incomplete and focuses on the behaviour specific to INTL in the context of POTS. It then continues just like the POTS UCM would (although it is not shown in this report as an individual UCM). The INTL feature, as defined in the contest description, refers to POTS for common behaviour. This also means that disconnections need to be managed by our UCMs. One of the assumptions in the contest was that a hang-up could occur only at some specific points in the scenarios. These occur where end points (bars) are inserted in the UCM. Therefore, a disconnection could happen instead of **dial PIN** or **dial**, or after **reject**. Hence, a disconnection UCM, not shown here, is implicitly composed with the INTL UCM at each of these locations on the map. Its triggering would prevent the other events to occur and terminate the call connection(s).

3.4 Integration of UCM Scenarios

Individual scenarios are useful for understanding the behaviour of one feature, but they can also be integrated together to form a *global UCM* (step (2) in Figure 1*)*. The assumption here is that performing the integration at this level of abstraction provides early insights in possible conflicts between features expressed as scenarios. Integration helps to ensure early consistency between individual maps. For instance, events and responsibilities that are not labeled correctly, that are omitted, or that are not at the same level of abstraction or in the same order become hard to integrate. Hence, they indicate that some individual maps might need to be fixed. Integration also helps to avoid ambiguous situations, the most common of which is non-determinism. A path segment that is a prefix to two different scenarios might imply the need for a way to decide which alternative to take in a global scenario. Merging several path segments together might also indicate that variables and data are required to distinguish between the different cases, similarly to multiplexers in circuit design. Many such design decisions can be made at this level.

Root Map

The following root map and plugin maps result from the integration of the thirteen features enumerated in Section 3.2. This integration was done with the *UCM Navigator* tool [34], a UCM editor developed in our research group, which outputs the next few figures. The *root map* (Figure 6) represents the global context in which sub-maps are plugged in. The diamonds in this UCM are called *stubs* and they serve as placeholders for *plugin maps*. The diamonds with filled lines (e.g., **post-dial**) are *static* stubs and they contain only one plugin map. They are basically used as an abstraction mechanism and for path refinement. The diamonds with dashed lines (e.g., **pre-dial**) are *dynamic* stubs and they may contain several plugin maps from which one or more are selected at run-time depending on the satisfyability of their associated preconditions. Plugins are maps that can also contain their own stubs.

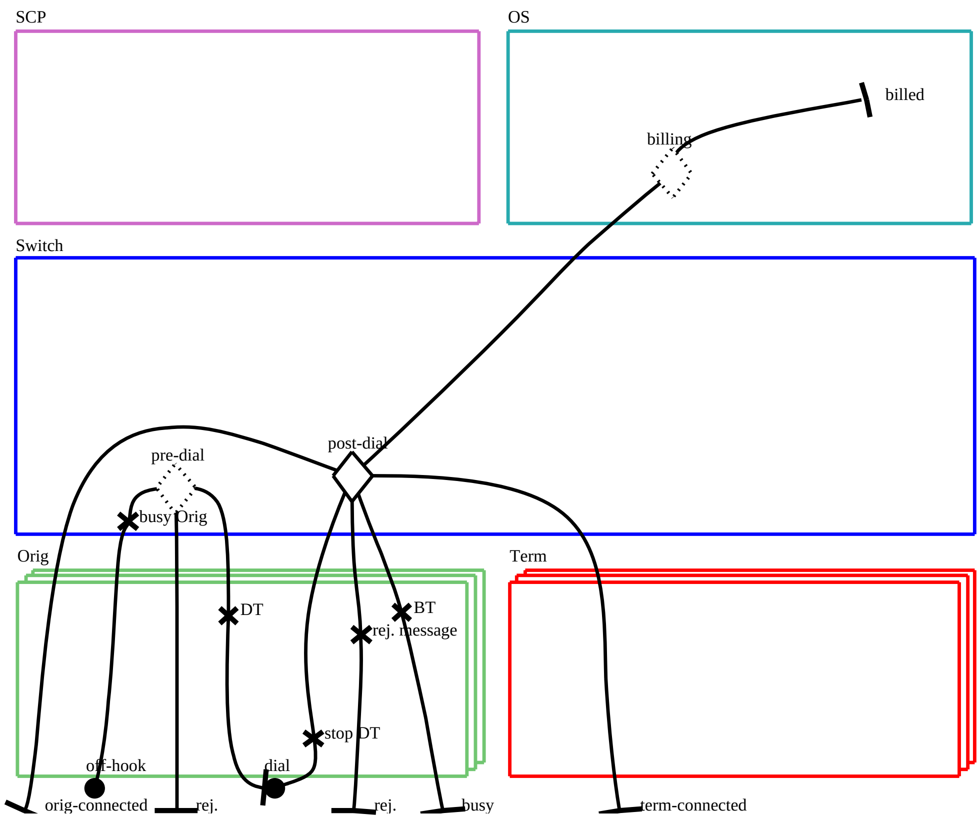


Figure 6 Root Map for Global UCM

Binding of Plugins to Stubs

One constructs a complete scenario by recursively selecting appropriate plugins for the stubs. Many figures in this section present plugins created for the FI contest. They are bound to the stub by associating the entry and exit points of the stub to the start and end points of the plugin map. The first stub in the root map, **pre-dial**, one entry point (*IN1*), and two exit points (*OUT1*, *OUT2*), as shown in Figure 7.

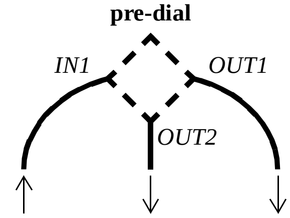


Figure 7 Entry and Exit Points on a Stub (**pre-dial**)

Each stub has its set of entry and exit points that may be bound to plugins. For instance, the default plugin for **pre-dial** is basically a straight path, whose start point is **POTS** and end point is **dial**, and which does nothing but connect *IN1* to *OUT1*. Hence, the binding is {(**POTS**, *IN1*), (**dial**, *OUT1*)}. *OUT2* remains unbound, and therefore this path (leading to **reject** in the root map) will never be followed when the default plugin is selected. This same stub has a second (and much more complex) plugin, illustrated in Figure 8.

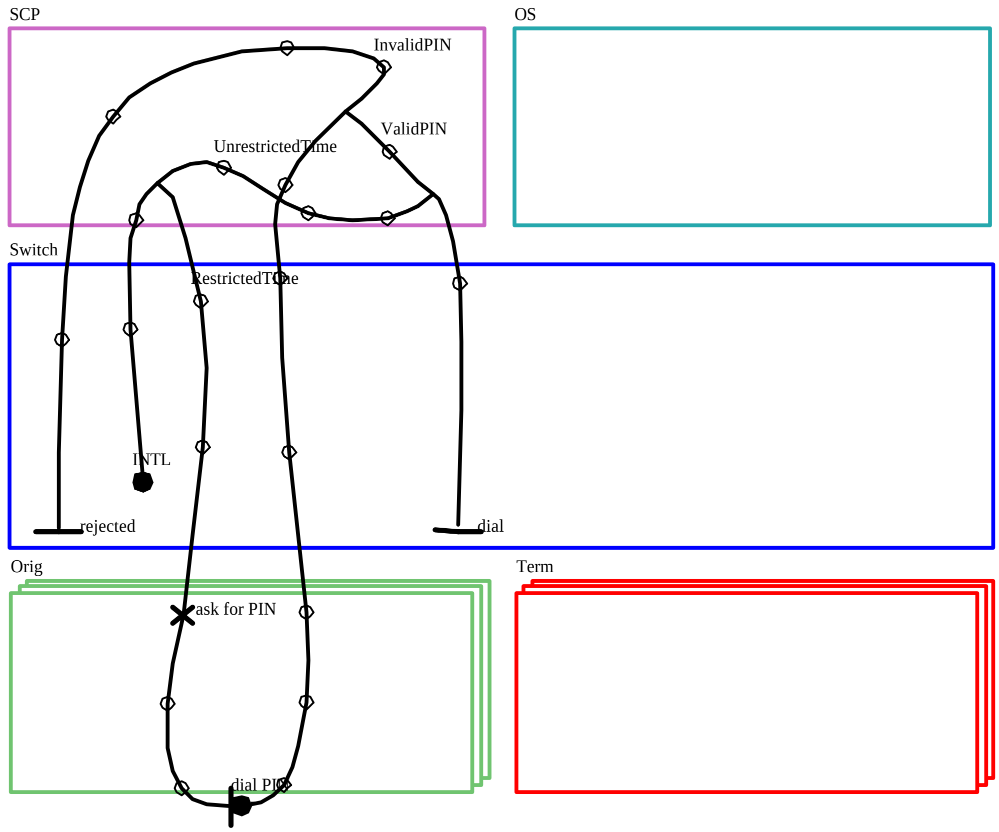


Figure 8 INTL Plugin for Pre-dial Stub in the Root Map[1]

Its binding is {(**INTL**, *IN1*), (**dial**, *OUT1*), (**rejected**, *OUT2*)}. With this plugin, it is possible to reach the second exit point that leads to a **reject** end point (itself leading to an eventual disconnection due to the implicit composition at each end point in the root map, as discussed in Section 3.3).

The INTL plugin of Figure 8 differs in other ways from the default plugin for **pre-dial**. Their preconditions are mutually exclusive, i.e., the user must be subscribed to INTL for this plugin to be selectable, and the user must not be subscribed to INTL for the default plugin to be selectable. Hence, the two plugins can never be active simultaneously. This alternate composition within the stub results from the nature of the individual features and from how they were integrated together. In essence, INTL is the only feature that deviates from all the others between the update of the busy status (**busyOrig**) and the dial tone (**DT**).

When a user is subscribed to INTL only, the flattening of the root map with the INTL plugin in the **pre-dial** stub and default plugins in the other stubs results in the individual UCM of Figure 5.

Other Relevant Plugins

To obtain a complete picture of the system with the four features that interest us (CND, INTL, INFB, and TCS), we now give an overview of the remaining appropriate plugins. Bindings will not be discussed unless they are not obvious from the figure.

1. The empty circles on the paths are *empty points* and are used for path transformations in the UCM Navigator. They are not part of the UCM notation as such.

The **post-dial** static stub in the root map contains by definition only one plugin, which is shown in Figure 9 (where **R** means Ringing, and **RR** stands for the remote AudibleRinging). In this UCM, several path segments are concurrent, as explicitly stated by the ||| operator in the Chisel diagrams. Some slight differences were introduced at this point due to the distributed nature of our system that cannot be so easily abstracted from with paths that cross components. For instance, an **RR** could occur at the originator after the terminator has picked up the phone, thus representing the fact that the system might take time to consume the **off-hook** event before deciding to stop the **R** and **RR** activities. This behaviour, which might reflect the real system better, contains the behaviour described in the Chisel diagrams, but also allows for other global scenarios.

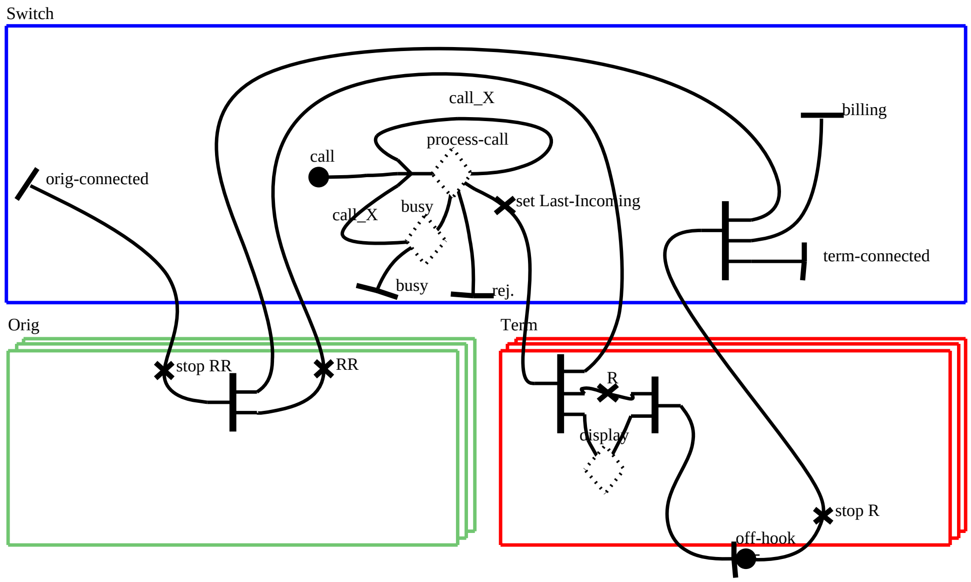
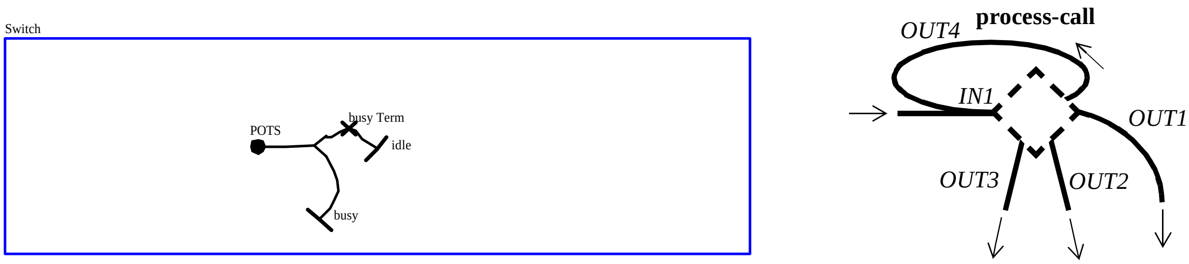


Figure 9 Plugin for Post-dial Static Stub in the Root Map

The default plugin for the **process-call** stub of Figure 9 is illustrated in Figure 10 (a). This is the point where the system checks whether or not the terminator side is busy. If so, the **busy** path is selected. Otherwise, the **idle** path is selected and the terminator status is set to busy (**busyTerm**). The binding is {(**POTS**, *IN1*), (**idle**, *OUT1*), (**busy**, *OUT3*)}.



(a) Default Plugin (b) Stub Entry/Exit Points

Figure 10 Default Plugin for Process-call Stub in Figure 9

The INFB plugin (Figure 11) does not override the default one, but occurs before. It simply analyzes some IN information in the SCP and then sets the called party as the paying party. The binding is {(**INFB**, *IN1*), (**callB**, *OUT4*)}.

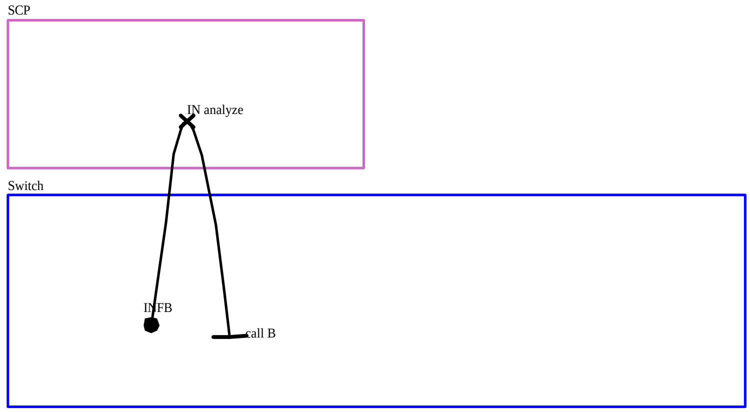


Figure 11 INFB Plugin for Process-call Stub in Figure 9

The TCS plugin of Figure 12 is similar in nature to the default one, except that it first checks whether or not the originator party is on the screening list. If so, then the call is rejected. It overrides the default plugin when the terminator party has subscribed to TCS. The binding in this case is {(**TCS**, *IN1*), (**idle**, *OUT1*), (**TCS-reject**, *OUT2*), (**busy**, *OUT3*)}.

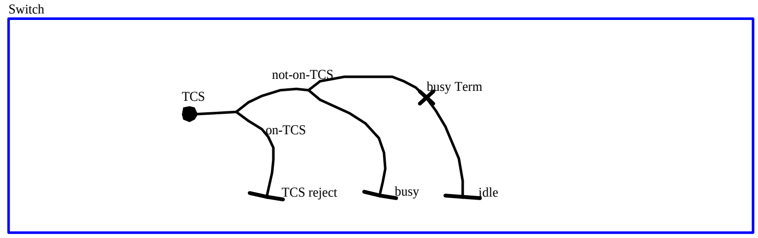


Figure 12 TCS Plugin for Process-call Stub in Figure 9

The **busy** stub in Figure 9 has one plugin that concerns us, the other being related to the features not discussed in this report. The default plugin in this case simply connects the entry point to the path leading to the **busy** event. The last stub in this figure, **display**, also has a straightforward default plugin that does nothing but connecting the entry point to the exit point. When the terminator side has CND active, the plugin shown in Figure 13 is used instead of the default one in order for the originator's number/name to be displayed.

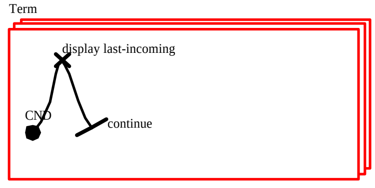


Figure 13 CND Plugin for Display Stub in Figure 9

Finally, the billing stub in the root map (Figure 6) contains two mutually exclusive plugins selected according to whether or not the terminator has subscribed to INFB. If so, then the terminator party (also referred as **B**) is charged with the incoming call, otherwise the originator (**A**) pays.

(a) Default Plugin (b) INFB Plugin

Figure 14 Default and INFB Plugins for Billing Stub in Root Map

3.5 Avoiding Feature Interactions

We claim that an integration of scenarios at the level of UCMs helps to avoid some trivial or artificial interactions between features. For instance, many potential interactions between INTL, INFB (or TCS), and CND are avoided because the features in each possible pairwise combination are allowed to proceed independently in the map. They are integrated using a sequence of three different stubs that encapsulate the features from their environment.

Important design decisions still need to be made at integration and composition time, something that cannot be easily automated. For example, interactions between features in one stub (e.g., INFB and TCS) are still possible, depending on the composition/decision mechanism used within the stub (**process-call** in our case). Maps with stubs show how localized the impact of a feature can be. They can be represented by only one plugin (INTL in **pre-dial**), or by several plugins along one or many paths (INFB in **process-call** and **billing**). This helps focusing on issues related to how a plugin (i.e., dynamic behaviour) is selected in one or more dynamic stubs. Since only a limited number of smaller UCMs have to be considered in a stub, it becomes easier to check that they have mutually exclusive but complete preconditions (to avoid non-determinism and unspecified behaviour), or that priorities need to be established. Hence, the design decisions are simpler. The integration becomes an interesting and useful step in a design process that includes UCMs, and it cannot be as trivial as the composition of states suggested by the Chisel approach. However, the composition of plugins in a stub should not be done at the UCM level, which is not an adequate notation for such details. A more appropriate notation, such as LOTOS or some agent meta-models, should be used instead.

Chisel diagrams specify normal behaviour, but they do not distinguish between what should be obliged and what should be permitted or even forbidden in a feature. For instance, in their respective Chisel diagrams, CND displays the incoming call and charges the call to the originator, while INFB does not display and charges the call to the terminator. Although this appears to be an interaction, it is somewhat artificial since these two features are obviously compatible in telephony systems. That is because CND *obliges* the display and *allows* for the terminator to pay (it is not forbidden), whereas INFB *allows* the display (it is not forbidden) and *obliges* the terminator to pay. Stated like this, these requirements, which are acted upon only at integration time, lead to a global system without such interactions between CND and INFB. This kind of information (modalities on the alternatives) would help to determine what stubs are required and how the default behaviour (POTS) is overridden. In our example, we had to infer this knowledge manually from our understanding of the *intent* of these features. A notation like the OPI model (Obligation-Permission-Interdiction) would make this distinction explicit in the description of a feature [7]. Supplemented with OPI concepts, UCMs could be used to better capture the intent of features in terms of scenarios, and not in terms of properties as it is usually the case.

4 LOTOS SPECIFICATION

4.1 LOTOS and the Synthesis & Validation Approach in a Nutshell

Overview of LOTOS

For the last decade, we observed that formal methods, such as LOTOS, SDL, MSCs, and Estelle, have proven their usefulness in capturing descriptions of complex, concurrent, and distributed systems. LOTOS is an algebraic specification language standardized by ISO [26]. Using LOTOS, the specifier describes a system by defining the temporal relations along the actions that constitute the system's externally observable behavior. Data abstractions can also be described by using *Abstract Data Types* (ADTs).

LOTOS is powerful at describing and prototyping distributed systems at many levels of abstraction through the use of *processes*, *hiding, parallel composition* and *multiway synchronization*. LOTOS is suitable for the integration of behavior and structure in a unique executable model. LOTOS models allow the use of many validation and verification techniques such as step-by-step execution (simulation), random walks, testing, expansion, model checking, and goal-oriented execution. Many tools can be utilized for the automation of these techniques, and several development cycles based on stepwise refinement are available.

Synthesis of Specifications from UCMs

The synthesis of LOTOS specifications, illustrated by our example scenario in Figure 15, allows for the rapid generation of prototypes that implement UCM scenarios. The behaviour of each component is translated into a LOTOS process that preserves the internal causality relationships between the responsibilities and events that are part of path segments crossing this component (right half of Figure 15). The architecture itself is converted to a structure (left half of Figure 15) where the processes are composed together through shared communication *channels*[1] (LOTOS gates). The causal relationships between the components are also considered during the construction of the processes. Decisions related to the nature of the message exchanges must then be made and documented.

1. We use the generic term *channel* to denote a communication link between two entities, not necessarily a SDL channel (a FIFO queue).

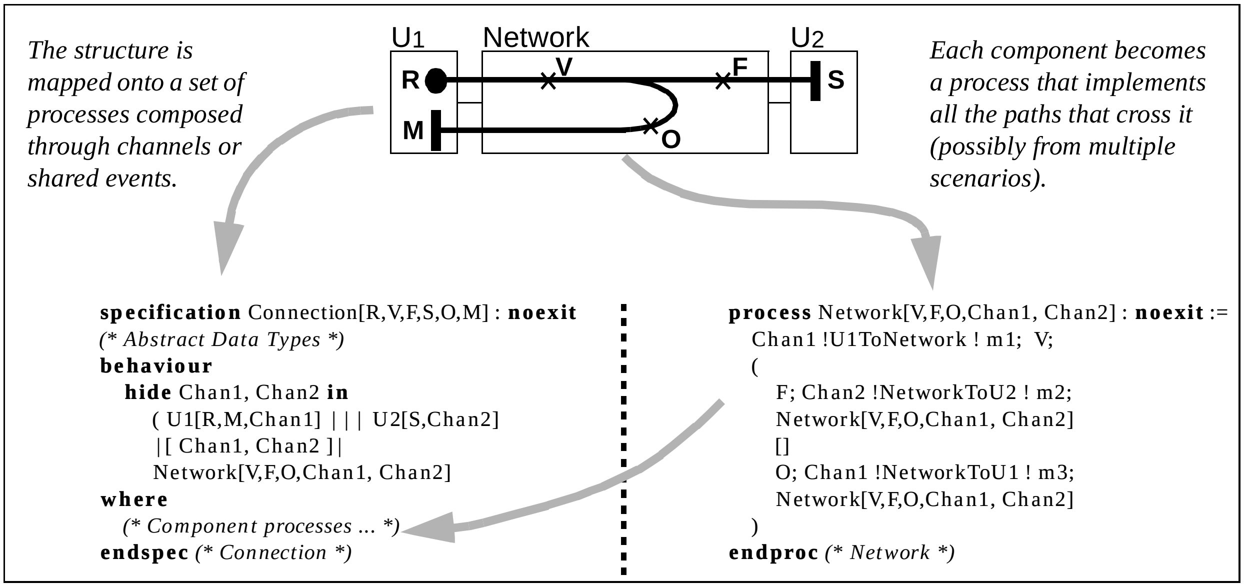


Figure 15 Synthesis of a LOTOS Specification from a UCM

LOTOS Validation

Since the synthesis is not automated, it becomes necessary to validate the specification against the UCMs, which correspond to the (informal) requirements. Four of the most common approaches to the validation of a LOTOS specification are simulation, equivalence checking, model checking, and functionality-based testing.

Simulation is the step-by-step execution of a specification. The designer takes the role of the environment, provides events to the specification, and observes the results (the next events). Although useful for debugging, simulation is probably the weakest validation technique available.

Equivalence checking usually requires a formal representation of (part of) the requirements, seldom available in the early stages of the design process. However, this approach is most useful when checking the conformity of one specification against another, after some refinement or modifications.

Model checking aims to validate a specification against safety, liveness, or responsiveness properties derived from the requirements. These properties can be expressed, for instance, in terms of temporal logic or μ-calculus formulas. In the LOTOS world, this technique usually requires that the specification be expanded into a corresponding model, which is some graph representation (labeled transition system, finite state machine, or Kripke structure) of the specification's semantics. On-the-fly model checking techniques, where the whole model does not have to be generated a priori, exist as well. Often, the languages used to define properties are very flexible and powerful, yet they can be quite complex; it is a difficult problem to determine whether a property really reflects the intents of informal requirements.

Functionality-based testing is concerned with the existence (or the absence) of traces, use cases, or scenarios in the specification. These scenarios reflect system functionalities, usually in terms of operational or user-centered instances of intended system behaviour. They can easily be transformed into black-box test cases that can be composed with the specification for validating the latter against requirements. Test cases are often more manageable and understandable than properties, and they relate more closely to informal requirements. However, they are usually less powerful and expressive than liveness or safety properties expressed in temporal logic. For example, a test suite that passes successfully does not prove the absence of errors in any way.

Among these four approaches, we favored functionality-based testing for the validation of the features and the detection of interactions. Simulation is not sufficient because there are just too many global sequences of events possible in the system. Equivalence checking is not possible because we aim to produce a first high-level specification from the scenarios. Since these requirements are expressed mostly operationally, UCMs and test cases are easier to extract than properties, so model checking should not be used at first. It could be used later on, however the state explosion problem can hardly be avoided in our case.

LOTOS Testing from UCMs using LOLA

LOLA (LOtos LAboratory) is a state exploration tool with application in simulation, testing, and transformation of LOTOS specifications [37]. It has the particularities of accepting Full LOTOS and of being available on several platforms (including SunOS, Linux and DOS). Its testing strategy is consistent with the *Testing Equivalence*. The LOTOS testing theory has a test assumption stating that the implementation (the specification in our case) communicates in a symmetric and synchronous way with external observers, the test processes. There is no notion of initiative of actions, and no direction can be associated to a communication.

In the following, we assume that *Success* is a special gate, not part of the specification under test, which is used in the test cases to indicate a successful execution. LOLA expands the composition of the specification and a test process in order to analyze whether the executions reach the success event or not. Three *verdicts* can occur after the execution of one test case:

- **Must pass**: all the possible executions (called *test runs*) were successful (they reached the *Success* event).
- **May pass**: some executions were successful, some unsuccessful (or inconclusive according to a depth limit).
- **Reject**: all executions failed to reach *Success* (they deadlocked or were inconclusive).

In the real world, test cases must be executed more than once when there is non-determinism in either the test or the implementation (under some fairness assumption). However, LOLA avoids this problem because it determines the response of a specification to a test by a *complete* state exploration of the following composition [36]:

```
SpecUnderTest[EventsSpec]
|[EventsSpec ∪ EventsTest]|
Test[EventsTest ∪ {Success}]
```

LOLA analyzes all the test terminations for *all possible evolutions* (test runs). The successful termination of a test run consists in reaching a state where the termination event (*Success*) is offered. A test run does not terminate if a deadlock or internal livelock is reached.

Validation test cases are usually derived from the UCMs in order to detect errors, incompleteness and inconsistencies. For most distributed systems, including telephony systems, the high (if finite) number of global states makes the generation of an exhaustive test suite impossible. Hence, it becomes essential to carefully select a small and finite set of validation test cases. To do so, we can base our strategy is based on the exploration of UCM paths, similarly to white-box approaches used for sequential programs. Depending on the targeted coverage, the critical nature of paths, and the cost associated to their traversal, we can choose to explore some paths, all combination of paths, some or all the temporal sequences resulting from concurrent paths, etc. For each selected abstract sequence of events/responsibilities (UCM routes), *acceptance* test cases (whose expected verdict is **Must pass**) and *rejection* test cases (whose expected verdict is **Reject**) can be generated.

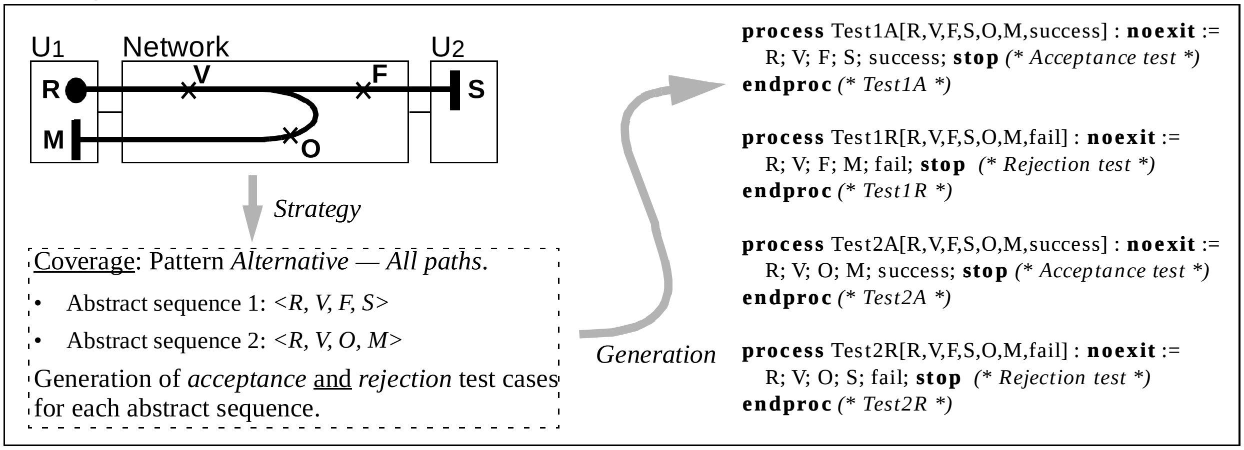


Figure 16 Derivation of Validation Test Cases from UCMs

Our sample scenario is reused again in Figure 16, to demonstrate the derivation of a set of test cases with the goal of covering all paths in the UCM. Each path linking a start point to an end point then becomes an abstract sequence that will be translated into a LOTOS test process (while considering the observable messages and data types defined during the synthesis). In this example, the rejection test cases were generated from the abstract sequence where a mutation was applied on the last event (a fault model called *off-by-one*).

Although this general test derivation approach could be used for validating our features, we chose instead to use a more detailed model that was available to us, namely the Chisel diagrams. These diagrams are described at a somewhat lower level of abstraction than UCMs, and therefore they bring more precision to the definition of the tests for individual features. In general, when one starts from informal requirements, such detailed description is not yet available. Hence, validation test cases are usually derived from UCMs, not from the requirements (contrarily to transition (4) in Figure 1). However, since the Chisel diagrams were given to us in the contest description, their use seemed appropriate (see Section 4.3). Moreover, we limited our scope to acceptance tests only (with a **Must pass** verdict expected). Rejection test cases are left for future work.

4.2 Synthesis

The current section relates to step (3) in Figure 1. Following the synthesis approach introduced in the previous section, we are now about to generate a LOTOS specification from the global UCM (Figure 6) and its plugins. This specification, presented in Appendix A, will serve as the basis for the validation of individual features against their requirements and for the detection of interactions.

This section provides general explanations about the synthesis of our LOTOS specification. We first discuss the data types needed to support the parameters, databases, and preconditions. We follow with the representation of the network (Figure 3) as a structure of LOTOS components.

Finally, for some components of the network, we present the construction of the processes' internal behaviour from the UCM paths that have responsibilities bound to these components.

Data Types

The abstract data types are mostly derived from the tabular descriptions in the contest description [25], except for the basic data types and operations (`Boolean`, `NaturalNumber`, `FBoolean`, `Element`, and `Set`), which are ADTs simpler than the ones in the International Standard (lines 80 to 227). They were simplified in order to become more efficient in our tools. The ADTs specific to the features are as follow (lines 228 to 853):

- `Time`: discrete time, counted in tics.
- `Address` and `AddressList`: user's address, and list thereof.
- `Cadence`: `Ring` or `SpecialTone` (not used by our restricted set of features).
- `PIN`: `validPIN` or `invalidPIN`, instead of a real personal identification number.
- `Message`: used for announcements.
- `TriggerName` and `ResponseType`: IN triggers and their responses.
- `LogType`, `LogRecord`, and `Log`: for the list of log records in the OS.
- `Feature` and `FList`: for lists of features.
- `SInfo` and `SDB`: for the database of subscriber information in the switch.
- `SCPit`, `SCPinfo` and `SCPDB`: for the database of feature parameters in the SCP.
- `StatItem`, `Stat` and `Status`: for the database of status items in the switch.
- `StubPath` and `SPList`: entry/exit points of each stub in the maps, and list thereof.

These abstract data types support the representation and the manipulation of information for the thirteen features described as UCMs in [35], and not only for the four features on which we focus in this report.

Structure

LOTOS gates were used to represent individual events shared between the network components (Figure 3). These components are represented as LOTOS processes and are synchronized on common gates. Each event in the Chisel diagrams of the contest description (i.e., each responsibility in the UCMs) is mapped onto a unique gate. Therefore, instead of using gate splitting for representing the on-hook and off-hook events on the user/switch interface (as in `user2switch!onHook` and `user2switch!offHook`), we have two individual gates (`onHook` and `offHook`). Having individual gates permits more specific compositions between processes and, more importantly, between the specification and the test processes.

Since we are designing the system from the user's viewpoint, some events will be observable while others will remain hidden within the system. Hence, the observable events are the ones on the switch-to-user and user-to-switch interfaces, and are enumerated in lines 59 to 79. The hidden events are those on the switch-to-SCP, SCP-to-switch, and to-OS interfaces (lines 860 to 870).

We also created four additional events. The hidden event `Time` is used by the switch to get the current time from a global clock. We use three other observable events to improve the testability of our specification. `Init` allows the initialization of all the databases used by the network components with users' values (likely to come from a test case). `CreateUser` is used to create users (originators and terminators) and specify their initial state. Finally, `Query`'s purpose is to allow a test case to verify the log in the OS at the end of the test.

The top-level process structure itself is derived from the network components and the way they interact with each other (lines 875 to 897). Figure 17 illustrates this structure with titled boxes for components, local variables and databases (with their type) between parentheses, and lines for the LOTOS synchronization operator (|[...]|). Because this is a binary operator, artificial groupings (boxes without titles) become necessary, and they may not have any logical meaning. Each of these lines represents the set of common gates on which the two sides may synchronize. The **GlobalTime** process stands for the global clock mentioned in the contest description, which is queried by the switch on several occasions.

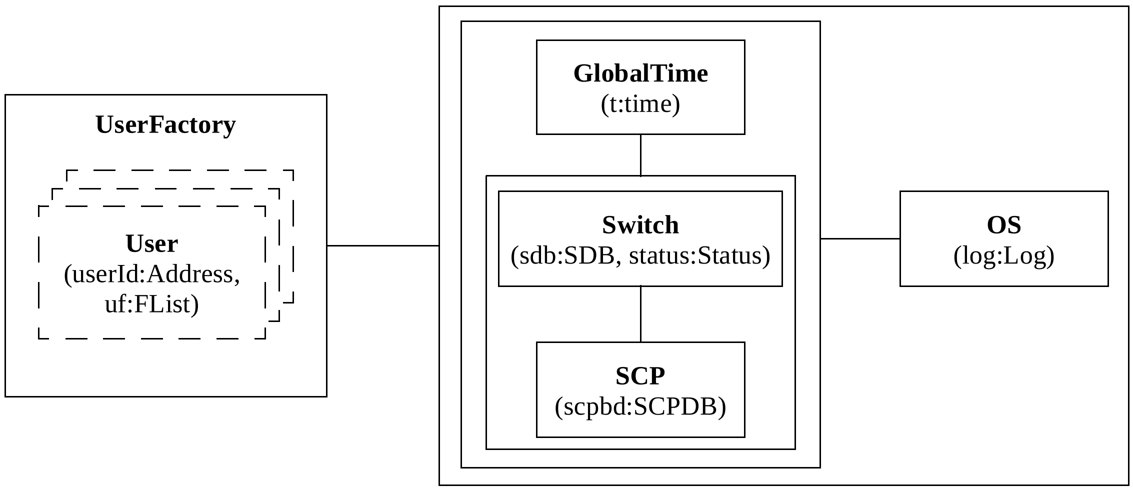


Figure 17 Top-Level Process Structure

The dashed boxes for **User** indicate that these processes are created dynamically within **UserFactory** and that they interleave (|||) with each other. Figure 18 presents a MSC that illustrates how we can create two users with their own identity and subscribed features.

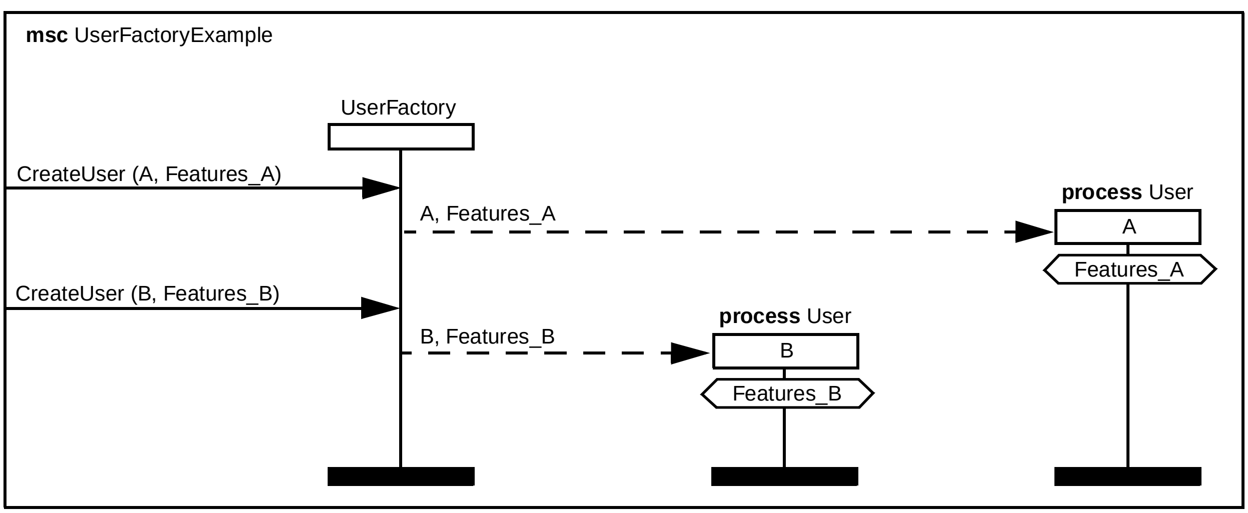


Figure 18 How the UserFactory Process Works

Components to which stubs are bound have sub-processes, one for each stub. Moreover, dynamic stubs may themselves have multiple sub-processes, one for each plugin. The stub process is then used to specify the composition between the possible plugins. Each of these processes receives a list of entry/exit points (type `SPList`) as input and then outputs another such list upon termination.

Process Behaviour

As illustrated in Figure 15, UCM paths define the behaviour of the components over which they pass. Components are thus responsible for the events and responsibilities bound to them, and for the implementation of their causal relationships. For the construction of process behaviour, we only consider the four features that interest us, and the others are left to future work. Moreover, for simplification purposes and for conformance to the Chisel diagrams, the specification considers only one call session, i.e., it is not possible to initiate a sequence of call sessions (the behaviour of the switch is not totally tail-recursive). We cannot possibly explain all the synthesis decisions that were made, but we illustrate the main concepts with three examples.

The `User` process has multiple path segments to take care of. The originator and terminator roles are merged together to form this unique process. Their integration results in seven alternatives between different multi-sequences (trees) of events. As an example, consider the path segment from INTL that crosses the originator in Figure 8. The abstract sequence <**ask for PIN**, **dial PIN**> has to be implemented somehow in the process. The resulting multi-sequence is specified in lines 1003 to 1012. `AskForPIN` is an announcement received from the switch through the `Announce` gate. At this point, we need to note that the generation of this announcement has to be reflected symmetrically in the `Switch` process. This event is then followed by two alternatives, the first one corresponding to the event in the abstract sequence, i.e., `Dial` with `PIN` as a parameter (to be provided by the test case, hence the `?` instead of the `!`), followed by a recursive instantiation of the `User` process. The second one, although not part of the abstract sequence, comes from the fact that `Dial` is a point where a disconnection may occur (Section 3.3), hence the `OnHook` followed by a **`stop`**. The `userId` parameters are used to distinguish between different instances of the `User` process. Again, these events have to be generated from the synchronizing process, which is `switch` in this case. Notice that this multi-sequence corresponds to the states 4, 5, and 13 in the Chisel diagram for INTL (Figure 4).

The second example, also from INTL, relates to the behaviour of the SCP. INTL has two paths crossing this component: the first one “implicitly” checks whether or not we are in the restricted *TeenTime* period (known by the SCP database), and the other checks the validity of the PIN (again from the SCP database). A call-return mechanism implements these “implicit” checks: they are caused by a `Resource` event and result in a `Response` event. Appropriate parameters for `Resource` include `user` and `time` for the checking of *TeenTime*, and the `Response` contains the boolean value resulting from the evaluation of the `IsInTeenTime` predicate (lines 1465 to 1468). For the PIN validation, `user` and `pin` are needed as input parameters, and this result in a `Response` containing the `CONTINUE` message when the PIN is valid. When the PIN is invalid, then a `SEND_TO_RESOURCE` message is sent back, followed by the reception of a `Resource` request resulting in a `DISCONNECT Response` (lines 1474 to 1487). Note the following points:

- These multi-sequences correspond to the states 1, 6, 8, 9, 11, and 13 in the Chisel diagram for INTL (Figure 4). Since there are two states numbered 13 in this diagram, we have to specify that we are referring to the one in the lower-left corner.
- These `Resource` requests and `Response` must be mirrored in the Switch (see lines 1200, 1201, and 1216 to 1226).

- The Switch is really the component that decides what to do with the result provided by the SCP for the checking of *TeenTime*. Therefore, the OR-fork (where paths split) should probably be located in the Switch. In general, UCMs do not claim or intend to specify where decisions are made, but it is always better to have the OR-forks reflect these locations when they are known.
- State 9 in Figure 4 regroups two events that can occur concurrently (or in any order). The Switch has to synchronize on these two events. Lines 1222 and 1223 of the specification in Appendix A specify that the Switch prescribes one ordering. We used only one possible refinement in order to reduce the state space during validation. We refined in this way many Chisel states that contained the ||| operator.

For the last example, we look at the specification of a simple stub, namely **display** (Figure 9). This stub has a process (`DisplayStub`) that is instantiated by process `User` at line 979, concurrently with the start of the ringing (`StartR`). Within this stub (described in lines 1029 to 1044), the two plugins (default and CND, see Figure 13) are specified as mutually exclusive alternatives. Since these are quite simple plugins, no other sub-processes seemed necessary. As soon as the terminator subscribes to CND, the CND plugin is selected. Note that this process has an `inPaths` parameter of type `SPList`, which allows the calling process to indicate from which entry points in the stub the events are coming. In our case, the stub has only one entry point and the process is instantiated with the value `inDisp1`. Upon successful termination, the process exits with another `SPList` that contains the list of exit points in the stub that should be activated. Again, since there is only one exit point, both plugins exit with `outDisp1` for this result. These values are then used by the calling process to reason about what happened within the stub (lines 981 to 984).

These examples have illustrated some of the basic concepts used to synthesize the LOTOS specification:

- Components are implemented as processes synchronized on their common channels/gates.
- Because of their reactive nature, most components are specified with implicit recursive behaviour.
- Hidden gates are used for what is not observable by the user.
- Path segments in one component are integrated together, often as alternatives (could also be integrated as concurrent multi-sequences, depending on the UCM context).
- UCM activities are implemented as gates or as messages exchanged between components.
- Composition with the disconnection phase is applied to specific points in the global UCM.
- ADTs are used to represent databases and operations, and to evaluate conditions.
- Symmetry is enforced in synchronized actions (actions in one process must be mirrored in the other synchronized processes, unless locally hidden).
- Chisel states with the ||| operator are refined into simpler sequences, for the reduction of the state space.

Several additional rules for define for the specification of the stubs:

- Components with stubs have sub-processes, one for each stub.
- Dynamic stubs may have multiple sub-processes, one for each plugin.
- The stub process is used to *specify the type of composition* between the possible plugins.

- Each stub process receives a list of entry/exit points as input and then outputs another such list upon termination.

These concepts have been used throughout the construction of the specification, although we deviated from them on several occasions while debugging the integration.

4.3 Testing

We are now about to derive test cases for validating POTS and the individual features against their requirements, the Chisel diagrams (step (4) in Figure 1).

Structure of the Test Suite

In LOTOS, the testing is done through the composition of test processes with the specification [9]. Often, LOTOS test processes are sequential, monolithic, and deterministic in nature. However, through process instantiation, LOTOS test processes can be built on top of each other, hence reusing part of previous test processes in new ones. We make use of this capability in our strategy. We will define shared processes that represent sub-sequences of test cases. We call these processes *common behaviours*. In the conformance testing framework used in telecommunication systems [27], these common behaviours correspond in a way to *test steps*, which may be instantiated from multiple test cases and other test steps.

Figure 19(b) shows the bottom level of LOTOS test processes, composed solely of common behaviour processes for POTS. They are reused by the POTS test cases, and also by common behaviour processes for individual features. On top of the latter, we construct test processes for individual features, and also for each pair of features. Common behaviour processes then become reusable by many test cases, which simplifies the generation of test suites and increases the consistency among test cases.

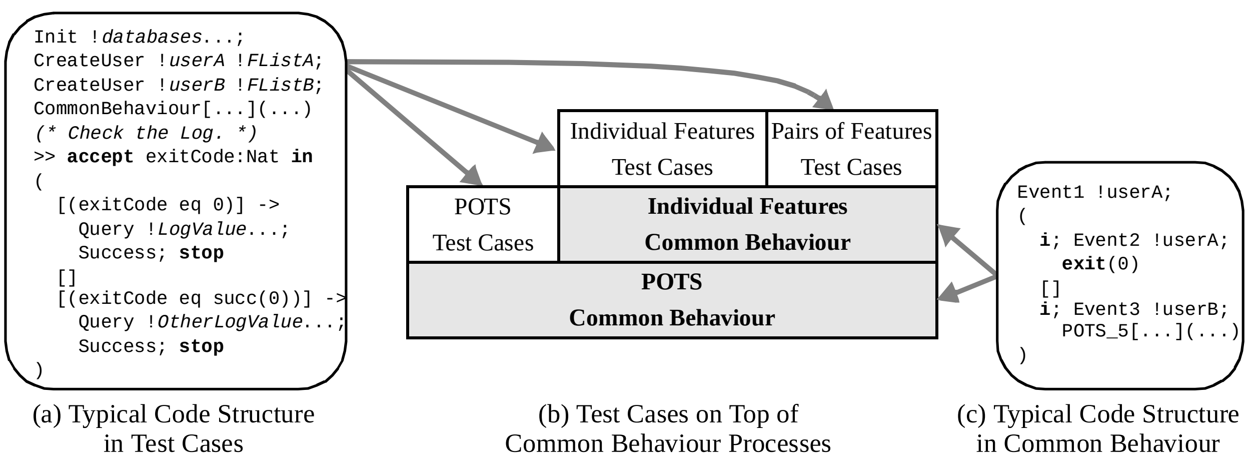


Figure 19 Construction of the Test Suite

Figure 19(c) presents the typical code structure in common behaviour processes. They are mainly composed of simple expressions that terminate with an *exit code* (**exit**(*n*)). With LOLA, test cases do not need to be sequential or deterministic, so alternatives and explicit non-determinism are allowed in common behaviour processes. Note that many alternatives are preceded by the internal action **i**. This non-determinism ensures, under LOLA, that all branches in the test case will by selected and covered at testing time.

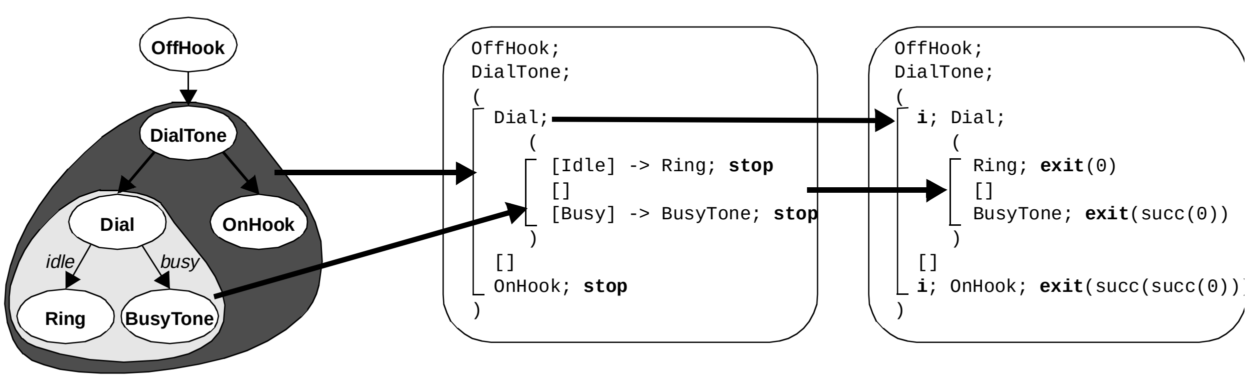
In Figure 19(a), the typical code structure illustrates that a test case provides the system configuration and verifies the exit codes. More specifically, the system is first initialized (by `Init`), users are created according to the mechanism shown in Figure 18, and the test cases themselves are performed by instantiating common behaviour processes. The exit code is then captured and used to validate the log against its predicted value.

During the testing, a deadlock in a test case for POTS or for an individual feature indicates that there is a bug that needs to be fixed. When all these test cases pass successfully, a deadlock in a test case for a pair of features indicates an unexpected interaction. Interactions will be covered in Section 5.

POTS Common Behaviour

We constructed a tester for POTS using six processes (lines 1534 to 1647). They have two parameters, representing the originator and terminator users, whose values are provided by the test processes. POTS states 1, 2, 4, 5, 13, and 15 were defined because they were referred to by the Chisel diagrams of other features. This is one of the main reasons why such common behaviour can be so easily reused.

A LOTOS *canonical tester* is a process that tests all of the behaviour of an implementation for conformance to a specification [9]. Inspired from this theory, we used the Chisel diagrams in order to obtain a reduced set of test cases, while maintaining a good validation power. In essence, a canonical tester has the same traces as the specification it aims to check, but it forces the coverage of all alternatives when the environment has a decision to make. For instance, Figure 20(a) shows a simplified Chisel diagram for POTS, for which a LOTOS interpretation is provided in (b). In its test process (c), the addition of an internal action **`i`** before `Dial` and `OnHook` (corresponding to the dark area) forces the composition to check both alternatives, which are both valid user inputs from the system's point of view (Figure 20(b)). In a similar way, if the system makes an internal decision, by using guarded behaviour or with hidden events, then the test process has to accept all possible outputs accordingly. The light shaded area presents a case where the system offers either `Ring` or `BusyTone` depending on its internal information about users' status. A real LOTOS canonical tester would also take care of all possible values associated to the parameters. We chose not to follow this strict rule because we wanted to generate common behaviour processes where the parameters are set during the initialization phase in a test case. Our test cases do not check all possible configurations, while a canonical tester would (which is seldom feasible).



(a) Chisel Diagram (b) System (c) Test Process

Figure 20 Example of a Canonical Tester for a Chisel Diagram

Note that we assigned an exit code to the leaf at the end of each branch in a Chisel diagram. This allows test processes to determine what branch has been selected, in order later to check the validity of the log collected by the OS.

Interleaved events in a Chisel diagram should, according to the canonical tester theory, require that all possible combinations of events be covered explicitly. For instance, state 9 in Figure 4 has `Announce ||| Resource`. Let's rename this `A ||| R` and assume these are events provided by the user to the system. Following the LOTOS expansion theorem, this expression is equivalent to `A;R [] R;A`, hence the canonical tester would need to be `i;A;R [] i;R;A`. However, we will leave `A ||| R` as is in the tester for two reasons:

- On several occasions, we implemented only one alternative in the system (see the end of the previous section). By this refinement, the system has already made the decision, and thus the user needs to accept it.
- Leaving the `|||` operator leads to simpler expressions.

Processes `POTS_1` to `POTS_15` represent the lowest layer of common behaviour, and will now be used, directly or indirectly, by almost all of the other test processes.

POTS Test Cases

Often, more than one test case will be required to cover a Chisel diagram, because initial states and conditions are necessary. POTS has only one *precondition*: whether or not the terminator side is busy. Hence, two test processes can cover all the states in the Chisel diagram (lines 1648 to 1714). Process `tPOTS1` tests the cases where the terminator side is not busy, whereas `tPOTS2` takes care of the cases where the terminator is busy. They both use `POTS_1` as their start point. Note that the names of all test processes start with a lowercase *t*, while common behaviour processes start with a lowercase *c* (except for the POTS common behaviour processes). Test processes for pairs of features are prefixed with *fi*.

Test Cases for Individual Features

These tests check that each feature acts properly when being the only one active (lines 1715 to 2123). The previous test suite (for POTS) still needs to be checked because, in the absence of any active feature, what remains must be the regular POTS behaviour.

Table 1 presents the 10 test processes used for the coverage of INTL, CND, INFB and TCS, according to their respective Chisel diagram. Each test was created by providing an initial configuration (according to the conditions shown in the Chisel diagrams and the individual UCMs) and by calling the appropriate common behaviour process. These tests were applied to the specification, and results were collected (steps (7), (8), and (9) in Figure 1).

For each test, we included its purpose (according to the preconditions that need to be satisfied by the initial configuration), the common behaviour process it uses[1], and how many unique global sequences were generated by its composition with the specification. Each of these traces could be represented as unique message sequence charts from the user's point of view. Some non-determinism inside the system (which would create many more global sequences) has been abstracted from in this experiment; on-the-fly reduction techniques, which preserve testing equivalence, have been used while testing with LOLA. All of them were successful, therefore we do not

1. A common behaviour processes can call other such processes. For instance, POTS_1 calls POTS_2, which in turn calls POTS4 and POTS_15. POTS 4 calls POTS_5 and POTS_13.

have any indication that POTS and the individual features are faulty in our system. The validation of the system then continues with the detection of unexpected interactions between pairs of features.

Feature	Test Process	Purpose According to Preconditions	Used Common Behaviour	Number of Global Sequences
INTL	tINTL1	TeenTime not restricted: allow call.	POTS_1	29
	tINTL2	TeenTime restricted, valid PIN: allow call.	cINTL1	30
	tINTL3	TeenTime restricted, invalid PIN: do not allow call.	cINTL2	2
CND	tCND1	Terminator idle: display.	cCND1	84
	tCND2	Terminator busy: do not display.	POTS_1	2
INFB	tINFB5	Terminator idle: affect billing.	POTS_1	29
	tINFB2	Terminator busy: do not affect billing.	POTS_1	2
TCS	tTCS1	Terminator idle, A not on Screened B: allow call.	cTCS1	29
	tTCS2	Terminator busy, A not on Screened B: busy tone.	cTCS2	2
	tTCS3	A on Screened B: announce screened message.	cTCS3	2

Table 1 Description of Test Processes for Individual Features

5 DETECTING FEATURE INTERACTIONS

5.1 Test Cases for Detecting FI

The tests generated in this section come from step (5) in Figure 1. In theory, if the same type of integration used for merging the individual UCMs and if the same composition used in the stub processes are used again during the generation of the test cases for pairs of features, then we should not find any inconsistencies, and perhaps not a single unexpected interaction. In practice however, integrating two features in a test sequence is much easier than integrating n features in a system (where $n > 2$). This is one of the main reasons why tests for pairs of features are necessary. Although they cannot cover everything there is to check, they represent a pragmatic and efficient way of attacking the problems of conformance to the requirements and interoperability between features.

Having a set of four features, we have to check $n*(n-1)/2 = 4*(4-1)/2 = 6$ different pairs of features[1]. We developed a test suite composed of six test processes (lines 2124 to 2864), described in Table 2. Each process contains many test cases that have different initial configurations. Their number can be found in the same table, as well as an enumeration of the common behaviour processes used, and the number of global sequences generated by LOLA using the *TestExpand* command.

We do not intend to explain the purpose of each of the 25 test cases. Comments in the code of the tests provide that information. As an example, we nevertheless present the purpose of the three test cases that validate the pair INTL-CND.

1. In this study, the assumption is that a feature cannot interact with itself. This is however an incorrect assumption in general. Hence, we would also need to cover the pairs INTL-INTL, INFB-INFB, CND-CND, and TCS-TCS. The numbers of pairs would become $n*(n+1)/2 = 4*(4+1)/2 = 10$.

FI Test Process	Number of Test Cases	Used Common Behaviour	Number of Global Sequences
fiINTL_CND	3	cCND1, cINTL2	170
fiINTL_INFB	3	POTS_1, cINTL1, cINTL2	61
fiCND_INFB	2	cCND1, POTS_1	86
fiINTL_TCS	9	cTCS1, cTCS2, cTCS3, cINTL1, cINTL2	74
fiCND_TCS	4	cCND1, cTCS2, cTCS3	90
fiINFB_TCS	4	cTCS1, cTCS2, cTCS3	35

Table 2 Description of Test Processes for Pairs of Features

Among the four combinations of its two preconditions, INTL has only 3 cases to check (TeenTime not restricted, TeenTime restricted and valid PIN, TeenTime restricted and invalid PIN), whereas CND has two other cases (terminator busy, terminator idle), unrelated to the ones of INTL. A Cartesian product would give us a total of 6 global cases. However, the 3 cases where the terminator is busy are not interesting to us. For these cases, CND acts exactly like POTS would. Hence the pair INTL-CND would act like the pair INTL-POTS, or in other words simply INTL, already covered by `tINTL1`, `tINTL2`, and `tINTL3`. With this purified domain partitioning, the three remaining cases provide new constraints on the values to be used in the preconditions attached to the global UCM (Figure 6). In essence, this UCM specifies the way these two features are integrated together, and the test cases have to reflect their end-to-end behaviour accordingly:

- Terminator idle, TeenTime not restricted (case 1): the end-to-end UCM acts like `cCND1` from the user's point of view.
- Terminator idle, TeenTime restricted and valid PIN (case 2): the end-to-end UCM acts first like `cINTL1` (part about the request for the PIN) and then like `cCND1` (display of the number) from the user's point of view. Unfortunately, because our common behaviour processes specify all events until the end of a scenario, `cINTL1` cannot be used as is and must be partly duplicated in `fiINTL_CND`, and then `cCND1` can be used.
- Terminator idle, TeenTime restricted and invalid PIN (case 3): the end-to-end UCM acts like `cINTL2` from the user's point of view.

If we had explicitly tested all the events that are currently hidden in the system, it would have been much more difficult to define reusable common behaviour. Having the system specified as a black box increases the reusability of these processes, although they do not ensure that what happens inside the system corresponds to what would be expected. We can only assume that if the end result is fine, then the system behaved properly.

5.2 Unexpected Interactions

With our first specification, all our test cases passed successfully, except for `fiINFB_TCS` (steps (7), (8), and (9) in Figure 1). LOLA returned three different traces that led to unexpected deadlocks. The first trace, presented below, is related to the first test case in `fiINFB_TCS`: the idle terminator (B) has subscribed to INFB and TCS, and the originator is not on the screening list. In this scenario, the originator (A) on-hooks first, but it is also billed instead of the terminator.

```
init  ! insert(sub(usera,noflist,undefined,undefined,noaddlist,validpin),
        insert(sub(userb,insert(tcs,insert(infb,noflist)),undefined,undefined,
                                       insert(userc,noaddlist),validpin),nosdb))
      ! nostatus ! noscpdb ! inittime;
offhook  ! usera;
dialtone  ! usera;
dial  ! usera ! userb;
startar  ! usera ! userb;
startr  ! userb ! usera;
offhook  ! userb;
stopar  ! usera ! userb;
stopr  ! userb ! usera;
i; (* time ! inittime *)
i; (* logbegin ! usera ! userb ! usera ! inittime *)
onhook  ! usera;
disconnect  ! userb ! usera;
i; (* time ! tic(inittime) *)
i; (* logend ! usera ! userb ! tic(inittime) *)
onhook  ! userb;
i; (* exit (0) *)
stop
```

Since we know on which network interfaces these events occurred, we can represent such LOTOS traces as synchronous MSCs, a form more appropriate for illustration of linear scenarios. The MSC for this trace is shown in Figure 21.

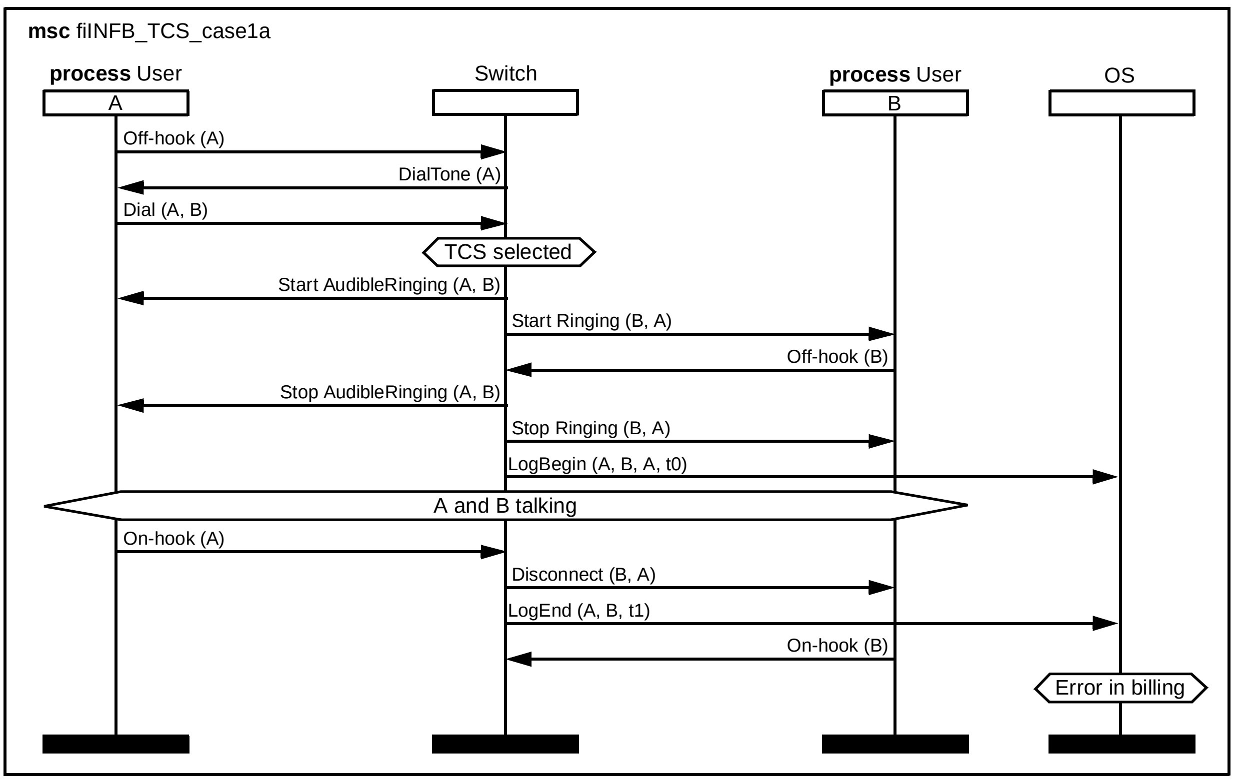


Figure 21 First FI, Originator Billed Instead of the Terminator

The error in the billing was detected when the test case queried the log from the OS and could not synchronize on the expected value. The problem here is that TCS was selected, but not INFB. Hence, the person to be billed was the default one, i.e., the originator.

The second interaction trace is similar in nature, but this time the terminator on-hooks first (Figure 22).

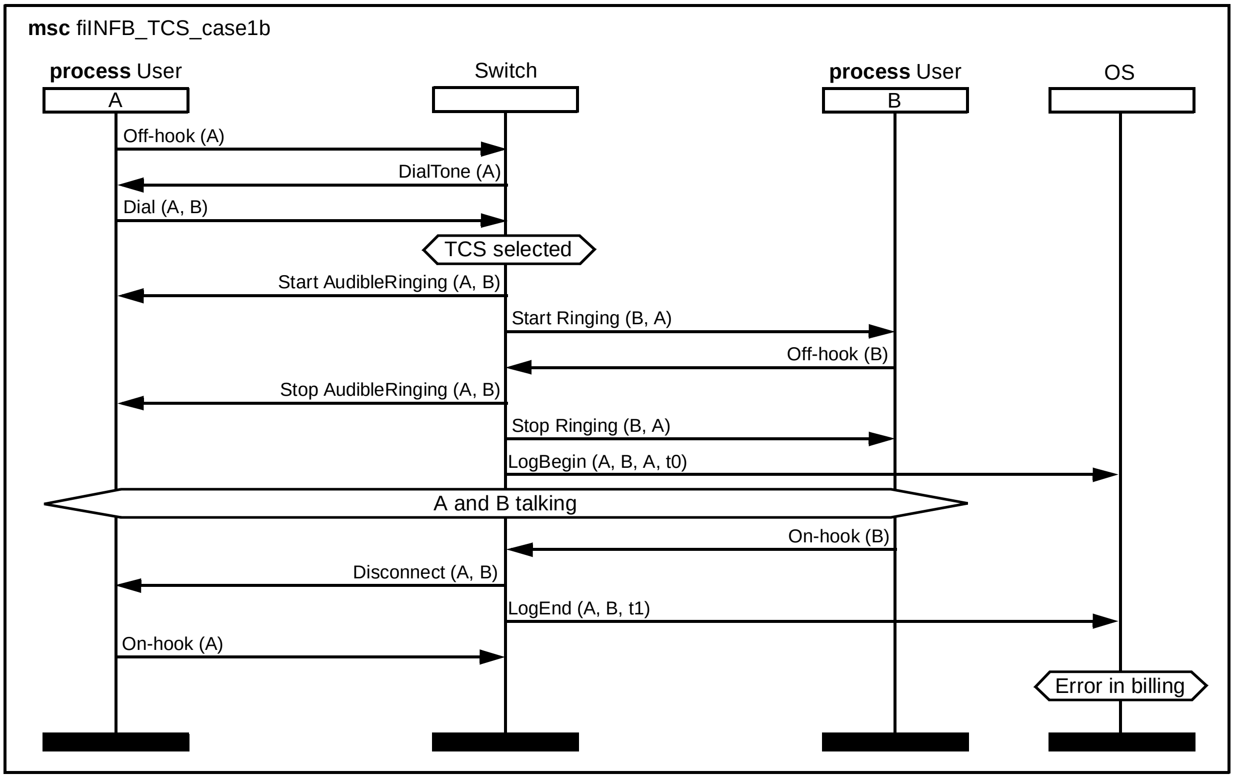


Figure 22 Second FI, Originator Billed Instead of the Terminator (Who On-hooks First)

The last interaction trace is related to the fourth test case in `fiINFB_TCS`: the idle terminator (B) has subscribed to INFB and TCS, and the originator is on the screening list (Figure 23). The call should be blocked by TCS, but it goes through because INFB was selected and not TCS. The deadlock occurs when the test case expects a specific announcement (ScreenedMessage) while the switch attempts to send something else (StartAudibleRinging or StartRinging).

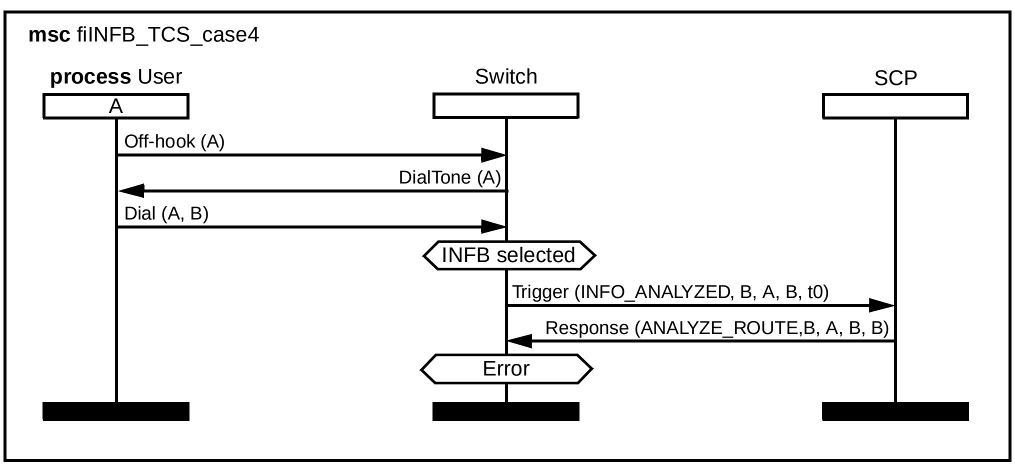


Figure 23 Third FI, Call Should Be Blocked but Is Not

Appendix B presents the erroneous part of our original specification. The choice between the TCS plugin and the INFB plugin in the **process-call** stub, which both override the default plugin, was left open (i.e., non-deterministic). When integrating the UCMs, we did not know if other types of constraints were necessary for these two features to work properly together. Even from a UCM perspective, a mutual exclusion would cause problems, but this is a detail that was somewhat buried down in the composition within the stub. This is why a more precise and rigorous detection technique appears necessary once the integration is completed.

A sensible solution to this problem would be to give a sequential priority to TCS over INFB in the stub, i.e., INFB would be selected only if TCS allows it. The specification in Appendix A implements this solution. In the end, all of our test cases (POTS, individual features, and pairs of features) passed successfully, and hence no expected interactions seemed to remain in the global specification.

Fixing the UCM

Giving TCS priority over INFB (and over the other features in the **process-call** stub) can be reflected back at the UCM level in different ways. One simple way, which does not necessarily reflect the stub structure in the current LOTOS specification, would be to move the TCS checking at a higher level that what it used to be in **process-call**. Therefore, we move the first condition from Figure 12 to Figure 9 and we remove the TCS plugin. As a result, the paths around this stub would be as prescribed by Figure 24:

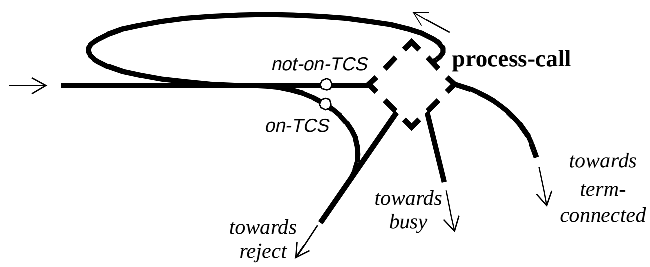


Figure 24 New Surroundings of Process-call Stub in Figure 9

In this figure, it is important for the feedback loop (used by call-forwarding features) to go back *before* the TCS screening list is checked. This is to ensure that intermediate originators in a forwarded call will be screened as well. Fixing a UCM could result in new types of interactions that were not present previously. For instance, the checking of the condition *on-TCS* is meaningful only if the terminator has subscribed to the TCS feature (otherwise, TCS becomes mandatory). This is why the test suite needs to be reapplied to the new resulting specification (this is called regression testing).

5.3 Ensuring Coverage with Probes

The generation of test cases from scenarios is an *a priori* approach to validation. We assume that the functional coverage is achieved when all tests execute as planned. However, the quality of the specification and of the test suite may be enhanced by using a syntactic approach called *structural coverage*. If some required coverage is not reached, new tests can be added *a posteriori*.

Probe insertion is a well-known white-box technique for monitoring software in order to identify portions of code that have been exercised, or to collect information for performance analysis. A program is instrumented with probes (generally counters initially set to 0) without any modification of its external functionality. Test cases "visit" these probes along the way, and the counters are incremented accordingly. Probes that have not been visited might indicate that the test suite is incomplete or that part the code is not reachable.

We have adapted this approach for LOTOS specifications (steps (6) in Figure 1). Special `(*_PROBE_*)` comments are added at specific places in the specification, and then they are translated automatically into hidden `Probe` gates with unique identifiers. Careful insertion of probes leads to a new specification that is observationally equivalent to the original one, and therefore they do not affect the verdicts of the tests. During testing, labeled transition systems (LTSs) resulting from the composition of each test with the specification can be generated, and occurrences of probes counted. If a probe is not visited by any of the test cases, then the structural coverage of the specification is incomplete. More specifically, this indicates that some code could be unreachable in the specification, or that the test suite is incomplete.

We inserted 55 probes in the specification of the system only (no need to cover the behaviour of the test as this is done through plain testing). Out of these, 5 were not covered by the whole test suite, but for good reasons (see Table 3). Therefore we conclude that the structural coverage of the specification by the validation test suite is adequate, and that no further test cases are required.

Probe Number	Line Number	Reason For Not Being Covered
P_35	1339	Case not specified yet. `OutBusy2` is used by one of the remaining unspecified features.
P_37	1353	Case not specified yet. `OutPC4` is used by one of the remaining unspecified features.
P_52	1509	Case not specified yet. The air interface is to be used by the Cellular feature.
P_53	1513	Case not specified yet. The air interface is to be used by the Cellular feature.
P_54	1518	Bug in LOLA's *TestExpand*. The OS is not reinstantiated as the occurrence of the internal probe is not forced by a test case.

Table 3 Probes Not Covered by the Test Suite

Note that we also measured the coverage of the test suites for POTS/individual features and for pairs of features. Both test suites covered all probes except the five already mentioned.

6 DISCUSSION

6.1 Adding New Features

Adding new features has a direct impact on the global UCM, the specification, and the test suite. Here are come comments on the scalability of this approach.

Impact on the UCMs

In our experience, the integration of three new features to the first 10 ones, which were already integrated together, did not have a major impact on the global UCM. The root map did not have to change, but a **busy** stub (with a new output path) had to be added in Figure 9. The disconnection UCM was slightly adjusted, and new plugins were created for the **process-call** and **billing** stubs.

The impact is probably proportional to how coupled the features are in a map. The more a map is decoupled and modular (for instance, by using stubs), the less likely major modifications will be necessary. More experiments on this aspect still need to be done in order to have better conclusions.

Impact on the Specification

Since the specification reflects the global UCM, the conclusions are basically the same as for the impact on the UCMs. In our experiment, we added a few new gates (for the new air interface) and added appropriate ADTs, with their operations, to support new data structures. Some previous types were also expanded to cover the new features. The impact on the structure is not really known because we have not fully implemented these three features yet.

Impact on the Test Suite

The addition of a feature has a profound impact on the test suite. Like before, we will distinguish between test suites for individual features and test suites for pairs of features. Each feature has a set of c preconditions (enumerated in Table 1), the conditions that may affect the result for one same stimulus. In theory, since each of these is either true or false, the upper bound on the number of combinations is $u = 2^c$. That is, the number of test cases for each feature grows exponentially with the number of preconditions, which is no surprise. We need one test case for each of these combinations, unless some of these are unnecessary. UCMs can help determine which are relevant and which are not by following the paths and associated conditions. For instance, when *TeenTime* is not restricted in INTL, whether the PIN is valid or not has no impact, and one of these two combinations can be dropped. Therefore, due to this test selection approach, there is a potential gain at this level. We define this gain as $g = u\text{-}t$, where t is the number of test cases actually present in our test suite (Table 4).

Feature	Number of Conditions (c)	Theoretical Upper Bound ($u = 2^c$)	Actual Number of Test Cases (t)	Gain ($g = u - t$)
INTL	2	4	3	1
CND	1	2	2	0
INFB	1	2	2	0
TCS	2	4	3	1

Table 4 Number of Test Cases for Individual Features

For pairs of features, UCMs can help again reducing the number of necessary test cases w.r.t. the upper bound. The number of conditions for a pair (c) is the cardinality of the union of the two sets of preconditions (as some conditions may be shared, which is a major cause of interaction), and the theoretical upper bound is again $u = 2^c$. However, we already know from the previous table which cases have to be looked at. Hence, we define p as the product of the two number of actual test cases ($t_1 * t_2$). This leads us to a better upper bound (b) defined as the minimum of u and p. As explained at the end of Section 5.1, we can get a better partitioning of the input domain by removing the cases that are equivalent, from a path point of view, to cases already covered in the test suite for individual features. Another gain can be achieved here ($g = b\text{-}t$), illustrated in Table 5.

Pair of Features	Number of Distinct Conditions (*c*)	Theoretical Upper Bound ($u = 2^c$)	Product of Number of Cases (*p*)	Better Upper Bound *b = MIN(u, p)*	Actual Number of Test Cases (*t*)	Gain (*g = b - t*)
INTL-CND	3	8	3*2 = 6	6	3	3
INTL-INFB	3	8	3*2 = 6	6	3	3
CND-INFB	1	2	2*2 = 4	2	2	0
INTL-TCS	4	16	3*3 = 9	9	9	0
CND-TCS	2	4	2*3 = 6	4	4	0
INFB-TCS	2	4	2*3 = 6	4	4	0

Table 5 Number of Test Cases for Pairs of Features

Note that a negative gain implies a non-optimal selection of test cases. This represents a simple way to measure the efficiency of this selection.

In conclusion, the number of pairs of features is $n*(n-1)/2$, and for each the number of test cases grows exponentially with the number of distinct conditions. The impact of the integration of a new feature will be higher if new types of conditions have to be accounted for in the input domain.

6.2 Performance

From a tool perspective, testing with LOLA seems a very efficient solution for the validation of prototypes and the detection of feature interactions. The compilation of this 2864-line specification and the execution of all the test cases take about 30 seconds on a low-end PC (Cyrix P150, Win95, 48MB RAM). This means that this technique can be used in an iterative and incremental process where numerous modifications, additions, debugging sessions, and executions of regression test suites need to be supported.

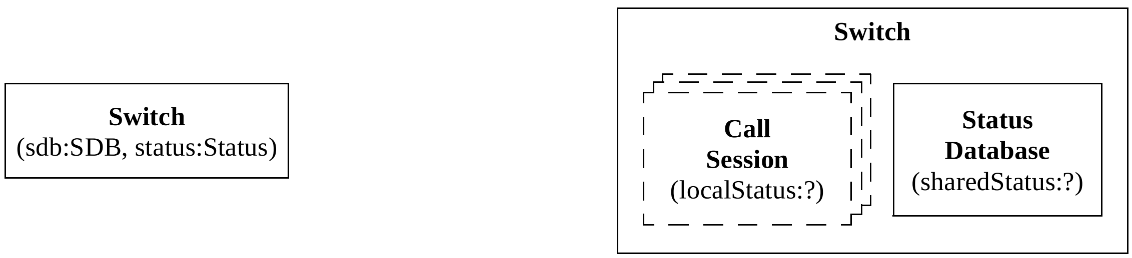
The verification of the structural coverage (with probes), which is usually performed towards the end of a macro-iteration in the design cycle, takes about 7 minutes of processing time on the same platform. For this part, internal actions must not be simplified through on-the-fly equivalence reductions (otherwise, we could not “observe” the probes in the resulting LTSs), and thus more time and resources are required.

Once probes are inserted, some specifications may result in a number of states too large to be handled by LOLA and similar tools. We see at least three practical solutions to this problem

- Use half the probes for a first measure, then use the other half for a second one. The set of probes visited is the union of the probes visited in each experiment. This is the best solution.
- Simplify the test processes by splitting them into many sub-tests (with an equivalent testing power).
- Use heuristics in the execution of the tests. LOLA allows to test a specification according to upper bounds in memory usage, or according to an estimate of the coverage of the possible branches. Although the functional coverage is not complete anymore, experience shows that a complete structural coverage might be achieved anyway.

6.3 Improved Call Structure

The abstract underlying structure in our UCMs is insufficient for the specification of all the features. The current behaviour of our switch is tightly coupled to the progression of one unique call session. For call sessions involving more than two parties (e.g., for the support of features such as INBL, INFR, 3WC, INCF and CW), the current call structure needs to be improved. Call sessions need to be instantiated upon request, and the status database needs to be decoupled from the current Switch process (Figure 25(a)) in order to be accessible to these sessions. A new **StatusDatabase** process (where appropriate values from *sdb* and *status* would be stored), with new query and update messages, would solve this problem (Figure 25(b)). ADTs need to be partitioned accordingly, and all of the specification must then reflect this modification. This kind of structure is similar to those used in many LOTOS specifications for telephony systems [20][22][23]. IN-like architectures, and especially the Basic Call Model, could also be considered.



(a) Current Switch (b) Improved Switch

Figure 25 Current and Improved Switch Structures

The UCM structure, derived from the given network structure (Figure 3), did not specify anything about the internals of the switch. To keep our synthesis straightforward, we did not introduce any new structural entities. However, for the sake of extensibility of the specification, this improvement can hardly be avoided. New components are needed at the specification level, and they probably need to be mirrored at the UCM level. However, this is left for future work.

6.4 Limitations of Plugins, Bindings, and Composition

We have observed the following limitations of the UCM notation while integrating the scenarios:

- Although the stub/plugin mechanism is useful for abstraction, modularity, and dynamic behaviour, its use in a global map makes the end-to-end scenarios more difficult to visualize at a first glance. Often, the reader has to mentally flatten the global map to get a better understanding of these scenarios.
- The binding of a plugin to a stub is done through an external mechanism (the binding relation), which is not visual.
- The composition of plugins in a stub is described at a lower level of abstraction. Not having this information (whether it is visual or not) at the UCM level makes the selection of plugins very ambiguous. On the other end, it allows for the designer to play with different alternatives and to decide which composition should be used. But the problem remains that once this composition has been decided, it is documented with another notation (LOTOS in our case).

Designers must use the stub/plugin mechanism with care. Otherwise, they might defeat the intent of UCMs which is to provide a good bird's eye view of the system.

6.5 Comparison with Other Techniques

We include here a short discussion on three related approaches to design with UCM and to detection of FI.

Agent Systems

A path that goes from UCMs to agent prototypes was illustrated by a feature interaction example in [15][16]. This approach also aims to avoid interactions at design time (with UCMs), but the main property of these agents (implemented in CLIPS and Java) remains the opportunistic avoidance of interactions at *run time*. However, the mapping between UCMs and these agents is still fairly immature, and detection/validation techniques on this agent environment are still ad hoc. For these reasons, and also because the FI contest was mainly about detection at design time (not avoidance), we took a somewhat different direction that led to the current exercise. How LOTOS would fit in a design process that involves UCMs and such agents is still the topic of future research.

GCS and GPRS

Design and validation of LOTOS specifications from UCMs have been performed for two previous projects: a *Group Communication Server* (GCS) in [4][6], and the packet-switched mobile telephony standard *General Packet Radio Services* (GPRS) in [5]. In both cases, the integration of the UCM scenarios was done directly at the LOTOS level. There was no global UCM, and no stub was used. Test cases were generated solely from the UCMs (requirements were in plain English, without anything similar to Chisel diagrams) and applied to the specification in order to validate the integration.

Since the burden of the integration is pushed down to the level of LOTOS, designers not too familiar with this language may have a hard time coping with the construction of the specification. Moreover, other people not involved in the LOTOS part (clients, marketing, management, etc.) would not know anything about how the individual UCMs fit together. For these two reasons, although the GCS and GPRS experiments were successful in the sense that moving to LOTOS directly also resulted in correct specifications and validated test suites, integrating the scenarios at the UCM level seems a better alternative.

In the GCS case study, we derived rejection test cases for each of the individual scenarios (as illustrated in Figure 16). In the current example, doing so will require a better knowledge of what could go wrong in the system (which can be anything right now). If Chisel diagrams had included branches labeled “reject”, “interdicted” or “forbidden”, directly in the requirements, then generating rejection test cases would have been easy. Again, an OPI-like notation would be of great help in this context.

Faci’s Approach

In [22], Faci presents a detection technique also based on the integration of scenarios and the use of the LOTOS testing theory. This approach makes a distinction between the concepts of composition and integration. *Composition*, noted $f_1|[]|f_2$, expresses the synchronization of features on their common actions with POTS and their interleaving on their independent actions. *Integration*, noted $f_1 * f_2$, expresses the extension of POTS with *n* features (two in the examples), such that each feature is able to execute all of its actions which are allowed in the context of POTS, when the other features are disabled. Features are captured as labeled transition systems (LTSs) instead of as UCMs. Integration relates very well to our own UCM integration (validated by the test cases for individual features), whereas the composition simply represents the generalized synchronization operator and does not relate to anything in our methodology. The approach states that an interaction exists between *n* features if their *integration* does not *conform* to their *composition*.

Conformance is checked through validation test cases, from the user's point of view, similarly to what we are doing. Test cases are derived manually (using "knowledge and experience") from the composition specification, and then they are applied to the integration specification. When a deadlock occurs between a test case and the integration specification, an interaction is said to be detected. This last specification is generated manually at the LTS level, which is far less scalable and modular than generating specifications from global UCMs. Indeed, all the examples provided in this thesis contained only pairs of features integrated together, for obvious complexity reasons. UCMs are a means to integrate scenarios while avoiding some interactions, and they allow for multiple complex features to be considered (13 in [35]).

Faci's approach leads to multiple feature interactions that we already referred to as trivial and artificial. Indeed, any integration operator (*) other than the generalized synchronization (|[]|) is very likely to cause deadlock situations. The test suite, although it could be generated almost automatically from the composition, is of low quality as it does not consider the way the features were integrated together. The test suites generated from UCMs are much more representative of the intended system behaviour, and they are more likely to be reusable down the road towards the implementation.

7 CONCLUSIONS

This report presented an approach for the avoidance and detection of feature interactions at design time. Features are captured as UCM scenarios, integrated in one global map with stubs and plugins, and then transformed into a LOTOS specification. Test cases are generated from the requirements (Chisel diagrams in our case) and from the UCMs. We use them to validate the integration and to detect unexpected interactions.

UCMs describe features (and systems in general) at an interesting level of abstraction. We showed how, during their integration, some interactions can be avoided by insuring deterministic and complete preconditions and by composing plugins in stubs according to the intent of the features. Many features can be considered in a global UCM, and they can be represented as one or more plugins in one or more stubs. Further design decisions are necessary when synthesizing the specification, although the burden of the integration is mostly taken care of at the UCM level. The canonical tester theory and test selection techniques based on UCMs help us generate reduced sets of test cases for individual features. Test suites for detecting interactions between pairs of features are constructed on top of existing test cases, hence promoting reuse and consistency among tests. The generation of these tests is guided by the integration done at the UCM level, which again reduces the number of necessary cases to cover. Several interactions between a pair of features were detected. They were caused by the composition of plugins in a stub, and this is where we fixed the problem at the LOTOS and UCM levels. The quality of the specification and of the validation test suite is finally assured by measuring the structural coverage through probe insertion. Good tool support for the UCM integration (UCM Navigator) and for the validation and coverage measurement of the LOTOS specification (LOLA) suggests that this approach can be used in an iterative and incremental design process.

Future Work

The following list enumerates several research issues and work items, some of which were already raised in this document:

- Improvement of the call process within the Switch for the support of features involving more than two users.
- Completion of the specification by integrating the remaining nine features. By observing the impact on the specification and on the number of test cases required for validation, it might be possible to learn new lessons.
- Derivation of rejection test cases from UCMs and/or the Chisel diagrams in order to detect more interactions.
- Comparison with other LOTOS-based techniques applied to the same set of features, by detecting interactions in our specification with their approaches (if the tools allow it) and by applying our test cases to their specifications. We could also observe how "trivial and artificial" interactions detected with their techniques have been avoided by our UCMs.
- Linkage of the OPI model to the UCM notation. The intent of a feature would be better described by indicating which events or paths are obliged, permitted, or forbidden to be in the implementation. This would also allow for an easy way of generating rejection test cases.
- Further study of the visualization of bindings and compositions of plugins.
- Finally, we could look at the best way of integrating this approach in a design process that generates agent prototypes from Use Case Maps.

Acknowledgements

This work has been supported by *Communications and Information Technology Ontario* (CITO). I would like to thank Dorin Petriu for providing me with the global UCM used in this case study. Andrew Miga and Chris Witham supported us with the UCM Navigator tool. I am grateful to Luigi Logrippo for useful and constructive comments on a first draft of this report. Finally, many thanks to the agent group at Carleton U. and Mitel, and to the LOTOS group at U. of Ottawa for useful discussions.

8 REFERENCES

[1] Aho, A., Gallagher, S., Griffeth, N., Scheel, C., and Swayne, D. (1998) "Sculptor with Chisel: Requirements Engineering for Communications Services". In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 45-63. http://www-db.research.bell-labs.com/user/nancyg/sculptor.ps

[2] Amyot, D. (1994) *Formalization of Timethreads Using LOTOS*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada. http://www.csi.uottawa.ca/~damyot/phd/msctheses.pdf

[3] Amyot, D., Bordeleau, F., Buhr, R. J. A., and Logrippo, L. (1995) "Formal support for design techniques: a Timethreads-LOTOS approach". In *FORTE VIII, 8th International Conference on Formal Description Techniques*, Montréal, October 1995. Chapman & Hall, 57-72. http://lotos.csi.uottawa.ca/~damyot/phd/forte95/forte95.pdf

[4] Amyot, D., Logrippo, L., and Buhr, R.J.A. (1997) "Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage". In: *CFIP 97, Ingénierie des protocoles*, Liège, Belgique, September 1997. http://www.csi.uottawa.ca/~damyot/cfip97/cfip97.pdf

[5] Amyot, D., Hart, N., Logrippo, L., and Forhan, P. (1998) “Formal Specification and Validation using a Scenario-Based Approach: The GPRS Group-Call Example”. In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Ottawa, Canada, January 1998. http://www.csi.uottawa.ca/~damyot/wrroom98/wrroom98.pdf

[6] Amyot, D. (1998) *Group Communication Server: A Scenario-Based Design Exercise*. Telecommunication Research Institute of Ontario, report #1388, Ottawa, Canada, June 1998. http://www.csi.uottawa.ca/~damyot/gcs/

[7] Barbuceanu, M., Gray, T., and Mankovski, T. (1998) “How To Make Your Agents Fulfil Their Obligations”. In: H.S. Nwana and D.T. Ndumu (Eds), *PAAM’98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998, 255-276.

[8] Boumezbeur, R. and Logrippo, L. (1993) “Specifying telephone systems in LOTOS”. *IEEE Communications Magazine*, 31 no. 8 (August), 38-45. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Papers/svtsl.ps.Z

[9] Brinksma, E. (1988) “A theory for the derivation of tests”. In: S. Aggarwal and K. Sabnani (Eds), *Protocol Specification, Testing and Verification VIII*, North-Holland, 63-74, June 1988.

[10] Buhr, R.J.A. and Casselman, R.S. (1995) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA.

[11] Buhr, R.J.A. (1997) *High Level Design and Prototyping of Agent Systems*, research project description. http://www.sce.carleton.ca/rads/agents/

[12] Buhr, R.J.A., Elammari, M., Gray, T., Mankovski, S., and Pinard, D. (1997) “Understanding and Defining the Behaviour of Systems of Agents, with Use Case Maps”. Poster session, *PAAM’97, Second Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, April 1997. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam97.pdf

[13] Buhr, R.J.A., Elammari, M., Gray, T., and Mankovski, S. (1998) “A High Level Visual Notation for Understanding and Designing Collaborative, Adaptive Behaviour in Multi-agent Systems”, *Hawaii International Conference on System Sciences (HICSS’98)*, Hawaii, January 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/agents-ucms.pdf

[14] Buhr, R.J.A., Elammari, M., Gray, T., and Mankovski, S. (1998) “Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example”, *Hawaii International Conference on System Sciences (HICSS’98)*, Hawaii, January 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hiccs98.pdf

[15] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) “High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example”. In: H. S. Nwana and D. T. Ndumu (eds.), *PAAM’98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam98.pdf

[16] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) “Feature-Interaction Visualization and Resolution in an Agent Environment”. In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 135-149. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/fiw98.pdf

[17] Buhr, R.J.A. (1998) “Use Case Maps as Architectural Entities for Complex Systems”. To appear in: *Transactions on Software Engineering*, IEEE, 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/tse98final.pdf

[18] Cameron, E.J., Griffeth, N., Linand, Y.-J., Nilson, Y.-J., Schnure, W.K. and Velthuijsen, H. (1994) “A Feature Interaction Benchmark for IN and Beyond”. In: L. G. Bouma and H. Velthuijsen (eds), *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands, May 1994. IOS Press, 1-23. http://www-db.research.bell-labs.com/user/nancyg/benchmark.ps

[19] Faci, M., Logrippo, L. and Stépien, B. (1989) “Formal Specification of telephone systems in LOTOS”, *Protocol Specification, Verification and Testing*, IX, North-Holland.

[20] Faci, M., Logrippo, L., and Stépien, B. (1991) “Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach”. *Computer Networks and ISDN Systems*, 21 (1991) 53-67. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Papers/telephone.CNIS9007.ps.Z

[21] Faci, M. and Logrippo, L. (1994) "Specifying Features and Analysing their Interactions in a LOTOS Environment". In: L. G. Bouma and H. Velthuijsen (eds), *Second International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 136-151. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Papers/Fits94.CameraReady.ps.gz

[22] Faci, M. (1995) *Detecting Feature Interaction in Telecommunications Systems Designs*. Ph.D. thesis, Department of Computer Science, University of Ottawa, November 1995. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Theses/mf_phd.ps.gz

[23] Faci, M., Logrippo, L., and Stépien, B (1997) "Structural Models for Telephone Specifications". In: *Computer Network & ISDN Systems*, 29 (1997) 501-528. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Papers/isdn95.ps.gz

[24] Griffeth, N.D. and Velthuijsen, N.D. (1994) "The Negotiating Agents Approach to Runtime Feature Interaction Resolution". In: L. G. Bouma and H. Velthuijsen (eds), *Second International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS press, 217-235. http://www-db.research.bell-labs.com/user/nancyg/fiw94.ps

[25] Griffeth, N.D., Tadashi, O., Grégoire, J.-C. and Blumenthal, R. (1998) "First Feature Interaction Detection Contest". In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 327-359. http://www.tts.lth.se:80/FIW98/contest.html

[26] ISO (1989), Information Processing Systems, Open Systems Interconnection, "LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", IS 8807.

[27] ISO/EIC (1991), Information Technology, Open Systems Interconnection, "Conformance Testing Methodology and Framework (CTMF)", IS 9646, ISO, Geneve. Also: CCITT X.290-X.294.

[28] ITU (1995), *Q.1200 (General) Recommendation Series*. Geneva.

[29] ITU (1996), "Recommendation Z. 120: Message Sequence Chart (MSC)". ITU, Geneva.

[30] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1993) *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, ACM Press.

[31] Kamoun, J. (1996) *Formal Specification and Feature Interaction Detection in the Intelligent Network*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Theses/jk_msc.ps.gz

[32] Kamoun, J. and Logrippo, L. (1998) "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model". In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press.

[33] Kimbler, K. and Søbirk, D. (1994) "Use case driven analysis of feature interactions". In: L. G. Bouma and H. Velthuijsen (eds), *Feature Interactions in Telecommunications Systems*, Amsterdam, The Netherlands, May 1994. IOS Press, 167-177.

[34] Miga, A. (1998) *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/am_thesis.pdf

[35] Petriu, D. (1998) *Feature Interaction Detection and Avoidance — Smart Design of Telephony System with Use Case Maps*. CITO report, Ottawa, Canada. To appear.

[36] Pavón, S., Larrabeiti, D., and Rabay, G. (1995) *LOLA—User Manual, version 3.6*. DIT, Universidad Politécnica de Madrid, Spain, LOLA/N5/V10 (February).

[37] Quemada, J., Pavón, S. and Fernández, A. (1988) "Transforming LOTOS Specifications with LOLA: The Parametrized Expansion". In: K. J. Turner (Ed), *Formal Description Techniques, I*, IFIP/North-Holland, 45-54.

[38] Stépien, B. and Logrippo, L. (1995) "Feature Interaction Detection using Backward Reasoning with LOTOS". In: S. Vuong (ed.), *Protocol Specification, Testing and Verification XIV*, Vancouver, 71-86. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Papers/pstv.94.book.ps.Z

[39] Stépien, B. and Logrippo, L. (1995) “Representing and Verifying Intentions in Telephony Features using Absract Data Types”. In: K. E. Cheng and T. Ohta (eds.), *Third International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press, 141-155. http://lotos.csi.uottawa.ca/~bernard/intention.ps.Z

[40] Tuok, R. (1996) *Modeling and Derivation of Scenarios for a Mobile Telephony System in LOTOS*. M.Sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Theses/rt_msc.ps.gz

[41] Turner, K.J. (1998) “Validating Architectural Feature Descriptions using LOTOS”. In: K. Kimbler and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications Software Systems*, IOS Press.

A LOTOS SPECIFICATION

Here is the fully commented LOTOS specification derived from our Use Case Maps. It contains the following elements:

- Modification history: lines 1 to 58.
- Definition of observable gates/events: lines 59 to 79.
- Basic data structures and operations (ADTs simpler than the International Standard's): lines 80 to 227.
- Data structures and operations for features: lines 228 to 853.
- Processes representing the components, the stubs and the plugins: lines 854 to 1533.
- POTS common behaviour: lines 1534 to 1647.
- POTS test cases: lines 1648 to 1714.
- Test cases for individual features: lines 1715 to 2123.
- Test cases for detecting interactions between pairs of features: lines 2124 to 2864.

```
1    (**************************************************************************)
2    (*  Feature Interactions from UCM                                         *)
3    (*  Version: 0.14c                                                        *)
4    (*  Date   : September 8, 1998                                            *)
5    (*  Authors: Daniel Amyot (damyot@csi.uottawa.ca)                         *)
6    (*           Dorin Petriu (dorin@sce.carleton.ca)                         *)
7    (*           SCE Department, Carleton University, Ottawa, Canada          *)
8    (*  History:                                                              *)
9    (*           Sep. 8, 1998 : Added Probes for structural coverage.         *)
10   (*           Aug. 19, 1998: Fixed part of INFB-TCS interaction in billing. *)
11   (*                          Solved the INFB-TCS FI in ProcessCallStub.     *)
12   (*           Aug. 13, 1998: Recomposed TCS as an alternative to INFB.     *)
13   (*                          No FI between TCS and CND, as expected.       *)
14   (*                          FI found between TCS and INFB in this way!    *)
15   (*           Aug. 12, 1998: TCS implemented and tested. No FI with INTL.  *)
16   (*           Aug. 11, 1998: New stub for Busy in Post-Dial. Conforms to the *)
17   (*                          new map with 13 features.                     *)
18   (*                          Added support for AirBegin and AirEnd (Cell). *)
19   (*                          Started implementing TCS (priority over others) *)
20   (*           July 12, 1998: Added INFB and 2 tests. The feature works.    *)
21   (*                          Added Query gate to OS.                       *)
22   (*                          All test cases now check the final billing Log. *)
23   (*                          Checking interactions between INTL and INFB.  *)
24   (*                            None found, as expected.                    *)
25   (*                          Checking interactions between CND and INFB.   *)
26   (*                            None found, as expected?                    *)
27   (*           July 10, 1998: Added a FList to users. CND now works.        *)
28   (*                          Restructured the test suite                   *)
29   (*                          Checks for interactions between CND and INTL. *)
30   (*                            None found, as expected.                    *)
31   (*           July 9, 1998 : Sets AudibleRinging and Ringing.              *)
32   (*                          Updated SCP. Now INTL works.                  *)
33   (*                          Updated the switch. CND almost works.         *)
34   (*           July 8, 1998 : Specified part of PostDialStub and            *)
35   (*                          ProcessCallStub to make POTS work. It does now. *)
36   (*           July 7, 1998 : Added lt and ge to type Time                  *)
37   (*                          Reimplemented SCPDB and query operations      *)
38   (*                          Reimplemented Status and query operations     *)
39   (*                          Created six generic POTS state processes      *)
40   (*                          Created two test processes for POTS and three *)
41   (*                            for INTL from the Chisel diagrams.          *)
42   (*                          Defined the mechanism for composing feature   *)
43   (*                            plugins in stubs.                           *)
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
44   (*           July 6, 1998 : Added types SPList, Status and SCPDB          *)
45   (*                          Added processes DisplayStub, PreDialStub      *)
46   (*                          Created process PostDialStub                  *)
47   (*                          Created Default & INTL plugins for PreDialStub *)
48   (*                          Worked on User and Switch for POTS/INTL       *)
49   (*                          Added two complex test processes for POTS.    *)
50   (*           July 5, 1998 : Added process GlobalClock                     *)
51   (*                          Added types Cadence, PIN, Message, TriggerName *)
52   (*                            ResponseType, Log, LogRecord, Feature, FList *)
53   (*                            SInfo, SDB, AddList                         *)
54   (*           July 4, 1998 : Added Boolean, adapted Address, and simplified *)
55   (*                          NaturalNumbers.                               *)
56   (*           July 2, 1998 : Created structure and process skeletons.      *)
57   (*           June 16, 1998: Modified IS8807 ADT.                          *)
58   (**************************************************************************)
59
60   specification FI_UCM[OffHook,      (* User2Switch                         *)
61                        OnHook,       (* User2Switch                         *)
62                        Dial,         (* User2Switch                         *)
63                        Flash,        (* User2Switch                         *)
64                        DialTone,     (* Switch2User                         *)
65                        StartAR,      (* Switch2User: Start AudibleRinging   *)
66                        StartR,       (* Switch2User: Start Ringing          *)
67                        StartCWT,     (* Switch2User: Start CallWaitingTone  *)
68                        StopAR,       (* Switch2User: Stop AudibleRinging    *)
69                        StopR,        (* Switch2User: Stop Ringing           *)
70                        StopCWT,      (* Switch2User: Stop CallWaitingTone   *)
71                        LineBusyTone, (* Switch2User                         *)
72                        Announce,     (* Switch2User                         *)
73                        Disconnect,   (* Switch2User                         *)
74                        Display,      (* Switch2User                         *)
75                        CreateUser,   (* NEW: For creating user instances.   *)
76                        Init,         (* NEW: Initialize switch for testing. *)
77                        Query         (* NEW: Allows to query OS' Log.       *)
78                       ]:noexit
79
80   (*====================================================*)
81   (*            Modified IS8807 ADT definitions          *)
82   (*====================================================*)
83
84   (* Types FBoolean, Element, and Set contain corrections *)
85   (* to the library from the International Standard 8870. *)
86   (* Type Boolean remains the same, but NaturalNumber was *)
87   (* simplified by removing unnecessary arithmetic and    *)
88   (* comparison operators                                 *)
89
90   type Boolean is
91   sorts
92       Bool
93   opns
94       true, false: -> Bool
95       not: Bool -> Bool
96       _ and _, _ or _, _ xor _,
97       _ implies _, _ iff _, _ eq _, _ ne _: Bool, Bool -> Bool
98   eqns
99       forall x, y: Bool
100      ofsort Bool
101        not (true) = false ;
102        not (false) = true ;
103        x and true = x ;
104        x and false = false ;
105        x or true = true ;
106        x or false = x ;
107        x xor y = x and not (y) or (y and not (x)) ;
108        x implies y = y or not (x) ;
```

```
      x iff y = x implies y and (y implies x) ;
      x eq y = x iff y ;
      x ne y = x xor y ;
endtype (* Boolean *)

(*********************************************)

type NaturalNumber is Boolean
sorts
    Nat
opns
    0: -> Nat
    Succ: Nat -> Nat
    _ + _: Nat, Nat -> Nat
    _ eq _, _ ne _: Nat, Nat -> Bool
eqns
    forall m, n: Nat
    ofsort Nat
      m + 0 = m ;
      m + Succ (n) = Succ (m) + n ;
    ofsort Bool
      0 eq 0 = true ;
      0 eq Succ (m) = false ;
      Succ (m) eq 0 = false ;
      Succ (m) eq Succ (n) = m eq n ;
      m ne n = not (m eq n) ;
endtype (* NaturalNumber *)

(*********************************************)

type        FBoolean is
formalsorts FBool
formalopns  true        : -> FBool
        not         : FBool -> FBool
formaleqns
    forall  x : FBool
    ofsort  FBool
    not(not(x)) = x;
endtype     (* FBoolean *)

(*********************************************)

type        Element is FBoolean
formalsorts Element
formalopns  _ eq _, _ ne _          : Element, Element -> FBool
formaleqns
    forall  x, y, z : Element
    ofsort  Element
        x eq y = true =>
    x       = y             ;

    ofsort  FBool
        x = y =>
    x eq y  = true          ;
        x eq y =true , y eq z = true =>
    x eq z  = true          ;

    x ne y  = not(x eq y)   ;
endtype     (* Element *)

(*********************************************)

type    Set is Element, Boolean, NaturalNumber
sorts   Set
opns    {}                                  :               -> Set
```



```
    Insert, Remove                          : Element, Set  -> Set
    _IsIn_, _NotIn_                         : Element, Set  -> Bool
    _Union_, _Ints_, _Minus_                : Set, Set      -> Set
    _eq_, _ne_, _Includes_, _IsSubsetOf_    : Set, Set      -> Bool
    Card                                    : Set           -> Nat

eqns    forall  x, y    : Element,
        s, t    : Set
    ofsort  Set

        x IsIn Insert(y,s) =>
    Insert(x, Insert(y,s))  = Insert(y,s)           ;
    Remove(x, {})           = {}                    ;
    Remove(x, Insert(x,s))  = s                     ;
        x ne y = true of FBool =>
    Remove(x, Insert(y,s))  = Insert(y, Remove(x,s));

    {} Union s              = s                     ;
    Insert(x,s) Union t     = Insert(x,s Union t)   ;

    {} Ints s               = {}                    ;
        x IsIn t =>
    Insert(x,s) Ints t      = Insert(x,s Ints t)    ;
        x NotIn t =>
    Insert(x,s) Ints t      = s Ints t              ;

    s Minus {}              = s                     ;
    s Minus Insert(x, t)    = Remove(x,s) Minus t   ;

    ofsort  Bool

    x IsIn {}               = false                 ;
        x eq y = true of FBool =>
    x IsIn Insert(y,s)      = true                  ;
        x ne y = true of FBool =>
    x IsIn Insert(y,s)      = x IsIn s              ;
    x NotIn s               = not(x IsIn s)         ;

    s Includes {}           = true                  ;
    s Includes Insert(x,t)  = (x IsIn s) and (s Includes t)    ;

    s IsSubsetOf t          = t Includes s          ;

    s eq t                  = (s Includes t) and (t Includes s);

    s ne t                  = not(s eq t)           ;

    ofsort  Nat

    Card({})                = 0                     ;
        x NotIn s =>
    Card(Insert(x,s))       = Succ(Card(s))         ;
endtype (* Set *)

(*============================================*)
(*          FI_UCM ADT definitions            *)
(*============================================*)

(* The Time type is mapped onto natural numbers. *)
type Time1 is NaturalNumber renamedby
sortnames
    Time for Nat
opnnames
    tic for succ
    initTime for 0
```

```
endtype (* Time1 *)

(* Additional comparison operators for time range. *)
type Time is Time1
opns
    _ lt _, _ ge _ : Time, Time -> Bool
eqns
    forall t1, t2 : Time
    ofsort Bool
      t1 lt initTime      = false ;
      initTime lt tic(t1) = true ;
      tic(t1) lt tic(t2)  = t1 lt t2 ;
      t1 ge t2            = not (t1 lt t2);
endtype (* Time *)

(*********************************************)

(* The Address type contains the Address sort, *)
(* which is an enumeration of user identifiers *)
(* or numbers that can be dialled.             *)
type Address is NaturalNumber
sorts Address
opns
    userA, userB, userC, anonymous, undefined, star69 : -> Address
    zeroPlus : Address -> Address
    map : Address -> Nat
    dest : Address -> Address
    _ eq _, _ ne _ : Address, Address -> Bool
eqns
    forall user1, user2 : Address
    ofsort Nat
      map(userA)            = 0;
      map(userB)            = succ(0);
      map(userC)            = succ(succ(0));
      map(anonymous)        = succ(succ(succ(0)));
      map(undefined)        = succ(succ(succ(succ(0))));  (* for CND *)
      map(star69)           = succ(succ(succ(succ(succ(0)))));  (* for RC *)
      map(zeroPlus(user1))  = succ(succ(succ(succ(succ(succ(0))))));  (* for CC *)
    ofsort Address
      dest(zeroPlus(user1)) = user1; (* for CC *)
    ofsort Bool
      user1 eq user2 = map(user1) eq map(user2);
      user1 ne user2 = not(user1 eq user2);
endtype (* Address *)

(* List of addresses, implemented as a set.                   *)
(* We avoid the problem with ISLA's renaming in actualization *)
type AddList0 is Set
actualizedby Address using
sortnames
    Address for Element
    Bool  for FBool
endtype (* AddList0 *)

type AddList is AddList0 renamedby
sortnames
    AddList for Set
opnnames
    NoAddList for {} (* Empty list of addresses *)
endtype (* AddList *)

(*********************************************)

(* The Cadence is either Ring or SpecialTone. *)
type Cadence is Boolean
```



```
sorts Cadence
opns
    specialRing, tone : -> Cadence
    _ eq _, _ ne _ : Cadence, Cadence -> Bool
eqns
    forall c1, c2 : Cadence
    ofsort Bool
      specialRing eq specialRing = true;
      specialRing eq tone        = false;
      tone eq specialRing        = false;
      tone eq tone               = true;
      c1 ne c2                   = not(c1 eq c2);
endtype (* Cadence *)

(*********************************************)

(* The PIN is either validPIN or invalidPIN *)
type PIN is Cadence renamedby
sortnames PIN for Cadence
opnnames
    validPIN for tone
    invalidPIN for specialRing
endtype (* PIN *)

(*********************************************)

(* The Message type is mainly for announcements *)
type Message is NaturalNumber
sorts Message
opns
    AskForPIN, displayMessage,
    collectedDigits, ScreenedMessage : -> Message
    map : Message -> Nat
    _ eq _, _ ne _ : Message, Message -> Bool
eqns
    forall m1, m2 : Message
    ofsort Nat
      map(AskForPIN) = 0;
      map(displayMessage)  = succ(0);
      map(collectedDigits) = succ(succ(0));
      map(ScreenedMessage) = succ(succ(succ(0)));
      (* Add new messages when necessary *)
    ofsort Bool
      m1 eq m2 = map(m1) eq map(m2);
      m1 ne m2 = not(m1 eq m2);
endtype (* Message *)

(*********************************************)

(* The TriggerName sort is an enumeration of *)
(* the names of IN triggers.                 *)
type TriggerName is NaturalNumber
sorts TriggerName
opns
    ORIGINATION_ATTEMPT, INFO_COLLECTED, INFO_ANALYZED,
      NETWORK_BUSY : -> TriggerName
    map : TriggerName -> Nat
    _ eq _, _ ne _ : TriggerName, TriggerName -> Bool
eqns
    forall m1, m2 : TriggerName
    ofsort Nat
      map(ORIGINATION_ATTEMPT) = 0;
      map(INFO_COLLECTED)      = succ(0);
      map(INFO_ANALYZED)       = succ(succ(0));
      map(NETWORK_BUSY)        = succ(succ(succ(0)));
```

```
    ofsort Bool
      m1 eq m2 = map(m1) eq map(m2);
      m1 ne m2 = not(m1 eq m2);
endtype (* TriggerName *)

(*********************************************)

(* The ResponseType sort is an enumeration of*)
(* the SCP responses to trigger messages.    *)
type ResponseType is NaturalNumber
sorts ResponseType
opns
    ANALYZE_ROUTE, CONTINUE, FORWARD_CALL, SEND_TO_RESOURCE,
      DISCONNECT : -> ResponseType
    map : ResponseType -> Nat
    _ eq _, _ ne _ : ResponseType, ResponseType -> Bool
eqns
    forall m1, m2 : ResponseType
    ofsort Nat
      map(ANALYZE_ROUTE)    = 0;
      map(CONTINUE)         = succ(0);
      map(FORWARD_CALL)     = succ(succ(0));
      map(SEND_TO_RESOURCE) = succ(succ(succ(0)));
      map(DISCONNECT)       = succ(succ(succ(succ(0))));
    ofsort Bool
      m1 eq m2 = map(m1) eq map(m2);
      m1 ne m2 = not(m1 eq m2);
endtype (* ResponseType *)

(*********************************************)

(* The type of log in the OS is Begin, End, AirBegin, or AirEnd *)
type LogType is NaturalNumber
sorts LogType
opns
    Begin, End, AirBegin, AirEnd : -> LogType
    map : LogType -> Nat
    _ eq _, _ ne _ : LogType, LogType -> Bool
eqns
    forall m1, m2 : LogType
    ofsort Nat
      map(Begin)    = 0;
      map(End)      = succ(0);
      map(AirBegin) = succ(succ(0));
      map(AirEnd)   = succ(succ(succ(0)));
    ofsort Bool
      m1 eq m2 = map(m1) eq map(m2);
      m1 ne m2 = not(m1 eq m2);
endtype (* LogType *)

(* A record for the Log.                                             *)
(* Can be l(Begin,X,Y,P,T) or l(End,X,Y,undefined, T) for regular logs *)
(* and l(AirBegin,X,undefined,undefined,T) or                        *)
(* l(AirEnd,X,undefined,undefined, T) for cellular logs.             *)
type LogRecord is Address, Time, LogType
sorts LogRecord
opns
    l : LogType, Address, Address, Address, Time -> LogRecord
    _ eq _, _ ne _ : LogRecord, LogRecord   -> Bool
eqns
    forall X1, X2, Y1, Y2, P1, P2 : Address,
           T1, T2                 : Time,
           LT1, LT2               : LogType
    ofsort Bool
      (LT1 eq LT2) and (X1 eq X2) and (Y1 eq Y2) and (P1 eq P2) and (T1 eq T2) =>
        l(LT1,X1,Y1,P1,T1) eq l(LT2,X2,Y2,P2,T2)  = true;
      not((LT1 eq LT2) and (X1 eq X2) and (Y1 eq Y2) and (P1 eq P2) and (T1 eq T2)) =>
        l(LT1,X1,Y1,P1,T1) eq l(LT2,X2,Y2,P2,T2)  = false;
      l(LT1,X1,Y1,P1,T1) ne l(LT2,X2,Y2,P2,T2) = not(l(LT1,X1,Y1,P1,T1) eq l(LT2,X2,Y2,P2,T2));
endtype (* LogRecord *)
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
(* List of log records (logs), implemented as a set.          *)
(* We avoid the problem with ISLA's renaming in actualization *)
type Log0 is Set
actualizedby LogRecord using
sortnames
    LogRecord for Element
    Bool  for FBool
endtype (* Logs0 *)

type Log is Log0 renamedby
sortnames
    Log for Set
opnnames
    NoLog for {} (* Empty list of log records *)
endtype (* Log *)

(*********************************************)

(* The Feature sort is an enumeration of the *)
(* features to which users can subscribe,    *)
(* including POTS.                           *)
type Feature is NaturalNumber
sorts Feature
opns
    POTS,  (* Plain Old Telephone System    *)
    CFBL,  (* Call Forward Busy Line        *)
    CND,   (* Call Name Delivery            *)
    INFB,  (* IN Freephone Billing          *)
    INFR,  (* IN Freephone Routing          *)
    INTL,  (* IN Teen Line                  *)
    TCS,   (* Terminating Call Screening    *)
    3WC,   (* Three-way Calling             *)
    INCF,  (* IN Call Forwarding            *)
    CW,    (* Call Waiting                  *)
    CC,    (* Charge Call                   *)
    (* Phase II features, plus one more. *)
    Cell,  (* Cellular                      *)
    RC,    (* Return Call                   *)
    ACB    (* Automatic Call Back (Dorin's) *) : -> Feature

    map : Feature -> Nat
    _ eq _, _ ne _ : Feature, Feature -> Bool
eqns
    forall m1, m2 : Feature
    ofsort Nat
      map(POTS) = 0;
      map(CFBL) = succ(0);
      map(CND)  = succ(succ(0));
      map(INFR) = succ(succ(succ(0)));
      map(INFB) = succ(succ(succ(succ(0))));
      map(INTL) = succ(succ(succ(succ(succ(0)))));
      map(TCS)  = succ(succ(succ(succ(succ(succ(0))))));
      map(3WC)  = succ(succ(succ(succ(succ(succ(succ(0)))))));
      map(INCF) = succ(succ(succ(succ(succ(succ(succ(succ(0))))))));
      map(CW)   = succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))));
      map(CC)   = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))));
      map(Cell) = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))));
      map(RC)   = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))))));
      map(ACB)  = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))))));
```

```
    ofsort Bool
      m1 eq m2 = map(m1) eq map(m2);
      m1 ne m2 = not(m1 eq m2);
endtype (* Feature *)

(* List of features, implemented as a set.                  *)
(* We avoid the problem with ISLA's renaming in actualization *)
type Flist0 is Set
actualizedby Feature using
sortnames
    Feature for Element
    Bool  for FBool
endtype (* Logs0 *)

type Flist is Flist0 renamedby
sortnames
    Flist for Set
opnnames
    NoFList for {} (* Empty list of features *)
endtype (* Flist *)

(*********************************************)

(* A record for the subscriber information.              *)
(* Format: sub(userID, Features, BLForward, LastIncoming, *)
(*              Screened, ChargePin)                      *)
type SInfo is AddList, FList, PIN
sorts SInfo
opns
    sub : Address,     (* User identifier             *)
          FList,       (* List of subscribed features *)
          Address,     (* BLForward, for CFBL         *)
          Address,     (* LastIncoming, for CND       *)
          AddList,     (* Screened list, for TCS      *)
          PIN          (* Charge PIN, for CC          *) -> SInfo
    _ eq _, _ ne _ : SInfo, SInfo   -> Bool
eqns
    forall s1, s2, bl1, bl2, li1, li2: Address,
           fl1, fl2: Flist,
           sl1, sl2: AddList,
           p1, p2: PIN
    ofsort Bool
      (s1 eq s2) and (fl1 eq fl2) and (bl1 eq bl2) and (li1 eq li2)
        and (sl1 eq sl2) and (p1 eq p2) =>
        sub(s1, fl1, bl1, li1, sl1, p1) eq sub(s2, fl2, bl2, li2, sl2, p2) = true;
      not((s1 eq s2) and (fl1 eq fl2) and (bl1 eq bl2) and (li1 eq li2)
        and (sl1 eq sl2) and (p1 eq p2)) =>
        sub(s1, fl1, bl1, li1, sl1, p1) eq sub(s2, fl2, bl2, li2, sl2, p2) = false;
      sub(s1, fl1, bl1, li1, sl1, p1) ne sub(s2, fl2, bl2, li2, sl2, p2) =
        not(sub(s1, fl1, bl1, li1, sl1, p1) eq sub(s2, fl2, bl2, li2, sl2, p2));
endtype (* SInfo *)

(* Database of subscriber records (SInfo), implemented as a set. *)
(* We avoid the problem with ISLA's renaming in actualization.   *)
type SDB0 is Set
actualizedby SInfo using
sortnames
    SInfo for Element
    Bool  for FBool
endtype (* SDB0 *)

type SDB1 is SDB0 renamedby
sortnames
    SDB for Set
opnnames
```



```
    NoSDB for {} (* Empty list of subscribers *)
endtype (* SDB1 *)

(* Query operators *)
type SDB is SDB1
opns
    (* Tells whether a subscriber has subscribed a particular feature *)
    has : Address, Feature, SDB -> Bool
    (* Sets/Gets the$LastIncoming caller *)
    setLastIncoming : Address, Address, SDB -> SDB
    getLastIncoming : Address, SDB -> Address
    (* Check whether the caller party is on the callee's TCS *)
    isOnTCS : Address, Address, SDB -> Bool (* Caller, Callee *)

eqns
    forall s1, s2, s3, bl1, li1, li2: Address,
           sl1: AddList,
           p1: PIN,
           f1, f2: Feature,
           fl    : FList,
           sdb   : SDB
    ofsort Bool
      has(s1, f1, NoSDB) = false;
      s1 eq s2 =>
        has(s1, f1, Insert(sub(s2,fl,bl1,li1,sl1,p1), sdb)) = f1 IsIn fl;
      s1 ne s2 =>
        has(s1, f1, Insert(sub(s2,fl,bl1,li1,sl1,p1), sdb)) = has(s1, f1, sdb);

      isOnTCS(s1, s2, NoSDB) = false;
      s3 eq s2 =>
        isOnTCS(s1, s2, Insert(sub(s3,fl,bl1,li1,sl1,p1), sdb)) = s1 IsIn sl1;
      s3 ne s2 =>
        isOnTCS(s1, s2, Insert(sub(s3,fl,bl1,li1,sl1,p1), sdb)) = isOnTCS(s1, s2, sdb);

    ofsort SDB
      setLastIncoming(s1, li1, NoSDB) = NoSDB;
      s1 eq s2 =>
        setLastIncoming(s1, li1, Insert(sub(s2,fl,bl1,li2,sl1,p1), sdb)) =
                                              Insert(sub(s2,fl,bl1,li1,sl1,p1), sdb);
      s1 ne s2 =>
        setLastIncoming(s1, li1, Insert(sub(s2,fl,bl1,li2,sl1,p1), sdb)) =
          Insert(sub(s2,fl,bl1,li2,sl1,p1), setLastIncoming(s1, li1,sdb));
    ofsort Address
      getLastIncoming(s1, NoSDB) = undefined;
      s1 eq s2  =>
        getLastIncoming(s1, Insert(sub(s2,fl,bl1,li1,sl1,p1), sdb)) = li1;
      s1 ne s2 =>
        getLastIncoming(s1, Insert(sub(s2,fl,bl1,li1,sl1,p1), sdb)) = getLastIncoming(s1, sdb);
endtype (* SDB *)

(*********************************************)

(* The SCPit sort is an enumeration of the   *)
(* SCP types of information in the database. *)
type SCPit is NaturalNumber
sorts SCPit
opns
    Redirect, TeenPIN, TeenTime, ForwardedTo : -> SCPit
    map : SCPit -> Nat
    _ eq _, _ ne _ : SCPit, SCPit -> Bool
eqns
    forall s1, s2 : SCPit
    ofsort Nat
      map(Redirect)    = 0;
      map(TeenPIN)     = succ(0);
```

```
      map(TeenTime)    = succ(succ(0));
      map(ForwardedTo) = succ(succ(succ(0)));
    ofsort Bool
      s1 eq s2 = map(s1) eq map(s2);
      s1 ne s2 = not(s1 eq s2);
endtype (* SCPit *)

(* Information records about the feature parameters in the SCP   *)
(* These heterogeneous records share the same format to simplify *)
(* the equations.                                                *)
type SCPinfo is SCPit, Address, Time, PIN
sorts SCPinfo
opns
    (* INFR: Redirect A B T1 T2 C                                               *)
    (*       -> scp(Redirect, A, B, T1, T2, C, validPIN)                        *)
    (* INTL: TeenPIN A PIN                                                      *)
    (*       -> scp(TeenPIN, A, undefined, initTime, initTime, undefined, PIN)  *)
    (* INTL: TeenTime A T1 T2                                                   *)
    (*       -> scp(TeenTime, A, undefined, T1, T2, undefined, validPIN)        *)
    (* INCF: ForwardedTo B C                                                    *)
    (*       -> scp(ForwardedTo, undefined, B, initTime, initTime, C, validPIN) *)
    scp : SCPit, Address, Address, Time, Time, Address, PIN -> SCPinfo
    _ eq _, _ ne _ : SCPinfo, SCPinfo -> Bool
eqns
    forall s1, s2               : SCPit,
           a1, a2, b1, b2, c1, c2: Address,
           t11, t12, t21, t22    : Time,
           pin1, pin2            : PIN
    ofsort bool
    (s1 eq s2) and (a1 eq a2) and (b1 eq b2) and (c1 eq c2) and (t11 eq t12) and (t21 eq t22) and
                                                   (pin1 eq pin2) =>
      scp(s1, a1, b1, t11, t21, c1, pin1) eq scp(s2, a2, b2, t12, t22, c2, pin2) = true;
    not((s1 eq s2) and (a1 eq a2) and (b1 eq b2) and (c1 eq c2) and (t11 eq t12) and (t21 eq t22)
                                                   and (pin1 eq pin2)) =>
      scp(s1, a1, b1, t11, t21, c1, pin1) eq scp(s2, a2, b2, t12, t22, c2, pin2) = false;
    scp(s1, a1, b1, t11, t21, c1, pin1) ne scp(s2, a2, b2, t12, t22, c2, pin2) =
          not(scp(s1, a1, b1, t11, t21, c1, pin1) eq scp(s2, a2, b2, t12, t22, c2, pin2))
endtype (* SCPinfo *)

(* Database of feature parameters (SCPinfo) in the SCP, implemented as a set. *)
(* We avoid the problem with ISLA's renaming in actualization.   *)
type SCPDB0 is Set
actualizedby SCPinfo using
sortnames
    SCPinfo for Element
    Bool  for FBool
endtype (* SCPDB0 *)

type SCPDB1 is SCPDB0 renamedby
sortnames
    SCPDB for Set
opnnames
    NoSCPDB for {} (* Empty list of feature parameters. *)
endtype (* SCPDB1 *)

(* Query operators *)
type SCPDB is SCPDB1
opns
    (* Tells whether this is an INTL restricted time or not *)
    IsInTeenTime : Address, Time, SCPDB -> Bool
    IsValidTeenPIN : Address, PIN, SCPDB -> Bool
eqns
    forall scpit        : SCPit,
           a1, a2, b, c : Address,
           t, t1, t2    : Time,
```



```
           p, p1, p2    : PIN,
           scpdb        : SCPDB
    ofsort Bool
    (* IsInTeenTime *)
      IsInTeenTime(a1, t, NoSCPDB) = false;
      (scpit eq TeenTime) and (a1 eq a2) and (t ge t1) and (t lt t2) =>
        IsInTeenTime(a1, t, Insert(scp(scpit, a2, b, t1, t2, c, p), scpdb)) = true;
      not((scpit eq TeenTime) and (a1 eq a2) and (t ge t1) and (t lt t2)) =>
        IsInTeenTime(a1, t, Insert(scp(scpit, a2, b, t1, t2, c, p), scpdb)) =
            IsInTeenTime(a1, t, scpdb);

    (* IsValidTeenPIN *)
      IsValidTeenPIN(a1, p1, NoSCPDB) = false;
      (scpit eq TeenPIN) and (a1 eq a2) and (p1 eq p2) =>
        IsValidTeenPIN(a1, p1, Insert(scp(scpit, a2, b, t1, t2, c, p2), scpdb)) = true;
      not((scpit eq TeenPIN) and (a1 eq a2) and (p1 eq p2)) =>
        IsValidTeenPIN(a1, p1, Insert(scp(scpit, a2, b, t1, t2, c, p2), scpdb)) =
            IsValidTeenPIN(a1, p1, scpdb);
endtype (* SCPDB *)

(*********************************************)

(* The StatItem sort is an enumeration of the  *)
(* status items in the switch in the database. *)
type StatItem is NaturalNumber
sorts StatItem
opns
    Busy, Ringing, AudibleRinging, ThreeWay, CallWaiting  : -> StatItem
    map : StatItem -> Nat
    _ eq _, _ ne _ : StatItem, StatItem -> Bool
eqns
    forall s1, s2 : StatItem
    ofsort Nat
      map(Busy)           = 0;
      map(Ringing)        = succ(0);
      map(AudibleRinging) = succ(succ(0));
      map(ThreeWay)       = succ(succ(succ(0)));
      map(CallWaiting)    = succ(succ(succ(succ(0))));
    ofsort Bool
      s1 eq s2 = map(s1) eq map(s2);
      s1 ne s2 = not(s1 eq s2);
endtype (* StatItem *)

(* Status records collected in the switch during calls *)
type Stat is Address, StatItem
sorts Stat
opns
    (* POTS: Busy A             -> stat(Busy, A, undefined)        *)
    (* POTS: Ringing A B        -> stat(Rigning, A, B)             *)
    (* POTS: AudibleRinging A B -> stat(AudibleRinging, A, B)      *)
    (* 3WC : ThreeWay X         -> stat(ThreeWay, X, undefined)    *)
    (* CW  : CallWaiting X      -> stat(CallWaiting, X, undefined) *)
    stat : StatItem, Address, Address -> Stat
    _ eq _, _ ne _ : Stat, Stat -> Bool
eqns
    forall a1, a2, b1, b2: Address,
           si1, si2: StatItem
    ofsort Bool
      (a1 eq a2) and (b1 eq b2) and (si1 eq si2) =>
        stat(si1, a1, b1) eq stat(si2, a2, b2) = true;
      not((a1 eq a2) and (b1 eq b2) and (si1 eq si2)) =>
        stat(si1, a1, b1) eq stat(si2, a2, b2) = false;
      stat(si1, a1, b1) ne stat(si2, a2, b2) =
        not(stat(si1, a1, b1) eq stat(si2, a2, b2));
endtype (* Stat *)
```

```
(* Database of status records in the switch, implemented as a set. *)
(* We avoid the problem with ISLA's renaming in actualization.   *)
type Status0 is Set
actualizedby Stat using
sortnames
    Stat for Element
    Bool  for FBool
endtype (* Status0 *)

type Status1 is Status0 renamedby
sortnames
    Status for Set
opnnames
    NoStatus for {} (* Empty list of status. *)
endtype (* Status1 *)

(* Query operators *)
type Status is Status1
opns
    (* Tells whether a subscriber is Idle or Busy *)
    isIdle, isBusy : Address,  Status -> Bool
eqns
    forall a1, a2, b1, b2 : Address,
           si1, si2       : StatItem,
           s              : Status
    ofsort Bool
    (* isIdle *)
      isIdle(a1, NoStatus) = true;
      (a1 eq a2) and (si2 eq Busy) =>
        isIdle(a1, Insert(stat(si2, a2, b2), s)) = false;
      not((a1 eq a2) and (si2 eq Busy)) =>
        isIdle(a1, Insert(stat(si2, a2, b2), s)) = isIdle(a1, s);
    (* isBusy *)
      isBusy(a1, s) = not(isIdle(a1, s));
endtype (* Status *)

(*============================================*)
(*         Stub Path ADT definitions          *)
(*============================================*)

(* Entry and exit points of each stub in the maps *)
type StubPath is NaturalNumber
sorts StubPath
opns
    inPreD1, outPreD1, outPreD2,      (* pre-dial stub     *)
    inPostD1, outPostD1, outPostD2, outPostD3, outPostD4,
      outPostD5,                      (* post-dial stub    *)
    inBill1, outBill2,                (* billing stub      *)
    inPC1, outPC1, outPC2, outPC3,
      outPC4,                         (* process-call stub *)
    inDisp1, outDisp1,                (* display stub      *)
    inBusy1, outBusy1, outBusy2       (* busy stub         *) : -> StubPath
    map : StubPath -> Nat
    _ eq _, _ ne _ : StubPath, StubPath -> Bool
eqns
    forall sp1, sp2 : StubPath
    ofsort Nat
      map(inPreD1)   (* From OffHook      *) = 0;
      map(outPreD1)  (* To Dial           *) = succ(map(inPreD1));
      map(outPreD2)  (* To Reject         *) = succ(map(outPreD1));
      map(inPostD1)  (* From Dial         *) = succ(map(outPreD2));
      map(outPostD1) (* To Term-Connected *) = succ(map(inPostD1));
      map(outPostD2) (* To Orig-Connected *) = succ(map(outPostD1));
      map(outPostD3) (* To Billing        *) = succ(map(outPostD2));
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
      map(outPostD4) (* To Reject         *) = succ(map(outPostD3));
      map(outPostD5) (* To Busy           *) = succ(map(outPostD4));
      map(inBill1)   (* From Post-Dial    *) = succ(map(outPostD5));
      map(outBill2)  (* To Result-OS      *) = succ(map(inBill1));
      map(inPC1)     (* From Call         *) = succ(map(outBill2));
      map(outPC1)    (* To Ring (Term)    *) = succ(map(inPC1));
      map(outPC2)    (* To Busy           *) = succ(map(outPC1));
      map(outPC3)    (* To Reject         *) = succ(map(outPC2));
      map(outPC4)    (* To stub itself    *) = succ(map(outPC3));
      map(inDisp1)   (* From PC stub      *) = succ(map(outPC4));
      map(outDisp1)  (* To OffHook        *) = succ(map(inDisp1));
      map(inBusy1)   (* From Process-Call *) = succ(map(outDisp1));
      map(outBusy1)  (* To Busy           *) = succ(map(inBusy1));
      map(outBusy2)  (* To Call_X         *) = succ(map(outBusy1));
    ofsort Bool
      sp1 eq sp2 = map(sp1) eq map(sp2);
      sp1 ne sp2 = not(sp1 eq sp2);
endtype (* StubPath *)

type SPList0 is Set
actualizedby StubPath using
sortnames
    StubPath for Element
    Bool  for FBool
endtype (* SPList0 *)

type SPList is SPList0 renamedby
sortnames
    SPList for Set
opnnames
    NoSPList for {} (* Empty list of path identifiers. *)
endtype (* SPList *)

(*============================================*)
(*          Behaviour Description             *)
(*============================================*)

behaviour

  (* Gates not visible to the users are set to be internal. *)
  (* Interfaces (e.g. Switch2User) are splitted into several*)
  (*   gates, one per type of message.                      *)
  hide Trigger,  (* Switch2SCP *)
       Resource, (* Switch2SCP *)
       Response, (* SCP2Switch *)
       LogBegin, (* 2OS        *)
       LogEnd,   (* 2OS        *)
       AirBegin, (* 2OS        *)
       AirEnd,   (* 2OS        *)
       Time      (* NEW: Used by the Switch to get the time *)
  in

  (* Get the Initial state from the environment *)
  Init ?InitSDB:SDB ?InitStatus:Status ?InitSCPDB:SCPDB ?currentTime:Time;
  (
    (* We create as many users as necessary.  *)
    UserFactory [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                 Disconnect, Display, CreateUser]
    |[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
      StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
      Disconnect, Display]|
    (
      (
        GlobalClock [Time](currentTime)
```

```
        |[Time]|
        Switch [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Trigger, Resource, Response, LogBegin,
                LogEnd, AirBegin, AirEnd, Time](InitSDB, InitStatus)
        |[Trigger, Resource, Response]|
        SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd](InitSCPDB)
      )
      |[LogBegin, LogEnd, AirBegin, AirEnd]|
      OS [LogBegin, LogEnd, AirBegin, AirEnd, Query](NoLog)
    )
  )

where

  (*******************************************************************)
  (* Process UserFactory: To create ans initialise necessary users .  *)
  (*******************************************************************)

  process UserFactory [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                       StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                       Disconnect, Display, CreateUser]: noexit :=
    CreateUser ?userId:Address ?userFeatures:FList;
    (
      (* Create the user *)
      User [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display] (userId, userFeatures)
      |||
      (* Prepare to accept new creation request *)
      UserFactory [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                   StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                   Disconnect, Display, CreateUser]
    )
  endproc (* UserFactory *)


  (*******************************************************************)
  (* Process User: To be instantiated by all users with a userId.     *)
  (*******************************************************************)

  process User [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display] (userId: Address, uf:FList): noexit :=

    (* POTS - Origination (Root map) *)
    OffHook !userId; (*_PROBE_*)
      User [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display] (userId, uf)
    []

    DialTone !userId;
      (
        Dial !userId ?userTo:Address;
          User [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display] (userId, uf)
        []
        OnHook !userId; (*_PROBE_*) stop
      )

    []

    LineBusyTone !userId;
      OnHook !userId; (*_PROBE_*) stop

    []

    (* POTS - Origination (post-dial default map) *)
    StartAR !userId ?userTo:Address;
    (
      OnHook !userId;
        StopAR !userId !userTo; (*_PROBE_*) stop
      []
      StopAR !userId !userTo;
        (* CONNECTED state! Use Disconnect map. *)
        (
          OnHook !userId; (*_PROBE_*) stop
          []
          Disconnect !userId !userTo;
            OnHook !userId; (*_PROBE_*) stop
        )
    )

    []

    (* POTS - Termination (post-dial default map) *)

    (
      (
        StartR !userId ?userFrom:Address; exit(userId, userFrom, uf, any SPList)
        |||
        DisplayStub[Display](userId, uf, Insert(inDisp1, NoSPList))
      )
      >>
      accept userId:Address, userFrom:Address, uf:FList, outPaths:SPList in
        (* outDisp1 is the only possible outPath... *)
        [outDisp1 IsIn outPaths] ->
          (
            OffHook !userId;
              StopR !userId !userFrom;
              (* CONNECTED state! Use Disconnect map) *)
                (
                  Disconnect !userId !UserFrom;
                    OnHook !userId; (*_PROBE_*) stop
                  []
                  OnHook !userId; (*_PROBE_*) stop
                )
            []
            (* userFrom has gon on-hook. *)
            StopR !userId !userFrom; (*_PROBE_*) stop
          )
    )

    []

    (* INTL - Origination (pre-dial INTL map) *)
    Announce !userId !AskForPIN;
      (
        Dial !userId ?p:PIN;  (*_PROBE_*)
          User [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display] (userId, uf)
        []
        OnHook !userId; (*_PROBE_*) stop
      )

    []
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
  Announce !userId !InvalidPIN;
    OnHook !userId; (*_PROBE_*) stop

  []

  (* TCS reject *)
  Announce !userId !ScreenedMessage;
    OnHook !userId; (*_PROBE_*) stop

  (* NOT TO FORGET: Flash may be interpreted as OnHook-OffHook... *)

  where

    (*******************************************************************)
    (* Stub Process DisplayStub: in map Post-Dial. One input path and   *)
    (*                             one output path                      *)
    (*******************************************************************)
    process DisplayStub[Display](userId:Address, uf:FList, inPaths:SPList) :
                                  exit(Address, Address, FList, SPList) :=
      (* In this stub, CND is optional (we do not have access to SCPDB). *)
      [CND NotIn uf] -> (* Can't put a probe here... Check P13=!P6+P7+P8 instead. *)
        (* POTS default plugin *)
        exit(userId, any Address, uf, Insert(outDisp1, NoSPList))
      []
      [CND IsIn uf] ->
        (* CND plugin. UserFrom provided by the switch. *)
        Display !userId ?userFrom:Address; (*_PROBE_*)
          exit(userId, any Address, uf, Insert(outDisp1, NoSPList))
    endproc (* DisplayStub *)

endproc (* User *)


(*******************************************************************)
(* Process Switch:                                                  *)
(*******************************************************************)
process Switch [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Trigger, Resource, Response, LogBegin,
               LogEnd, AirBegin, AirEnd, Time] (sdb:SDB, status:Status)
               : noexit :=

  OffHook ?userFrom:Address [isIdle(userFrom, status)];
    PreDialStub[OnHook, Trigger, Response, Resource, Announce, Dial, Time]
               (Insert(inPreD1, NoSPList), userFrom, sdb,
       (* Set userFrom Busy *)
       Insert(stat(Busy, userFrom, undefined), status))
    >>
    accept userFrom:Address, sdb:SDB, status:Status, outPaths:SPList in
      (* outPreD1: Ok *)
      [outPreD1 IsIn outPaths] ->
        (
          DialTone !userFrom;
            (
              (
                Dial !userFrom ?userTo:Address;
                  PostDialStub[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
                               StartR, StartCWT, StopAR, StopR, StopCWT,
                               LineBusyTone, Announce, Disconnect, Display,
                               Trigger, Resource, Response, LogBegin, LogEnd,
                               Time](Insert(inPostD1, NoSPList), userFrom, userTo, sdb, status)
                >>
                accept userFrom:Address, userTo:Address, userPay:Address, sdb:SDB, status:Status, outPat
                                         hs:SPList in
                  (* Output path 1 comes before 2,3, which can work as alternatives
```



```
                     (for disconnection purposes). *)
  (* Paths 4, 5, and {1,2,3} are alternative paths. *)
  [outPostD1 IsIn outPaths] ->
  (
    (* userFrom may have disconnected... *)
    [isBusy(userFrom, status)] ->
      Time ?t:time;
        LogBegin !userFrom !userTo !userPay !t;
        (
          (* Orig-Connected, and Orig disconnects (POTS 7) *)
          [outPostD2 IsIn outPaths] ->
            (
              OnHook !userFrom;
              (
                let status:Status = Remove(stat(Busy, userFrom, undefined), status) in
                (
                  Disconnect !userTo !userFrom;
                    exit(userFrom, userTo, sdb, status)
                  |||
                  Time ?t:time;
                    LogEnd !userFrom !userTo !t;
                      exit(userFrom, userTo, sdb, status)
                )
                >> accept userFrom:Address, userTo:Address, sdb:SDB, status:Status in
                  OnHook !userTo;
                  (
                    let status:Status = Remove(stat(Busy, userTo, undefined), status) in
                             (*_PROBE_*)
                      stop
                  )
              )
            )
          []
          (* Term-Connected, and Term disconnects (POTS 10) *)
          [outPostD3 IsIn outPaths] ->
            (
              OnHook !userTo;
              (
                let status:Status = Remove(stat(Busy, userTo, undefined), status) in
                (
                  Disconnect !userFrom !userTo;
                    exit(userFrom, userTo, sdb, status)
                  |||
                  Time ?t:time;
                    LogEnd !userFrom !userTo !t;
                      exit(userFrom, userTo, sdb, status)
                )
                >> accept userFrom:Address, userTo:Address, sdb:SDB, status:Status in
                  OnHook !userFrom;
                  (
                    let status:Status = Remove(stat(Busy, userFrom, undefined), status)
                          in (*_PROBE_*)
                      stop
                  )
              )
            )
        )
    []
    [IsIdle(userFrom, status)] ->
      (
        (* UserFrom has gone on-hook. *) (*_PROBE_*)
        stop
      )
  )
  []
```

```
                [outPostD4 IsIn outPaths] ->
                  (
                    (* Reject path. *)
                    Announce !userFrom !ScreenedMessage;
                    OnHook !userFrom;
                    (
                      let status:Status = Remove(stat(Busy, userFrom, undefined), status) in
                                          (*_PROBE_*)
                        stop
                    )
                  )
                []
                [outPostD5 IsIn outPaths] ->
                  (
                    (* Busy. (POTS 15) *)
                    LineBusyTone !userFrom;
                      OnHook !userFrom;
                      (
                        let status:Status = Remove(stat(Busy, userFrom, undefined), status) in
                                          (*_PROBE_*)
                          stop
                      )
                  )
            )
            []
            OnHook !userFrom; (* From POTS 7-12? *)
            (
              (* Set userFrom Idle *)
              let status:Status = Remove(stat(Busy, userFrom, undefined), status) in  (*_PROBE_*)
                stop
            )
          )
        )
        |||
        (* outPreD2: reject *)
        [outPreD2 IsIn outPaths] ->
          (
            OnHook !userFrom; (* From INTL 12 *)
            (
              (* Set userFrom Idle *)
              let status:Status = Remove(stat(Busy, userFrom, undefined), status) in (*_PROBE_*)
                stop
            )
          )

  where

    (********************************************************************)
    (* Stub Process PreDialStub:                                        *)
    (********************************************************************)
    process PreDialStub[OnHook, Trigger, Response, Resource, Announce, Dial, Time]
              (inPaths: SPList, userFrom: Address, sdb: SDB,
               status: Status):exit (Address, SDB, Status, SPList) :=
      (* In this stub, INTL is mutually exclusive with all other features. *)

      (* INTL plugin *)
      [has(userFrom, INTL, sdb)] ->
        (* NEW EVENTS: we believe that the INTL information should be located in the SCP. *)
        (* Is the time in the subscriber's TeenTime interval? *)
        Time ?time:Time; (* Get the current time *)
        Resource !INTL !userFrom !time;
        Response !INTL !userFrom ?inTeenTime:Bool;
        (
          (* Unrestricted time for INTL *)
          [not(inTeenTime)] -> (*_PROBE_*)
```



```
            exit (userFrom, sdb, status, Insert(outPreD1, NoSPList))
          []
          (* Restricted time for INTL *)
          [inTeenTime] ->
            Trigger !ORIGINATION_ATTEMPT !userFrom !userFrom !undefined !time ;
              Response !SEND_TO_RESOURCE !userFrom ?m:message;
                Announce !userFrom !m;
                  (
                    OnHook !UserFrom; (*_PROBE_*) stop (* INTL 13 *)
                    []
                    Dial !userFrom ?pin:PIN;
                      Resource !userFrom !pin;
                        (
                          Response !CONTINUE !userFrom !userFrom !undefined; (*_PROBE_*)
                            exit (userFrom, sdb, status, Insert(outPreD1, NoSPList))
                          []
                          Response !SEND_TO_RESOURCE !userFrom !invalidPIN;
                            Resource !userFrom !undefined;
                              Announce !userFrom !invalidPIN;
                                Response !DISCONNECT !userFrom !undefined; (*_PROBE_*)
                                  exit (userFrom, sdb, status, Insert(outPreD2, NoSPList))
                        )
                  )
        )

      []

      (* Default plugin *)
      [not(has(userFrom, INTL, sdb))] -> (*_PROBE_*)
          exit (userFrom, sdb, status, Insert(outPreD1, NoSPList))
    endproc (* PreDialStub *)


    (********************************************************************)
    (* Stub Process PostDialStub:                                       *)
    (********************************************************************)
    process PostDialStub[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
                         StartR, StartCWT, StopAR, StopR, StopCWT,
                         LineBusyTone, Announce, Disconnect, Display,
                         Trigger, Resource, Response, LogBegin, LogEnd, Time]
              (inPaths: SPList, userFrom: Address, userTo:Address, sdb: SDB,
               status: Status):exit (Address, Address, Address, SDB, Status, SPList) :=

      (* Use the processCallStub first *)
      ProcessCallStub[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
                  StartR, StartCWT, StopAR, StopR, StopCWT,
                  LineBusyTone, Announce, Disconnect, Display,
                  Trigger, Resource, Response, LogBegin, LogEnd, Time]
                      (Insert(inPC1, NoSPList), userFrom, userTo, sdb, status)
      >>
      accept userFrom:Address, userTo:Address, userPay:Address, sdb:SDB, status:Status, outPaths:SPList
                                             in
      (* All choices are mutually exclusive here. *)
        (* outPC1: Idle *)
        [outPC1 IsIn outPaths] ->
          (
            (* Set userTo Busy *)
            let status:Status = Insert(stat(Busy, userTo, undefined), status) in
            (
              StartAR !userFrom !userTo; (*_PROBE_*)
                exit (userFrom, userTo, userPay, any SDB, Insert(stat(AudibleRinging, userFrom, userTo),
                                                  Insert(stat(Ringing, userTo, userFrom), status)))
              |||
              StartR !userTo !userFrom;  (*_PROBE_*)
                exit (userFrom, userTo, userPay, any SDB, Insert(stat(AudibleRinging, userFrom, userTo),
```

```
                                                      Insert(stat(Ringing, userTo, userFrom), status)))
                |||
                (* For SetDisplayStub and CND plugin for ProcessCallStub *)
                (
                  [has(userTo, CND, sdb)] ->
                    (* Update LastIncoming and Display it. *)
                    Display !userTo !userFrom; (*_PROBE_*)
                      exit (userFrom, userTo, userPay, setLastIncoming(userTo, userFrom, sdb), any Status)
                  []
                  [not(has(userTo, CND, sdb))] ->
                    (* Do nothing special *) (*_PROBE_*)
                    exit (userFrom, userTo, userPay, sdb, any Status)
                )
              )
              >>
              accept userFrom:Address, userTo:Address, userPay:Address, sdb:SDB, status:Status in
                OffHook !userTo; (* POTS 5 *)
                  (
                    StopAR !userFrom !userTo; (*_PROBE_*)
                      exit ( userFrom, userTo, userPay, sdb,
                             Remove(stat(AudibleRinging, userFrom, userTo),
                               Remove(stat(Ringing, userTo, userFrom), status)),
                             Insert(outPostD1, Insert(outPostD2, Insert(outPostD3, NoSPList))) )
                    |||
                    StopR !userTo !userFrom; (*_PROBE_*)
                      exit ( userFrom, userTo, userPay, sdb,
                             Remove(stat(AudibleRinging, userFrom, userTo),
                               Remove(stat(Ringing, userTo, userFrom), status)),
                             Insert(outPostD1, Insert(outPostD2, Insert(outPostD3, NoSPList))) )
                  )
                []
                (* Disconnection possible here... *)
                OnHook !userFrom; (* POTS 5 *)
                  (
                    (* Set userFrom Idle *)
                    let status:Status = Remove(stat(Busy, userFrom, undefined), status) in
                      (
                        StopAR !userFrom !userTo; (* Set userTo idle after the synchronization *)
                                                                (*_PROBE_*)
                          exit (userFrom, userTo, userPay, sdb,
                                Remove(stat(Busy, userTo, undefined),
                                  Remove(stat(AudibleRinging, userFrom, userTo),
                                    Remove(stat(Ringing, userTo, userFrom), status))),
                                Insert(outPostD1, NoSPList))
                        |||
                        StopR !userTo !userFrom; (*_PROBE_*)
                          exit (userFrom, userTo, userPay, sdb,
                                Remove(stat(Busy, userTo, undefined),
                                  Remove(stat(AudibleRinging, userFrom, userTo),
                                    Remove(stat(Ringing, userTo, userFrom), status))),
                                Insert(outPostD1, NoSPList))
                      )
                  )
            )
          []
          (* outPC2: Busy *)
          [outPC2 IsIn outPaths] ->
            (
              (* Invoke BusyStub *)
              BusyStub[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
                       StartR, StartCWT, StopAR, StopR, StopCWT,
                       LineBusyTone, Announce, Disconnect, Display,
                       Trigger, Resource, Response, LogBegin, LogEnd, Time]
                          (Insert(inBusy1, NoSPList), userFrom, userTo, userPay, sdb, status)
              >>
              accept userFrom:Address, userTo:Address, userPay:Address, sdb:SDB, status:Status,
                                                      outPaths:SPList in
                [outBusy1 IsIn outPaths] ->
                  (* CC: invalid PIN *) (*_PROBE_*)
                  exit (userFrom, userTo, userPay, sdb, status, Insert(outPostD5, NoSPList))
                []
                [outBusy2 IsIn outPaths] ->
                  (* TO BE DONE *) (*_PROBE_*)
                  stop
            )
          []
          (* outPC3: Reject *)
          [outPC3 IsIn outPaths] ->
            (
              (* TCS Reject *) (*_PROBE_*)
              exit (userFrom, userTo, userPay, sdb, status, Insert(outPostD4, NoSPList))
            )
          []
          (* outPC4: Back to stub *)
          [outPC4 IsIn outPaths] ->
          (
            (*_PROBE_*) stop (* TO BE DONE *)
          )

      where

        (***********************************************************)
        (* Stub Process ProcessCallStub:                           *)
        (***********************************************************)
        process ProcessCallStub [OffHook, OnHook, Dial, Flash, DialTone,
                  StartAR, StartR, StartCWT, StopAR, StopR, StopCWT,
                  LineBusyTone, Announce, Disconnect, Display,
                  Trigger, Resource, Response, LogBegin, LogEnd, Time]
                      (inPaths: SPList, userFrom: Address, userTo:Address,
                       sdb: SDB, status: Status)
                  : exit (Address, Address, Address, SDB, Status, SPList) :=

          (* CND will be taken care of at outPC1, after all these plug-ins. *)

          (* TCS (reject path) has priority over the other features. *)
          [has(userTo, TCS, sdb) and isOnTCS(userFrom, userTo, SDB)] -> (*_PROBE_*)
              (* Caller on the list. Reject call. *)
              exit (userFrom, userTo, userFrom, sdb, status, Insert(outPC3, NoSPList))
          []
          (* Remaining features *)
          [not(has(userTo, TCS, sdb) and isOnTCS(userFrom, userTo, SDB))] ->
          (
            (* INFB *)
            [has(userTo, INFB, sdb)] -> (*_PROBE_*)
              PluginINFB[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
                         StartR, StartCWT, StopAR, StopR, StopCWT,
                         LineBusyTone, Announce, Disconnect, Display,
                         Trigger, Resource, Response, LogBegin, LogEnd, Time]
                           (inPaths, userFrom, userTo, sdb, status)
            []
            (* Default *)
            [not(has(userTo, INFB, sdb))] -> (*_PROBE_*)
              PluginDefault[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
                            StartR, StartCWT, StopAR, StopR, StopCWT,
                            LineBusyTone, Announce, Disconnect, Display,
                            Trigger, Resource, Response, LogBegin, LogEnd, Time]
                              (inPaths, userFrom, userTo, sdb, status)
          )
        where

```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
        process PluginINFB[OffHook, OnHook, Dial, Flash, DialTone,
                StartAR, StartR, StartCWT, StopAR, StopR, StopCWT,
                LineBusyTone, Announce, Disconnect, Display,
                Trigger, Resource, Response, LogBegin, LogEnd, Time]
                    (inPaths: SPList, userFrom: Address, userTo:Address,
                     sdb: SDB, status: Status)
                : exit (Address, Address, Address, SDB, Status, SPList) :=

          (* INFB plugin for ProcessCallStub *)
          Time ?t:Time;
            Trigger !INFO_ANALYZED !userTo !userFrom !userTo !t;
              Response !ANALYZE_ROUTE !userTo !userFrom !userTo !userTo;
              (
                [IsIdle(userTo, status)] -> (*_PROBE_*)
                  (* Called party (userTo) pays. *)
                  exit (userFrom, userTo, userTo, sdb, status, Insert(outPC1, NoSPList))
                []
                [IsBusy(userTo, status)] -> (*_PROBE_*)
                  exit (userFrom, userTo, userTo, sdb, status, Insert(outPC2, NoSPList))
              )
        endproc (* PluginINFB *)

        process PluginDefault [OffHook, OnHook, Dial, Flash, DialTone,
                StartAR, StartR, StartCWT, StopAR, StopR, StopCWT,
                LineBusyTone, Announce, Disconnect, Display,
                Trigger, Resource, Response, LogBegin, LogEnd, Time]
                    (inPaths: SPList, userFrom: Address, userTo:Address,
                     sdb: SDB, status: Status)
                : exit (Address, Address, Address, SDB, Status, SPList) :=

          (* Default plugin for ProcessCallStub *)
          [IsIdle(userTo, status)] -> (*_PROBE_*)
            exit (userFrom, userTo, userFrom, sdb, status, Insert(outPC1, NoSPList))
          []
          [IsBusy(userTo, status)] -> (*_PROBE_*)
            exit (userFrom, userTo, userFrom, sdb, status, Insert(outPC2, NoSPList))
        endproc (* PluginDefault *)

      endproc (* ProcessCallStub *)

      (************************************************************)
      (* Stub Process BusyStub:                                   *)
      (************************************************************)
      process BusyStub [OffHook, OnHook, Dial, Flash, DialTone,
                StartAR, StartR, StartCWT, StopAR, StopR, StopCWT,
                LineBusyTone, Announce, Disconnect, Display,
                Trigger, Resource, Response, LogBegin, LogEnd, Time]
                    (inPaths: SPList, userFrom: Address, userTo:Address,
                     userPay:Address, sdb: SDB, status: Status)
                : exit (Address, Address, Address, SDB, Status, SPList) :=

        (* POTS default plugin. *)
          exit (userFrom, userTo, userPay, sdb, status, Insert(outBusy1, NoSPList))
          (* No probe here. Obviously covered for now. *)
        (* TO BE DONE: other plugins. *)
      endproc (* BusyStub *)

    endproc (* PostDialStub *)

endproc (* Switch *)


(*******************************************************************)
(* Process SCP:                                                    *)
(*******************************************************************)
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
process SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd]
            (scpbd:SCPDB) : noexit :=

  (* From INTL: check if time is within TeenTime limits. *)
  Resource !INTL ?user:Address ?time:Time;
    Response !INTL !user !IsInTeenTime(user, time, scpbd); (*_PROBE_*)
      SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd](scpbd)
  []
  (* From INTL: Origination attempt to connect. Ask for PIN. *)
  Trigger !ORIGINATION_ATTEMPT ?user:Address ?user2:Address !undefined ?t:Time [user eq user2];
    Response !SEND_TO_RESOURCE !user !AskForPIN; (*_PROBE_*)
      SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd](scpbd)
  []
  (* From INTL: check PIN *)
  Resource ?user:Address ?pin:PIN;
  (
    [IsValidTeenPIN(user, pin, scpbd)] ->
      Response !CONTINUE !user !user !undefined; (*_PROBE_*)
        SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd](scpbd)
    []
    [not(IsValidTeenPIN(user, pin, scpbd))] ->
      Response !SEND_TO_RESOURCE !user !invalidPIN;
        Resource ?user:Address !undefined;
          Response !DISCONNECT !user !undefined; (*_PROBE_*)
            SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd](scpbd)
  )
  []
  (* From INFB: IN Analyze *)
  Trigger !INFO_ANALYZED ?userTo:Address ?userFrom:Address ?userTo2:Address ?t:Time [userTo eq userTo2]
    Response !ANALYZE_ROUTE !userTo !userFrom !userTo !userTo; (*_PROBE_*)
      SCP [Trigger, Resource, Response, LogBegin, LogEnd, AirBegin, AirEnd](scpbd)
endproc (* SCP *)


(*******************************************************************)
(* Process OS:                                                     *)
(*******************************************************************)
process OS [LogBegin, LogEnd, AirBegin, AirEnd, Query](log:Log) : noexit :=
  LogBegin ?From:Address ?To:Address ?Paying:Address ?t:Time; (*_PROBE_*)
    OS[LogBegin, LogEnd, AirBegin, AirEnd, Query]
      (Insert(l(Begin, From, To, Paying, t),log))
  []
  LogEnd ?From:Address ?To:Address ?t:Time;  (*_PROBE_*)
    OS[LogBegin, LogEnd, AirBegin, AirEnd, Query]
      (Insert(l(End, From, To, undefined, t),log))
  []
  (* For Phase II features *)
  AirBegin ?From:Address ?t:Time; (*_PROBE_*)
    OS[LogBegin, LogEnd, AirBegin, AirEnd, Query]
      (Insert(l(AirBegin, From, undefined, undefined, t),log))
  []
  AirEnd ?From:Address ?t:Time; (*_PROBE_*)
    OS[LogBegin, LogEnd, AirBegin, AirEnd, Query]
      (Insert(l(AirEnd, From, undefined, undefined, t),log))
  []
  (* NEW functionality which allow a test case to check the Log. *)
  Query !log; (*_PROBE_*)
    OS[LogBegin, LogEnd, AirBegin, AirEnd, Query](log)
endproc (* OS *)


(*******************************************************************)
(* Process GlobalClock: computes the relative time incrementally and*)
(*                       provides timestamps (t) when required.     *)
(*******************************************************************)
```

```

  process GlobalClock [Time](t:Time) : noexit :=
    Time !T; GlobalClock[Time](tic(t)) (* No probe here. Obviously covered... *)

  endproc (* GlobalClock *)


(*==================================================================*)
(*                                                                  *)
(*                           POTS PROCESSES                         *)
(*                                                                  *)
(*==================================================================*)

(* These six processes represent common test sequences (repreenting   *)
(* a canonical tester) among many features (POTS states 1, 2, 4, 5,   *)
(* 13 and 15). 3WC and CW are not covered entirely by these.          *)
(* They all exit so that we can check the Log afterwards in test cases*)
(* using these processes.                                             *)

process POTS_1 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom:Address, userTo:Address)
                : exit(Nat) :=
    OffHook !userFrom;
      POTS_2 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
              Disconnect, Display, Success] (userFrom, userTo)
endproc (* POTS_1 *)


process POTS_2 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom:Address, userTo:Address)
                : exit(Nat) :=
    DialTone !userFrom; (* State 2 *)
      (
        i; OnHook !userFrom; exit(succ(succ(succ(succ(0)))))  (* State 17 *)
        []
        i; Dial !userFrom !userTo; (* State 3 *)
          (
             POTS_4 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom, userTo)
             []
             POTS_15 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom, userTo)
          )
      )
endproc (* POTS_2 *)


process POTS_4 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom:Address, userTo:Address)
                : exit(Nat) :=
    (
      StartAR !userFrom !userTo; exit(userFrom, userTo)
      |||
      StartR !userTo !userFrom; exit(userFrom, userTo)
    )
    >> accept userFrom:Address, userTo:Address in
    (
      i; POTS_5 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
```



```
                 Disconnect, Display, Success] (userFrom, userTo)
      []
      i; POTS_13[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                 Disconnect, Display, Success] (userFrom, userTo)
    )
endproc (* POTS_4 *)


process POTS_5 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom:Address, userTo:Address)
                : exit(Nat) :=
    OffHook !userTo;
    (
      (* State 6 *)
      StopAR !userFrom !userTo; exit(userFrom, userTo)
      |||
      StopR !userTo !userFrom; exit(userFrom, userTo)
    )
    >> accept userFrom:Address, userTo:Address in
    (
      i; OnHook !userFrom;            (* State 7 *)
      Disconnect !userTo !userFrom;   (* State 8 *)
      onHook !userTo; exit(0)         (* State 9 *)
      []
      i; OnHook !userTo;              (* State 10 *)
      Disconnect !userFrom !userTo;   (* State 11 *)
      OnHook !userFrom; exit(succ(0)) (* State 12 *)
    )
endproc (* POTS_5 *)


process POTS_13[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom:Address, userTo:Address)
                : exit(Nat) :=
    OnHook !userFrom;
    (
      (* State 14 *)
      StopAR !userFrom !userTo; exit(succ(succ(0)))
      |||
      StopR !userTo !userFrom; exit(succ(succ(0)))
    )
endproc (* POTS_13 *)


process POTS_15[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success] (userFrom:Address, userTo:Address)
                : exit(Nat) :=
    LineBusyTone !userFrom;
    OnHook !userFrom; exit(succ(succ(succ(0)))) (* State 16 *)
endproc (* POTS_15 *)


(*==================================================================*)
(*                                                                  *)
(*                           TEST PROCESSES                         *)
(*                                                                  *)
(*==================================================================*)

(**********)
(** POTS **)
(**********)
```

```
(* TEST CASES *)
process tPOTS1 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Init, CreateUser, Query, Success] : noexit :=
  (* Cases where userB is not busy. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, NoFList, undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !NoFList;
    POTS_1 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display, Success] (userA, userB)

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(InitTime)),
               Insert(l(Begin, userA, userB, userA, InitTime), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tPOTS1 *)

process tPOTS2 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Init, CreateUser, Query, Success] : noexit :=
  (* Cases where userB is busy. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, NoFList, undefined, undefined, NoAddList, validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !NoFList;
    POTS_1 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display, Success] (userA, userB)

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tPOTS2 *)

(**********)
(** INTL **)
(**********)

(* COMMON BEHAVIOUR *)
process cINTL1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Success] : exit(Nat) :=
  (* Cases where TeenTime is restricted and A provides the valid PIN. *)
    OffHook !userA;
      Announce !userA !AskForPIN;
      (
        i; Dial !userA !validPIN;
          POTS_2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                 Disconnect, Display, Success] (userA, userB)
        []
        i; OnHook !userA; exit(succ(succ(succ(succ(succ(0))))))
      )
endproc (* cINTL1 *)

process cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Success] : exit(Nat) :=
  (* Cases where TeenTime is restricted and A does not provide the valid PIN. *)
    OffHook !userA;
      Announce !userA !AskForPIN;
      (
        i; Dial !userA !invalidPIN;
          Announce !userA !invalidPIN;
            OnHook !userA; exit (succ(succ(succ(succ(succ(succ(0)))))))
        []
        i; OnHook !userA; exit(succ(succ(succ(succ(succ(0))))))
      )
endproc (* cINTL2 *)

(* TEST PROCESSES *)
process tINTL1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where TeenTime is not restricted. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, NoFList, undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, tic(tic(initTime)), tic(tic(tic(initTime))), undefined,
                                              validPIN), NoSCPDB)
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !NoFList;
  POTS_1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success](userA, userB)

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userA, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tINTL1 *)

process tINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
  (* Cases where TeenTime is restricted and A provides the valid PIN. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, NoFList, undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !NoFList;
  cINTL1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userA, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tINTL2 *)

process tINTL3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where TeenTime is restricted and A does not provide the valid PIN. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, NoFList, undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !NoFList;
  cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection *)
      [(exitCode eq succ(succ(succ(succ(succ(0)))))) or (exitCode eq
                                        succ(succ(succ(succ(succ(succ(0)))))))] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tINTL3 *)

(*********)
(** CND **)
(*********)

(* COMMON BEHAVIOUR *)
process cCND1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Success] : exit(Nat) :=
  (* Starts at POTS state 2. *)
```



```
  (* Should Display the originator’s number. *)
      DialTone !userA;
        Dial !userA !userB;
        (
          StartAR !userA !userB; exit(userA, userB)
          |||
          StartR !userB !userA; exit(userA, userB)
          |||
          Display !userB !userA; exit(userA, userB)
        )
        >> accept userFrom:Address, userTo:Address in
        (
          i; POTS_5 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                     StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                     Disconnect, Display, Success] (userFrom, userTo)
          []
          i; POTS_13[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                     StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                     Disconnect, Display, Success] (userFrom, userTo)
        )
endproc (* cCND1 *)

(* TEST PROCESSES *)
process tCND1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Should Display the originator’s number. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(CND, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(CND, NoFList);
    OffHook !userA;
      cCND1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(InitTime)),
               Insert(l(Begin, userA, userB, userA, InitTime), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tCND1 *)

process tCND2 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
               StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
               Disconnect, Display, Init, CreateUser, Query, Success] : noexit :=
  (* Cases where userB is busy. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(INFB, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
```

```
  CreateUser !userB !Insert(INFB, NoFList);
    POTS_1 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display, Success] (userA, userB)

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tCND2 *)

(*********)
(** INFB **)
(*********)

(* NO SPECIAL COMMON BEHAVIOUR *)
(* TEST PROCESSES *)
process tINFB1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where B is not Busy. Affect the billing. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(INFB, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(INFB, NoFList);
  POTS_1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success](userA, userB)

    (* Check the Log. UserB should be charged. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userB, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tINFB1 *)

process tINFB2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where B is Busy. Do not affect billing. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(INFB, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(INFB, NoFList);
  POTS_1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
         Disconnect, Display, Success](userA, userB)

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tINFB2 *)

(********)
(** TCS **)
(********)

(* COMMON BEHAVIOUR *)
process cTCS1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
              Disconnect, Display, Success] : exit(Nat) :=
  OffHook !userA;
    DialTone !userA; (* State 2 *)
      (
        i; OnHook !userA; exit(succ(succ(succ(succ(0)))))  (* State 17 *)
        []
        i; Dial !userA !userB; (* State 3 *)
          POTS_4[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                 Disconnect, Display, Success](userA, userB)
      )
endproc (* cTCS1 *)

process cTCS2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
              Disconnect, Display, Success] : exit(Nat) :=
  OffHook !userA;
    DialTone !userA; (* State 2 *)
      (
        i; OnHook !userA; exit(succ(succ(succ(succ(0)))))  (* State 17 *)
        []
        i; Dial !userA !userB; (* State 3 *)
          POTS_15[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                 Disconnect, Display, Success](userA, userB)
      )
endproc (* cTCS2 *)

process cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
              Disconnect, Display, Success] : exit(Nat) :=
  OffHook !userA;
    DialTone !userA; (* State 2 *)
      (
        i; OnHook !userA; exit(succ(succ(succ(succ(0)))))  (* State 17 *)
        []
        i; Dial !userA !userB; (* State 3 *)
          Announce !userA !ScreenedMessage;
            OnHook !userA;
              exit(succ(succ(succ(succ(0)))))  (* TCS State 4, same as POTS State 17 *)
      )
endproc (* cTCS3 *)

(* TEST PROCESSES *)
process tTCS1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
```

```
              Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where B is not Busy and A is not screened. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userC, NoAddList), validPIN),
                                                 NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, NoFList);
    cTCS1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. UserA should be charged. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(InitTime)),
               Insert(l(Begin, userA, userB, userA, InitTime), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tTCS1 *)

process tTCS2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
              Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where B is Busy and A is not screened. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userC, NoAddList), validPIN),
                                                 NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, NoFList);
    cTCS2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tTCS2 *)

process tTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
              StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
              Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
  (* Cases where A is screened. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userA, NoAddList), validPIN),
                                                 NoSDB))
       !NoStatus
       !NoSCPDB
```



```
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, NoFList);
    cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
endproc (* tTCS3 *)

(*===================================================================*)
(*                                                                   *)
(*                        FI TEST PROCESSES                          *)
(*                                                                   *)
(*===================================================================*)

(**************)
(* INTL - CND *)
(**************)

process fiINTL_CND[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                   StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                   Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
(* Should Display the originator's number. *)

(
(* Cases where TeenTime is not restricted. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(CND, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, tic(tic(initTime)), tic(tic(tic(initTime))), undefined,
                                                 validPIN), NoSCPDB)
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(CND, NoFList);
    OffHook !userA;
      cCND1 [OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
             StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
             Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userA, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* tINTL2 *)
```

```
(* Cases where TeenTime is restricted and A provides the valid PIN. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(CND, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(CND, NoFList);
    OffHook !userA;
      Announce !userA !AskForPIN;
        Dial !userA !validPIN;
          cCND1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                 StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                 Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userA, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* tINTL3 *)
(* Cases where TeenTime is restricted and A does not provide the valid PIN. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(CND, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(CND, NoFList);
  cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection *)
      [(exitCode eq succ(succ(succ(succ(succ(0)))))) or (exitCode eq
                                        succ(succ(succ(succ(succ(succ(0)))))))] ->
        Query !NoLog;
          Success; stop
    )
)
endproc (* fiINTL_CND *)

(***************)
(* INTL - INFB *)
(***************)

process fiINTL_INFB[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                    StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
                    Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
(* Should Display the originator's number. *)

(
(* Cases where TeenTime is not restricted. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(INFB, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, tic(tic(tic(initTime))),
                     tic(tic(tic(tic(initTime)))), undefined, validPIN), NoSCPDB)
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(INFB, NoFList);
  POTS_1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success](userA, userB)

    (* Check the Log. UserB should be charged. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(tic(InitTime)))),
               Insert(l(Begin, userA, userB, userB, tic(tic(InitTime))), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* tINTL2 *)
(* Cases where TeenTime is restricted and A provides the valid PIN. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(INFB, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(tic(initTime))), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(INFB, NoFList);
  cINTL1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log. UserB should be charged. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(tic(InitTime)))),
               Insert(l(Begin, userA, userB, userB, tic(tic(InitTime))), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
```

```
(* tINTL3 *)
(* Cases where TeenTime is restricted and A does not provide the valid PIN. *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(INFB, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(tic(initTime))), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(INFB, NoFList);
  cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection *)
      [(exitCode eq succ(succ(succ(succ(succ(0)))))) or (exitCode eq
                                              succ(succ(succ(succ(succ(succ(0)))))))] ->
        Query !NoLog;
          Success; stop
    )
)
endproc (* fiINTL_INFB *)

(****************)
(** CND - INFB **)
(****************)

process fiCND_INFB[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=
(
  (* Should Display the originator's number. *)
  (* Cases where B is not Busy. Affect the billing. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(CND, Insert(INFB, NoFList)), undefined, undefined, NoAddList, validPIN),
                                              NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(CND, Insert(INFB, NoFList));
    OffHook !userA;
      cCND1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display, Success]

    (* Check the Log. UserB should be charged. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
              Insert(l(Begin, userA, userB, userB, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
```



```
(
  (* Cases where B is Busy. Does not affect billing. *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(CND, Insert(INFB, NoFList)), undefined, undefined, NoAddList, validPIN),
                                              NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(CND, Insert(INFB, NoFList));
  POTS_1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success](userA, userB)

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )

)
endproc (* fiCND_INFB *)

(**************)
(* INTL - TCS *)
(**************)

process fiINTL_TCS[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                   StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                   Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=

(
(* Cases where A's TeenTime is not restricted, A is not on B's TCS list, and B is idle. *)
(* tTCS1 and tINTL1 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userC, NoAddList), validPIN),
                                              NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, tic(tic(initTime)), tic(tic(tic(initTime))), undefined,
                                              validPIN), NoSCPDB)
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
    cTCS1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
              Insert(l(Begin, userA, userB, userA, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
```

```
[]
(
(* Cases where A's TeenTime is not restricted, A is not on B's TCS list, and B is Busy. *)
(* tTCS2 and tINTL1 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userC, NoAddList), validPIN),
                                               NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !Insert(scp(TeenTime, userA, undefined, tic(tic(initTime)), tic(tic(tic(initTime))), undefined,
                                              validPIN), NoSCPDB)
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
  cTCS2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
        StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
        Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is not restricted, and A is on B's TCS list. *)
(* tTCS3 and tINTL1 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userA, NoAddList), validPIN),
                                               NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !Insert(scp(TeenTime, userA, undefined, tic(tic(initTime)), tic(tic(tic(initTime))), undefined,
                                              validPIN), NoSCPDB)
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
    cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is restricted, A has the valid PIN, A is not on B's TCS list, and B is idle. *
                                              )
(* tTCS1 and tINTL2 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userC, NoAddList), validPIN),
                                               NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
          Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
```



```
  CreateUser !userB !Insert(TCS, NoFList);
  cINTL1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userA, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is restricted, A has the valid PIN, A is not on B's TCS list, and B is busy. *
                                                  )
(* tTCS2 and tINTL2 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
  cINTL1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is restricted, A has the valid PIN, A is on B's TCS list. *)
(* tTCS3 and tINTL2 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userA, NoAddList), validPIN),
                                               NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
    OffHook !userA;
      Announce !userA !AskForPIN;
      (
        i; Dial !userA !validPIN;
          DialTone !userA;
            Dial !userA !userB;
```

```
              Announce !userA !ScreenedMessage;
                OnHook !userA;
                  exit(succ(succ(succ(succ(0)))))  (* TCS State 4, same as POTS State 17 *)
        []
        i; OnHook !userA; exit(succ(succ(succ(0))))
      )

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is restricted, A has an invalid PIN, A is not on B's TCS list, and B is idle*)
(* tTCS1 and tINTL3 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, invalidPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
  cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection *)
      [(exitCode eq succ(succ(succ(succ(succ(0)))))) or (exitCode eq
                                          succ(succ(succ(succ(succ(succ(0)))))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is restricted, A has an invalid PIN, A is not on B's TCS list, and B is busy*)

(* tTCS1 and tINTL3 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, invalidPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, NoAddList, validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
  cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection *)
      [(exitCode eq succ(succ(succ(succ(succ(0)))))) or (exitCode eq
                                          succ(succ(succ(succ(succ(succ(0)))))))] ->
```



```
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A's TeenTime is restricted, A has an invalid PIN, A is on B's TCS list *)
(* tTCS1 and tINTL3 *)
  Init !Insert(sub(userA, Insert(INTL, NoFList), undefined, undefined, NoAddList, invalidPIN),
        Insert(sub(userB, Insert(TCS, NoFList), undefined, undefined, Insert(userA, NoAddList), validPIN),
                                        NoSDB))
       !NoStatus
       !Insert(scp(TeenTime, userA, undefined, initTime, tic(tic(initTime)), undefined, validPIN),
        Insert(scp(TeenPIN, userA, undefined, initTime, initTime, undefined, validPIN), NoSCPDB))
       !InitTime;
  CreateUser !userA !Insert(INTL, NoFList);
  CreateUser !userB !Insert(TCS, NoFList);
  cINTL2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
         StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
         Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* No connection *)
      [(exitCode eq succ(succ(succ(succ(succ(0)))))) or (exitCode eq
                                          succ(succ(succ(succ(succ(succ(0)))))))] ->
        Query !NoLog;
          Success; stop
    )
)
endproc (* fiINTL_TCS *)

(*************)
(* CND - TCS *)
(*************)

process fiCND_TCS[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                  StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                  Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=

(
(* Cases where A is not on B's TCS list, and B is idle. Should Display the originator's number. *)
(* tTCS1 and tCND1 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(CND, NoFList)), undefined, undefined, NoAddList, validPIN),
                                        NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(CND, NoFList));
    OffHook !userA;
      cCND1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
            StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
            Disconnect, Display, Success]

    (* Check the Log *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(InitTime)),
               Insert(l(Begin, userA, userB, userA, InitTime), NoLog));
          Success; stop
```

```
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A is not on B's TCS list, and B is busy. *)
(* tTCS2 and tCND2 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(CND, NoFList)), undefined, undefined,
                                              Insert(userC, NoAddList), validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(CND, NoFList));
    cTCS2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A is on B's TCS list, and B is busy. Do not display. *)
(* tTCS3 and tCND2 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(CND, NoFList)), undefined, undefined,
                                              Insert(userA, NoAddList), validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(CND, NoFList));
    cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A is on B's TCS list, and B is idle. Do not display. *)
(* tTCS3 and tCND1 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(CND, NoFList)), undefined, undefined,
                                              Insert(userA, NoAddList), validPIN), NoSDB))
       !NoStatus
```


Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features

```
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(CND, NoFList));
    cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
endproc (* fiCND_TCS *)


(**************)
(* INFB - TCS *)
(**************)

process fiINFB_TCS[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
                   StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
                   Disconnect, Display, Success, CreateUser, Query, Init] : noexit :=

(
(* Cases where A is not on B's TCS list, and B is idle. Affect the billing. *)
(* tTCS1 and tINFB1 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(INFB, NoFList)), undefined, undefined,
                                              Insert(userC, NoAddList), validPIN), NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(INFB, NoFList));
    cTCS1[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. UserB should be charged. *)
    >> accept exitCode:Nat in
    (
      (* One connection *)
      [(exitCode eq 0) or (exitCode eq succ(0))] ->
        Query !Insert(l(End, userA, userB, undefined, tic(tic(InitTime))),
               Insert(l(Begin, userA, userB, userB, tic(InitTime)), NoLog));
          Success; stop
      []
      (* No connection *)
      [not( (exitCode eq 0) or (exitCode eq succ(0)) )] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A is not on B's TCS list, and B is busy. Do not affect the billing. *)
(* tTCS2 and tINFB2 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(INFB, NoFList)), undefined, undefined,
                                              Insert(userC, NoAddList), validPIN), NoSDB))
```

```
      !Insert(stat(Busy, userB, undefined), NoStatus)
      !NoSCPDB
      !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(INFB, NoFList));
    cTCS2[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A is on B's TCS list, and B is busy. Do not affect the billing. *)
(* tTCS3 and tINFB2 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(INFB, NoFList)), undefined, undefined,
                                              Insert(userA, NoAddList), validPIN), NoSDB))
       !Insert(stat(Busy, userB, undefined), NoStatus)
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(INFB, NoFList));
    cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
          Success; stop
    )
)
[]
(
(* Cases where A is on B's TCS list, and B is idle. Do not affect the billing. *)
(* tTCS3 and tINFB1 *)
  Init !Insert(sub(userA, NoFList, undefined, undefined, NoAddList, validPIN),
        Insert(sub(userB, Insert(TCS, Insert(INFB, NoFList)), undefined, undefined,
                                              Insert(userA, NoAddList), validPIN), NoSDB))
       !NoStatus
       !NoSCPDB
       !InitTime;
  CreateUser !userA !NoFList;
  CreateUser !userB !Insert(TCS, Insert(INFB, NoFList));
    cTCS3[OffHook, OnHook, Dial, Flash, DialTone, StartAR, StartR,
          StartCWT, StopAR, StopR, StopCWT, LineBusyTone, Announce,
          Disconnect, Display, Success]

    (* Check the Log. *)
    >> accept exitCode:Nat in
    (
      (* No connection only. *)
      [(exitCode eq succ(succ(succ(0)))) or (exitCode eq succ(succ(succ(succ(0)))))] ->
        Query !NoLog;
```



```
          Success; stop
    )
)
endproc (* fiINFB_TCS *)

endspec (* FI_UCM *)
```

B ERRONEOUS SPECIFICATION FOR STUB PROCESS-CALL

Here is the part of the incorrect LOTOS specification that was replaced by lines 1361 to 1395 in the correct specification of appendix A.

```
1361          process ProcessCallStub [OffHook, OnHook, Dial, Flash, DialTone,
1362                    StartAR, StartR, StartCWT, StopAR, StopR, StopCWT,
1363                    LineBusyTone, Announce, Disconnect, Display,
1364                    Trigger, Resource, Response, LogBegin, LogEnd, Time]
1365                        (inPaths: SPList, userFrom: Address, userTo:Address,
1366                         sdb: SDB, status: Status)
1367                    : exit (Address, Address, Address, SDB, Status, SPList) :=
1368
1369            (* CND will be taken care of at outPC1, after all these plug-ins. *)
1370
1371            (* TCS *)
1372            [has(userTo, TCS, sdb)] ->
1373              PluginTCS[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
1374                        StartR, StartCWT, StopAR, StopR, StopCWT,
1375                        LineBusyTone, Announce, Disconnect, Display,
1376                        Trigger, Resource, Response, LogBegin, LogEnd, Time]
1377                          (inPaths, userFrom, userTo, sdb, status)
1378            []
1379            (* INFB *)
1380            [has(userTo, INFB, sdb)] ->
1381              PluginINFB[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
1382                         StartR, StartCWT, StopAR, StopR, StopCWT,
1383                         LineBusyTone, Announce, Disconnect, Display,
1384                         Trigger, Resource, Response, LogBegin, LogEnd, Time]
1385                           (inPaths, userFrom, userTo, sdb, status)
1386            []
1387            (* Default *)
1388            [not(has(userTo, INFB, sdb)) and not(has(userTo, TCS, sdb))] ->
1389              PluginDefault[OffHook, OnHook, Dial, Flash, DialTone, StartAR,
1390                            StartR, StartCWT, StopAR, StopR, StopCWT,
1391                            LineBusyTone, Announce, Disconnect, Display,
1392                            Trigger, Resource, Response, LogBegin, LogEnd, Time]
1393                              (inPaths, userFrom, userTo, sdb, status)
1394          where
1395
1396            process PluginTCS [OffHook, OnHook, Dial, Flash, DialTone,
1397                    StartAR, StartR, StartCWT, StopAR, StopR, StopCWT,
1398                    LineBusyTone, Announce, Disconnect, Display,
1399                    Trigger, Resource, Response, LogBegin, LogEnd, Time]
1400                        (inPaths: SPList, userFrom: Address, userTo:Address,
1401                         sdb: SDB, status: Status)
1402                    : exit (Address, Address, Address, SDB, Status, SPList) :=
1403
1404              (* TCS plugin for ProcessCallStub *)
1405              [isOnTCS(userFrom, userTo, SDB)] ->
1406                (* Caller on the list. Reject call. *)
1407                exit (userFrom, userTo, userFrom, sdb, status, Insert(outPC3, NoSPList))
1408              []
1409              [not(isOnTCS(userFrom, userTo, SDB))]->
1410                (* Caller NOT on the list. Continue. *)
1411                (
1412                  [IsIdle(userTo, status)] ->
1413                    exit (userFrom, userTo, userFrom, sdb, status, Insert(outPC1, NoSPList))
1414                  []
1415                  [IsBusy(userTo, status)] ->
1416                    exit (userFrom, userTo, userFrom, sdb, status, Insert(outPC2, NoSPList))
1417                )
1418            endproc (* PluginTCS *)
```