Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS

D. Amyot[1], L. Charfi[1], N. Gorse[1], T. Gray[2],
L. Logrippo[1], J. Sincennes[1], B. Stepien[1], and T. Ware[2]
[1]*TSERG/SITE, University of Ottawa, and [2]Mitel Corp., Ottawa, Canada*
{damyot, lcharfi, ngorse, luigi, jack, bernard}@site.uottawa.ca
{tom_gray, tom_ware}@mitel.com

Abstract. A methodology for feature design, specification, and validation is presented. The methodology is based on Use Case Maps for the description of features and on LOTOS with its tools for animation of the features, for feature validation, and for feature interaction detection. It has been developed as a collaborative project between the University of Ottawa and Mitel Corporation, and is being used experimentally to design, specify and validate the features of Mitel's new PBX.

1 Introduction

A methodology for feature design, specification, and validation, is being developed in a joint project between Mitel Corporation and the University of Ottawa. This methodology uses *Use Case Maps* (UCMs) [5][7] for the design and documentation of features, and *LOTOS* [12] for the formal specification of features and for their formal verification, including the detection of undesirable interactions.

The UCM notation allows designers to describe scenarios visually. System-wide scenario paths connect abstract responsibilities and (possibly) components in causal flows. It has been shown in [2][6] that UCMs can be useful for describing telephony features and agent systems, and for reasoning about potential interactions at a high level of abstraction, without commitment to detailed design decisions.

LOTOS is an algebraic language with a history of applications to validation of distributed systems in general and to feature interaction detection in particular (see [3][9][15][22][23], among an extensive literature).

UCMs and LOTOS are complementary in many ways. UCMs provide a visual notation to capture, integrate, and reason about functional requirements, whereas LOTOS provides a means for formalization, abstract prototyping, and validation. Since in the normal software development process the former activities precede the latter, our method starts with the use of UCMs and continues with the use of LOTOS (although finer interleaving is possible).

It should be noted at the outset that this methodology is based on existing ideas, documented in publications by the authors of this paper or by other authors in the UCM and LOTOS community. Some of the ideas of the methodology were previously applied to an artificial example, i.e. the features described in the 1998 Feature Interaction Contest [2], and to another example involving the mobile data standard GPRS [3]. The contribution of this paper is to show how these ideas were combined for use in an industrial context for the development of an agent-based private branch exchange (PBX). A related approach was presented in [17].

After an introduction to UCMs, the paper provides a brief overview of the system architecture and of the basic call model. Basic LOTOS concepts are then recalled, and the generation of a LOTOS prototype from the UCMs is discussed. The validation of the LOTOS model is done through test cases derived from the UCMs, and the detection of undesirable interactions between pairs of features is also based on a testing approach.

2 Use Case Maps (UCMs)

Use Case Maps are used to emphasize abstract sequencing (*causal paths*) among the most relevant, interesting, and critical functionalities of reactive and distributed systems, which are composed of *responsibilities*. UCMs can be *unbound*, i.e. without reference to components, or *bound*, in which case they will show components. Note that only bound UCMs will be seen in this paper. Unbound UCMs, however, can be useful at the earliest stages of design, when the components are still undefined. Responsibilities along causal paths can be internal to a component or can be observable. They can be chosen to be of different granularities, as required by the stage of design were they are defined, and usually well above the level of specific message exchanges. Therefore, UCMs can represent specific scenarios, as well as abstract (generic) ones and can cover multiple scenario instances. UCMs are often highly reusable and, normally, the coarser the grain of their responsibilities, the greater their reusability.

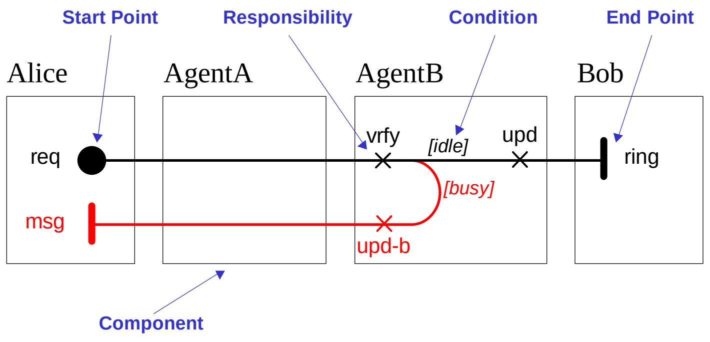


Figure 1 A simple Use Case Map.

Figure 1 shows a simple bound UCM where a user (Alice) attempts to establish a telephone call with another user (Bob) through some network of agents. Each user has an agent responsible for managing subscribed telephony features such as *Outgoing Call Screening* (OCS). Alice first sends a connection request (req) to the network through her agent. This request causes the called agent to verify (vrfy) whether the called party is idle or busy (conditions are italicized). If he is, then there will be some status update (upd) and a ring signal will be activated on Bob's side (ring). Otherwise, a different update will occur (upd-b) and an appropriate message (stating that Bob is not available) will be prepared and sent back to Alice (msg).

A scenario starts with a *start point* which represents a *triggering event* that can be associated with certain *pre-conditions* (filled circle labeled req) and ends with one or more *end points*, representing *resulting events* that can be associated with certain *post-conditions* (bars), in our case ring or msg. Intermediate responsibilities (vrfy, upd, upd-b) have been activated along the way. A *causal path* goes from a start point to an end point. In this example, the responsibilities are allocated (bound) to abstract components (boxes Alice, AgentA, Bob and AgentB), which could be seen as objects, processes, agents, databases, or even roles, actors, or persons.

Figure 1 is a simple diagram, yet it conveys much information in a compact form, and it allows for requirements engineers and designers to use two dimensions (structure and behavior) to evaluate architectural alternatives for their systems. This view is reusable, in the sense that system behavior is shown at a level close to the functional level, without implementation details. The same view can be of interest not only to designers, but also to product managers and even users.

2.1 UCMs, Messages, and Architectural Reasoning

Use Case Maps can be refined in terms of *Message Sequence Charts* (MSC) [14] or UML interaction diagrams [24]. As mentioned, UCMs do not explicitly define message exchanges between components, but, in the design process, messages need to be constructed in such a way that the causal relationships between responsibilities from different components are satisfied. There are usually many ways to do so, depending on the available interfaces, communication channels, and protocols.

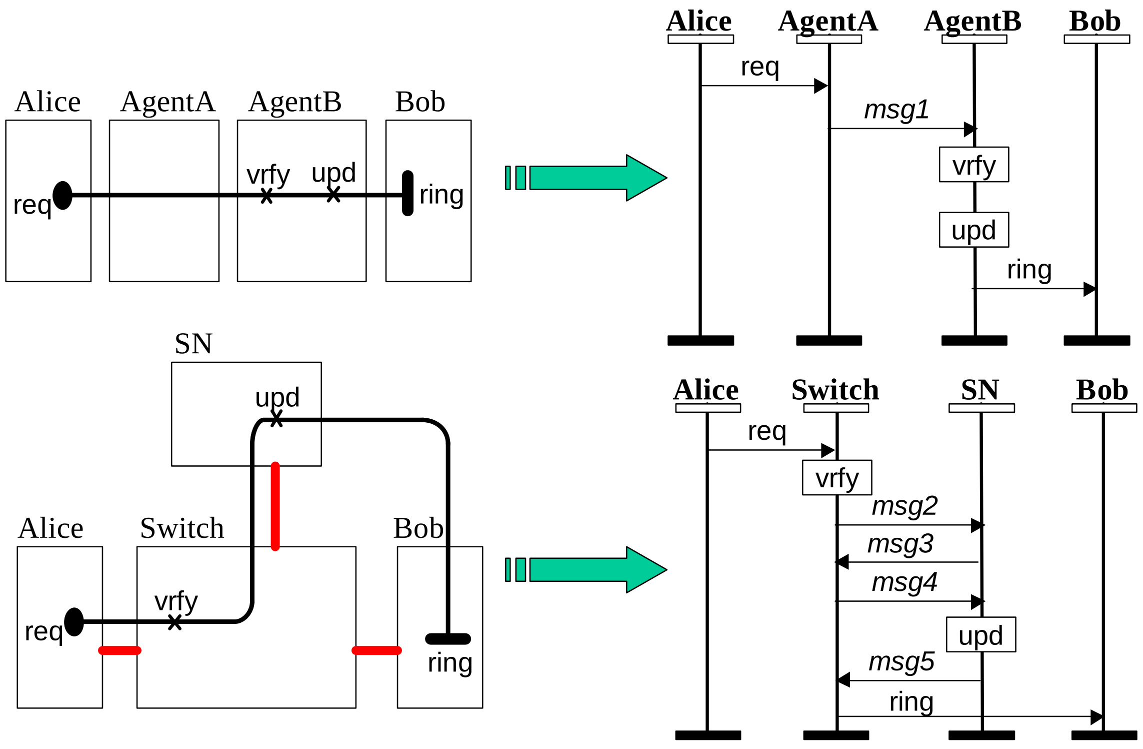


Figure 2 Refining UCMs with message exchanges.

The causal path <req, vrfy, upd, ring>, which represents a successful scenario extracted from Figure 1, will serve as an example. At the top of Figure 2, this path is bound to our component structure, where implicit communication channels exist between users and their respective agents, and between agents. Start points and end points are interpreted essentially as messages themselves, whereas responsibilities are MSC actions performed by components. New messages (e.g. *msg1*) are necessary for the causal flow to remain valid. These messages can be of various nature and complexity. UCMs allow designers to describe functionalities even when messages are not known in advance, e.g. when complex negotiations are involved between agents.

UCMs enable one to reuse the same paths on different architectural alternatives. For example, the UCM at the bottom of Figure 2 reuses the same path as the top UCM, but on a different set of components. Here, the responsibilities are allocated to traditional telephony components such as a switch and a service node (SN), with different dependencies. Lines have been added to represent communication links, hence constraining the potential senders and receivers of each message. Different decisions about the protocols and control can lead to multiple solutions, one of which is given on the right (note that SN and Bob cannot communicate directly).

Since they can be easily decoupled from structures, UCM paths improve the reusability of scenarios and lead to behavior patterns that can be used across a wide range of applications. On many occasions, UCMs may provide helpful visual patterns that stimulate thinking and discussion about system issues and that may be reused. Note also that this notation enables evaluation of architectural alternatives to be done at a high level of abstraction, without early commitment to messages and protocols as in Message Sequence Charts.

2.2 Additional UCM Notation

To introduce further notation elements, new features can be added to the basic scenarios. Figure 3 abstracts from the component instances introduced in Figure 1. The components do not refer to Bob and Alice any longer, rather they refer to more generic call origination and termination roles (for both users and agents). Dashed components are called *slots* and may be populated with different instances at different times. They can represent roles of a particular class of components.

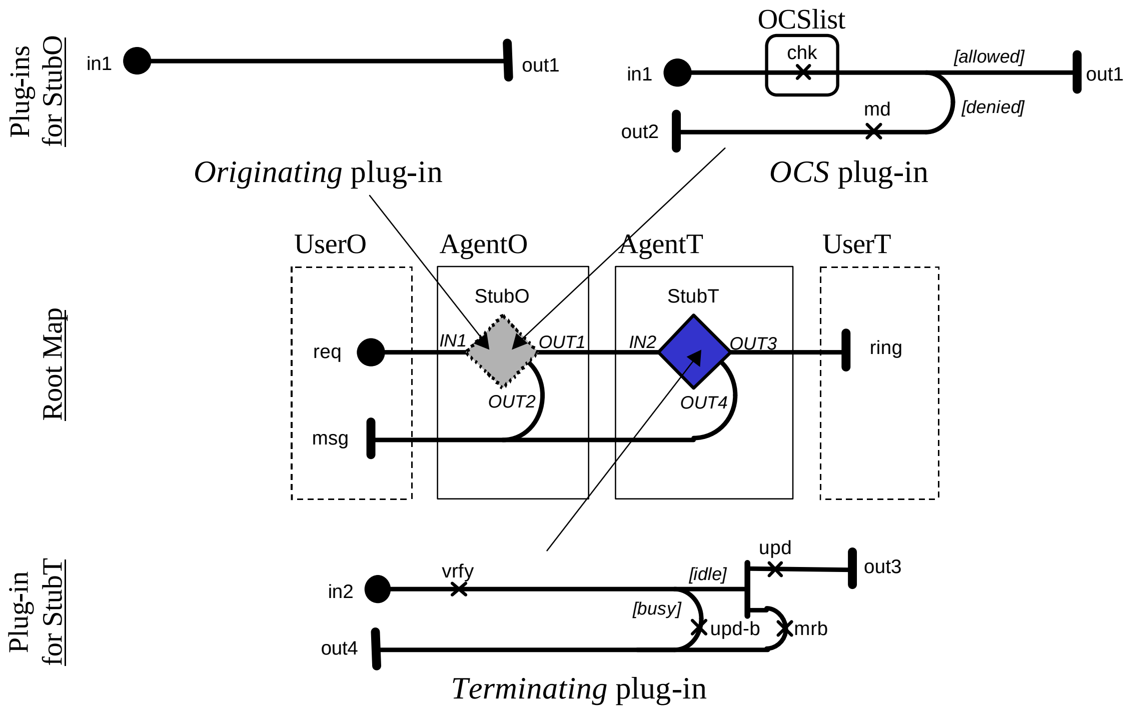


Figure 3 More complex call connection and new notation elements.

In [5][7], Buhr introduces an architectural notation with different types of components and richer semantics (active processes, passive objects, groupings, pools of objects, interrupt service requests, agents, mutex, etc.). These components can also be dynamic, i.e. can be created, destroyed, stored, and moved around in a way similar to data. Since their definition would take much space, the next examples will provide only a few intuitive descriptions.

The middle part of Figure 3 shows an enhanced version of the UCM in Figure 1 that represents a whole class of related scenario instances. It is called a *root map* because this

UCM possesses containers (called *stubs*) for sub-maps (called *plug-ins*), and this UCM is not itself a sub-map. Stubs are of two kinds:

- ***Static stubs***: represented as plain diamonds (e.g. StubT). They contain only one plug-in, hence enabling hierarchical decomposition of complex maps.
- ***Dynamic stubs***: represented as dashed diamonds (e.g. StubO). They may contain several plug-ins, whose selection can be determined at run-time according to a selection policy (often described with pre-conditions).

Path segments coming in and going out of stubs have been identified on the root map (italicized labels). For instance, the originating dynamic stub StubO has two plug-ins: *Originating* and *OCS*. The start point of the *Originating* plug-in (in1) is bound to the incoming path segment *IN1*, and the end point out1 is bound to the outgoing segment *OUT1*. Figure 3 makes use of similar labels for a clear binding relation between plug-ins and stubs, but in general names can be different and the relation has to be described explicitly.

The OCS plug-in shows a new component (the passive object OCSlist) that represents a list of screened numbers that the originating user (UserO) is forbidden to contact. If UserO is subscribed to the Outgoing Call Screening service, then the *OCS* plug-in is selected instead of the *Originating* plug-in. In this case, the called number is checked against the list (chk). If the call is denied, an appropriate message is sent back to the originating party (md).

The *Terminating* plug-in improves on the original UCM by allowing the update (upd) and the ring result to be accompanied, concurrently, by a ring-back signal to be prepared and sent back to the originating party (mrb). Concurrency is represented here by an AND-fork (thin bar). The notation allows for alternative paths (OR-fork and OR-join, as in the *Terminating* plug-in), concurrent paths (AND-fork and AND-join), shared responsibilities, exception paths, timers, failure points, error handling, and (a)synchronous interactions between paths, to name but a few elements.

Once stubs are defined at key points on a path, it becomes simple to add new plug-ins, which could represent new features in our example. Existing maps and plug-ins can further be decomposed or extended (e.g. when a radically different service is added) with new paths and new static and dynamic stubs. By selecting plug-ins for the stubs in the integrated view, one can obtain a flattened map, which still contains multiple possible end-to-end scenarios.

This method of hierarchical representation, using maps containing at different locations several types of stubs for different features and other system characteristics, appears to be general and flexible, i.e. adaptable to many architectures. Obviously, it is important that designers exercise much care in properly defining the root maps and the stubs to allow for the type of modifications and refinements that can be expected at various stages of design.

Systems for the new converged voice and data marketplace are expected to be highly dynamic in many respects. They should be dynamically customizable to suit the needs of individual enterprises. They should also adapt to rapidly changing network environments, and to highly dynamic applications. UCMs, with their dynamic components, pluggable behavior, compact form and multiple levels of abstraction, make dynamic architectures understandable and hence easier to implement. This capability was found to be lacking in other popular software engineering description techniques or CASE tools that were examined.

A graphical tool, called *UCM Navigator*, has been developed at Carleton University in order to help drawing UCMs [18][25]. Among other features, this tool maintains various bindings (plug-ins to stubs, responsibilities to components, etc.), as well as the pre- and post-conditions. Also, it allows users to navigate much like a Web browser, and to visit and edit the plug-ins related to stubs of all levels.

3 Basic System Architecture

The highly dynamic nature of modern telecommunications systems was noted above. This characteristic must be addressed while maintaining a required quality of service. The system in consideration is a new architecture [27] being designed by Mitel. Call processing goes through three types of components (agents):

- ***Device Agents*** (DAGENT): handle the devices, i.e. the physical endpoint of a call (e.g. telephone, a computer capable of voice over IP, diary, etc.).
- ***Personal Agents*** (PAGENT): know the restrictions and privileges given to specific users. They also know about the preference relationships between a user and his/her devices.
- ***Functional Agents*** (FAGENT): represent the functional role of the endpoint of a call (e.g. president, director, secretary, etc.).

Many instances of DAGENTs, PAGENTs and FAGENTs can be active simultaneously for one user. From a scenario perspective, it is also helpful to distinguish between the roles played by these components (e.g. originating or terminating). The features are processed in the components as appropriate.

The exact functionalities of these components are not important for the understanding of the methodology and thus will not be described further.

4 Use Case Maps for the Call Model and UCM-based FI Detection

The Mitel system architecture, including the various features, was captured using UCMs, for which an example is shown in Figure 4. This figure shows part of the Root UCM, limited to the originating DAGENT, PAGENT, FAGENT elements. This scenario starts with the acquisition of a string representing the role or person to be called. Note the existence of several dynamic stubs, which contain the plug-in UCM(s) describing the stub as well as the appropriate selection policy. Feature UCMs are plugged into these stubs.

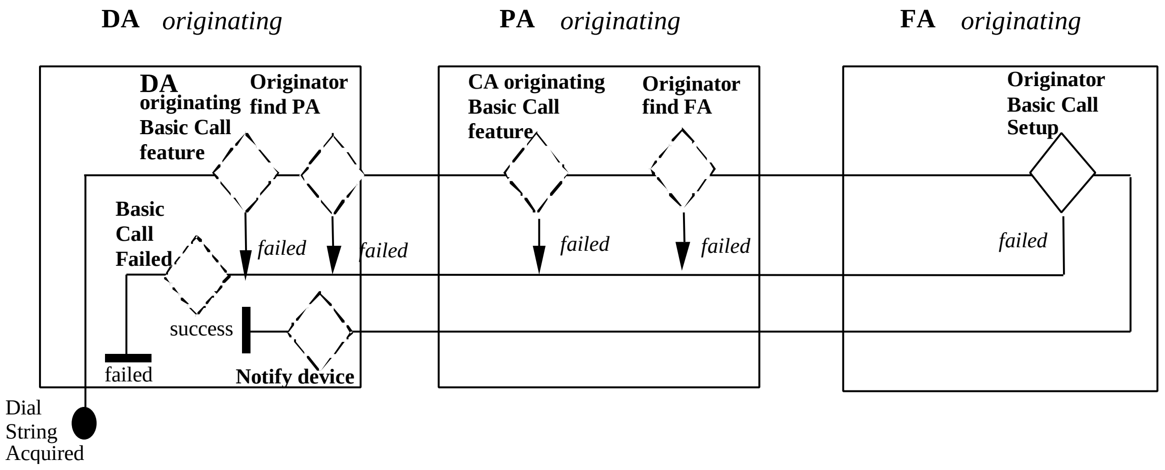


Figure 4 Part of the Basic Call Root UCM.

More than 100 UCMs, created by Mitel designers, were structured in a hierarchical way to describe the 10 features that were considered in this project. Nearly 60 stubs were used along the way, and many plug-in UCMs were reused in more than one stub. The responsibilities are found in the plug-ins, sometimes several levels deep in the hierarchy. This set of UCMs is currently being extended to support additional features. Due to the already large number of stubs and functionalities in the current UCMs, and given the reusability of these, we expect to need few additional UCMs to capture the extra features.

UCMs can help directly in the detection and avoidance of many undesirable interactions before any prototype is generated [2]. By inspecting specific locations such as dynamic stubs, designers may select appropriate strategies. For instance, a selection policy where preconditions are incomplete or overlapping (non-deterministic) is likely to cause problems. The detection and resolution of many non-determinism issues can hence be done locally at the stub level, which is, in our experience, far better than at the whole system level. The correctness of selection policies is further validated when testing the LOTOS specification (Section 7.2).

5 LOTOS Concepts

Although LOTOS has been used for the specification of telephony systems, by our team and others [2][3][9][15][22][23], overall its application has been limited, SDL being far more common [8][13][16]. For this reason, some LOTOS concepts will be explained by contrasting them with the corresponding SDL concepts. Although the two languages largely overlap in range of applicability, there are substantial differences. While SDL requires the identification of specific system components, specific directional messages between components, and specific system states, LOTOS can specify systems in abstract terms as partial orders of generic responsibilities, to be refined later into exchanges of directional messages. In particular, while SDL actions are messages that are sent from one component to another, LOTOS actions are events that are not necessarily directional and can involve simultaneously several system components. This synchronization concept is explained below. Such events cannot be implemented directly, and their implementation may require a protocol. They represent abstract conceptual behavioral units, which can be useful at certain stages of design. It can be seen then that in this sense UCM responsibilities correspond closely to LOTOS actions. This has important consequences: UCMs can be mapped into LOTOS fairly easily, whether components are identified or not [3]. Therefore, LOTOS may be more appropriate for formal specification and validation at the initial stages of design. SDL is more implementation-oriented and is more appropriate for the later stages of design [1].

Although LOTOS is mostly inspired by CCS [19], its interprocess communication mechanism is direct multi-way synchronization between components, as in CSP [11]. This is similar to SDL's remote procedure call, but it can occur between any number of components. Think of connecting a number of electrically charged wires together: this will establish a common charge; similarly, in LOTOS, a synchronization event among different processes will establish common values. Unlike SDL, LOTOS has no concept of built-in queues between components. If queues are needed, they must be explicitly specified.

Hereafter, a brief overview of LOTOS syntax and semantics is given. LOTOS represents the behavior of the system by using *actions* and *behavior expressions*. Actions represent basic behaviors of the system, at the targeted conceptual level, for example, `req` or `ring`. There is also an internal (invisible) action written as **`i`**. Three basic behavior expressions exist: **`stop`** (also called deadlock or unsuccessful termination), **`exit`** (successful termination) and process instantiation `P[G](V)`, where `P` is the name of a LOTOS process, `G` is a set of gate parameters, and `V` is a set of value parameters. Given behavior expressions B, B_1, B_2, etc. and actions a, a_1, a_2, etc., LOTOS operators can be used to construct more complex behavior expressions as follows:

`a; B`: action prefix operator	B_1 `[>` B_2: disable operator
B_1 `[]` B_2: choice operator	B_1 `||` B_2: full synchronization operator
B_1 `>>` B_2: enable operator	B_1 `|||` B_2: interleaving operator
`hide` `a` **`in`** `B`: internalizes (transforms to **i**) all offerings of actions `a` in `B`.	B_1 `|[`a_1, a_2, `...` a_n`]|` B_2: selective synchronization operator

LOTOS also includes a basic *Abstract Data Type* formalism, called ACT ONE, which is used to represent data abstractions (in this, LOTOS is similar to SDL). Boolean *value expressions* can be used to *guard* actions. Data can be associated with actions in two basic ways: `!`, meaning value offer, and `?`, meaning value query. These can be combined in actions, for example:

```
COM !p_1 !dialTone ?telephone:number
```

denotes an action where on gate `COM`, the current value of the variables `p_1` and `dialTone` are offered, and a value for `telephone` is queried simultaneously. Offers and queries are called *experiments*. Note that when no data is involved, actions are simply gate names. With data, actions are gate names with experiments, i.e. data offers and queries. Sequences of LOTOS actions are called *traces*. Traces should include only visible actions; however internal actions often are also shown, for completeness.

LOTOS operators make it possible to represent clearly telephony system structures in terms of agents that coexist independently (interleave), communicate (selective synchronization), follow each other's actions (enable), or can interrupt each other (disable). Internal system actions can be hidden. Several specification *styles* are possible in LOTOS [26]. Some styles are appropriate for representing abstractly system requirements, while others are appropriate for representing implementation structures.

6 Translation from Use Case Maps to LOTOS

The translation from UCMs to LOTOS corresponds to the formalization and mapping of an abstract, semiformal model into a formal and executable one, but still remaining at the design specification level. An analytic approach is taken for this translation, i.e. it is first done manually and then it is verified formally. In order to do this mapping, the UCM system is analyzed and missing information, confusions or errors are detected and corrected. Experience shows that several potential design errors can be caught at this stage.

6.1 Translation Guidelines

We have defined guidelines to construct LOTOS specifications from UCMs:

- *Start points* and *end points* are usually represented by LOTOS gates in the prototype. They can then be controlled and observed during the validation.
- UCM *responsibilities* are also represented as gates, sometimes with additional message exchanges (to ensure causality across components).
- LOTOS gates representing UCM responsibilities and channels that are not observable to users are hidden through the **`hide`** operator.
- UCM *components* are represented as processes synchronized on their shared channels/gates. The structure is specified mostly in a resource-oriented style [26], with multi-way synchronization (`|[...]|`) and interleaving (`|||`) operators.
- Containment of components is maintained. If a component, represented by a LOTOS process, contains sub-components, then the processes representing these are instantiated (and possibly defined) within the former process.

- If multiple *path segments* (possibly from different UCM scenarios) cross one component, they are integrated together in the same LOTOS process, often as alternatives.
- Elementary processes are specified mostly in a state-oriented style, with choice (`[]`) and action prefix (`;`) operators, and with guarded behaviors (`[...]->`).
- Abstract data types are used to represent databases, operations, and *conditions* (LOTOS guard expressions).
- Symmetry is enforced in synchronized actions: actions in one component/process must be mirrored in the other synchronized processes, unless locally hidden.
- Components with *stubs* have sub-processes, one for each stub. These processes specify the selection policy, i.e. the type of composition between the possible plug-ins.
- *Dynamic stubs* may have multiple sub-processes, one for each plug-in, whereas *static stubs* are refined directly by the process representing their only plug-in.
- Stub processes receive a list of entry/exit points as input and then output another such list upon termination.

6.2 Interprocess Synchronization

The translation process starts by analyzing the UCM model to determine an appropriate specification structure. During this analysis, the LOTOS designer makes choices concerning the mapping of agents, stubs, plug-ins, etc., according to the guidelines mentioned in Section 6.1. Since UCMs are used at a higher level of abstraction than LOTOS, this analysis gives the opportunity of inspecting the documentation and detecting missing parts, contradictions, or other problems.

Let us consider the following processes (where gate parameters within brackets represent the visible channels):

```
process DAGENT [ DA_to_PA, PA_to_DA ]: exit :=
      DA_to_PA; exit
endproc
process PAGENT [ PA_to_DA, DA_to_PA ]: exit :=
      DA_to_PA; exit
endproc
```

These can be combined to require synchronization over all their gates as follows:

```
DAGENT [ DA_to_PA, PA_to_DA ]
 |[ DA_to_PA, PA_to_DA ]|
PAGENT [ PA_to_DA, DA_to_PA ]
```

The synchronizing points between processes `DAGENT` and `PAGENT` are then all actions that involve the gates `DA_to_PA` and `PA_to_DA`. According to LOTOS semantics, if one of the processes wants to fire an action with gate `DA_to_PA` or `PA_to_DA`, the other process must also be willing to do this for the action to become executable. Synchronizing on an action also implies an agreement on the values of the parameters of the event, i.e. the experiments.

Combined with the interleaving operator `|||`, experiments may be used also to provide a more dynamic kind of synchronization. This subtlety is called *gate splitting* [4]. It involves including in actions experiments that play the role of addresses. For example, an action that starts by `a!1` (where `1` is a constant) will only synchronize with other actions that start in the same way, and, if `m` and `n` are value identifiers, an action that starts by `a!m` will only synchronize with other actions `a!n` if `m=n`. Let us consider that a DAGENT should communicate with two different PAGENTS. In a simplistic approach, one could consider a set of gates (one for each direction) for the communication with each of the two PAGENTs. This would produce an undesirable static structure, inappropriate for the dynamic nature of agent systems: notice that the set of gates of a DAGENT would have to be changed whenever a PAGENT is added or removed. Gate splitting makes it possible to have more dynamic structures. A typical application is that of a switch that may wish to synchronize with any phone, but one at a time. All the phone processes are interleaved (run independently of each other) while the switch is ready to synchronize with any of them:

```
switch[COM]
|[COM]|
(     phone[COM](p_1)
 ||| phone[COM](p_2)
 ||| ...
 ||| phone[COM](p_n) )
```

An experiment is included in an action to uniquely identify each phone such that the switch may selectively synchronize with a specific one. To query a telephone number from the process phone `p_1`, the switch could specify the action:

```
COM !p_1 !dialTone ?telephone:number
```

The previous definitions of DAGENT and PAGENT can be extended with gate splitting to a system containing three processes (e.g. two PAGENTs and one DAGENT). PAGENT processes are instantiated with a parameter that predefines their identity. The process definitions become:

```
process DAGENT [ DA_to_PA, PA_to_DA ]: exit :=
  DA_to_PA !PAgent_a !connect; exit
  []
  DA_to_PA !PAgent_b !disconnect; exit
endproc

process PAGENT [PA_to_DA, DA_to_PA](PAgent:PAgentID): exit :=
  DA_to_PA !PAgent ?message:Data; exit
endproc
```

And the system behavior is as follows:

```
DAGENT [ DA_to_PA, PA_to_DA ]
|[ DA_to_PA, PA_to_DA ]|
( PAGENT [ PA_to_DA, DA_to_PA ](PAgent_a of PAgent_ID)
  |||
  PAGENT [ PA_to_DA, DA_to_PA ](PAgent_b of PAgent_ID) )
```

Each instance of PAGENT outputs its own identifier (`PAgent_a` and `PAgent_b`, respectively) and queries a message of sort *Data*, on the same gate `DA_to_PA`. The DAGENT may either synchronize with PAGENT `PAgent_a` through the event `DA_to_PA !PAgent_a !connect` or with PAGENT `PAgent_b` through the event `DA_to_PA !PAgent_b !connect`. In all cases, the DAGENT offers both the identifier of the PAGENT and the message it wishes to send. For any action to be executed, one PAGENT needs to agree on the first experiment (the PAGENT identifier) and accept the second one (the message).

By using these techniques, we can have in our specification many instances of these processes, each instance created by a process of recursive instantiation, on demand.

Instances of DAGENT, PAGENT, and FAGENT are agents composed of several static and dynamic stubs. These stubs can in turn contain other stubs and so on. A static stub is represented as a single process. A dynamic stub is represented by the instantiation of a process of the same name. This process handles the instantiation (most of the time as an exclusive choice, depending on the selection policy) of one of the sub-processes, where each sub-process corresponds to a plug-in of the dynamic stub.

These choices result in a LOTOS specification that is close to the UCM model. It also makes the translation of features a straightforward technique, consisting in following the UCM model and replacing the plug-ins.

The features considered so far in this projects are: Outgoing Call Screening (OCS), Call Forward Always (CFA), Call Forward on Busy (CFB), Call Hold (CH), Recall (RC), Call Pickup (CP) and Call Transfer (CT). As mentioned above, they are described (with three other features not yet specified in LOTOS) in more than 100 UCMs. Their translation into LOTOS resulted in a specification of over 3,000 lines.

7 Validating the System with LOTOS

The LOTOS specification is executable and thus constitutes a prototype of the final system. This means that, for example, it is possible to execute the specification in order to see its response to possible actions at each state (step-by-step), to see whether the specification accepts/rejects certain scenarios, and to analyze a large number of scenarios by means of reachability analysis and model-checking tools. Often these three methods are used in this order to achieve different types of assurance on the quality of the specification. The main technique used in this project so far has been the second one (acceptance/rejection test scenarios), which is a form of specification-level testing [21].

7.1 UCM-Based Testing for LOTOS

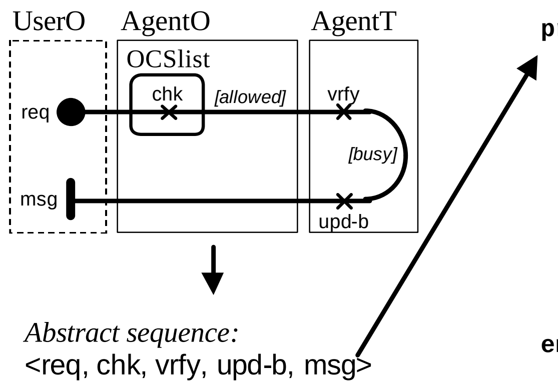
Test scenarios are manually extracted from the UCM model. Since the latter is a scenario-based model to start with, scenario extraction is carried out by simply following possible paths in the model. This works well for the derivation of basic system and individual feature scenarios because these rely directly on the specification. The sequence of activities that compose a given path is extracted from the UCM. That sequence is translated into a LOTOS process. The resulting scenario can be used for testing the LOTOS specification of the system in order to verify the consistency between the specification and the UCM, and to validate the requirements.

We can derive *abstract validation test sequences* from UCMs. For most realistic systems, the high (if finite) number of global states makes the generation of exhaustive test suites impossible. Hence, it becomes essential to select carefully a small and finite set of validation test cases. To do so, we base our strategy on the exploration of UCM paths, following a strategy similar to the white-box approaches used for sequential programs. Depending on the targeted functional coverage, we can choose to explore some paths, all combination of paths, some or all the temporal sequences resulting from concurrent paths, etc. We call *testing patterns* these selection strategies for UCMs. A number of testing patterns can be defined based on UCM constructs, such as:

- **Alternatives**: All results; All paths; All path combinations; All combinations of sub-conditions within a complex condition.
- **Concurrent**: One combination (a temporal sequence); Some combinations; All combinations.
- **Loops**: One iteration; At most two iterations; 0, 1, n, and n+1 iterations.

These patterns help to document test selection strategies related to the functional coverage of designs represented with UCMs. However, this list is not exhaustive as other UCM constructs exist and patterns based on combinations of constructs can also be defined.

As an example, suppose a test, whose goal is to check a scenario where a call is initiated, goes through OCS, but is refused because the terminating party is already busy. In terms of the UCMs in Figure 3, this corresponds to the path shown in Figure 5:



```
process OCStest[...]: noexit:=
  req !User1 !User2;
  u2a !User1 !Agent1 ! req !User2;
  chk !Agent1 ! allowed;
  a2a !Agent1 !Agent2 ! req;
  vrfy !Agent2 ! busy;
  upd_b !Agent2;
  a2a !Agent2 !Agent1  busy;
  a2u !Agent1 !User1 ! busy;
  msg !User1 !busy;
  success; stop
endproc
```

Possibly hidden

Figure 5 From UCM paths to LOTOS tests.

The only possible abstract sequence is shown in the figure. This sequence translates into the test `OCStest`. The content of this process may vary depending on architectural decisions taken during the generation of the LOTOS specification from the UCMs. For instance, local responsibilities (`chk`, `vrfy`, `upd-b`) and communication between users and agents (channels `u2a` and `a2u`) and between agents (`a2a`) may be hidden. The italicized actions would hence be removed from the test, leaving only the actions visible to the users (`req` and `msg`). Note also how gate splitting can help identify the users and agents involved in the various events and messages (as seen in Section 6.2). Various rejection test cases, whose purpose would be to improve the robustness and the correctness of the design, could also be defined for the same abstract sequence. Note also that conditions accumulated along a path (*[allowed]* and *[busy]*) must be satisfied. This helps defining the initial configuration of the system (instances, databases, etc.) for each test to pass successfully.

Through the use of tools such as LOLA [21], validation test suites consisting of dozens of test cases can be checked against large specifications within several minutes or even seconds. When the specification is non-deterministic, LOLA checks all possible outcomes, so test cases do not need to be applied more than once to achieve a good coverage. Functional coverage criteria to see if all UCM paths are covered are also applied [3]. Although this testing technique appears to be limited (tests and specifications being both derived from the same UCMs), it proves to be a practically feasible approach to validation and verification. The main reason is that functional tests are much closer to the requirements and simpler to validate than a specification that integrates all features.

7.2 Verification and Interaction Detection

The purpose of scenarios derived from the UCM model is twofold: to verify the LOTOS specification against the UCMs (and hence to validate the requirements) and to test the system for interactions. Furthermore, scenarios are derived with three types of tests in mind:

- ***Basic System Properties***: the basic properties that the system must possess are tested (i.e. basic call without features). The scenarios involve simple tests such as calling another party that is idle and calling another party that is busy.

- ***Individual Features Properties***: the features taken individually are tested. Each feature needs to be working properly by itself before being integrated with other features for detecting interactions.

- ***Feature Interactions***: undesirable feature interactions are tested. These test scenarios aim to reveal problems that a given feature might inflict upon other features.

Extracting the first two types of scenarios is fairly straightforward, and their goal is to verify that the LOTOS specification is consistent with the UCMs. Extracting scenarios for feature interaction detection involves understanding the properties of the features and their behaviors in relation to each other and to the system. Some features can interact with many while others can interact only with a few. The strategy used consists in analyzing the features and to target those pairs of features that appear to present the risk of interaction [17]. This analysis requires a good knowledge of the system and of its features. It was carried out in close collaboration between the design team and the formal specification team.

Each feature interaction scenario has a precondition (or initial configuration), that describes a desired state of affairs before the scenario is tested. An example scenario is (see also Section 8):

User **B** is registered to *Outgoing Call Screening* and screens calls to user **C**. **A** calls **B** and talks with her. **A** puts **B** on hold, then calls **C**. **A** talks to **C**, then **A** transfers **B** (held) to **C** (talking). Transfer fails because of **B**'s **OCS**, **B** gets a fast busy tone, **C** is back with **A**.

UCMs can help pruning out scenarios that are unlikely to cause interactions, hence UCMs can guide the selection of feature interaction test cases. For example, many potential interactions are avoided between features that are allowed to proceed independently in different stubs in a UCM. To a certain extent, these stubs encapsulate the features from their environment. Although UCM-based selection of interaction test cases has been little used, this idea seems promising and is currently the topic of further study[20].

Many other problems were detected and solved while specifying the LOTOS prototype. Beyond this, the testing phase helped detecting more subtle problems, many of which resulted from unclear expected interactions between features and from race conditions. So far, we have developed and checked 20 test cases for the basic system properties, 15 test cases for the individual properties of our 7 features, and 20 tests for the detection of interaction between pairs of features. However, we observed that the initial set of UCMs was well built and that real problems were scarce. This increased our confidence in the effectiveness of the UCM-based design approach. Nevertheless, it is expected that, as features are added, the human ability to prevent and detect interaction by design and inspection will decrease, and thus our testing method will prove more valuable.

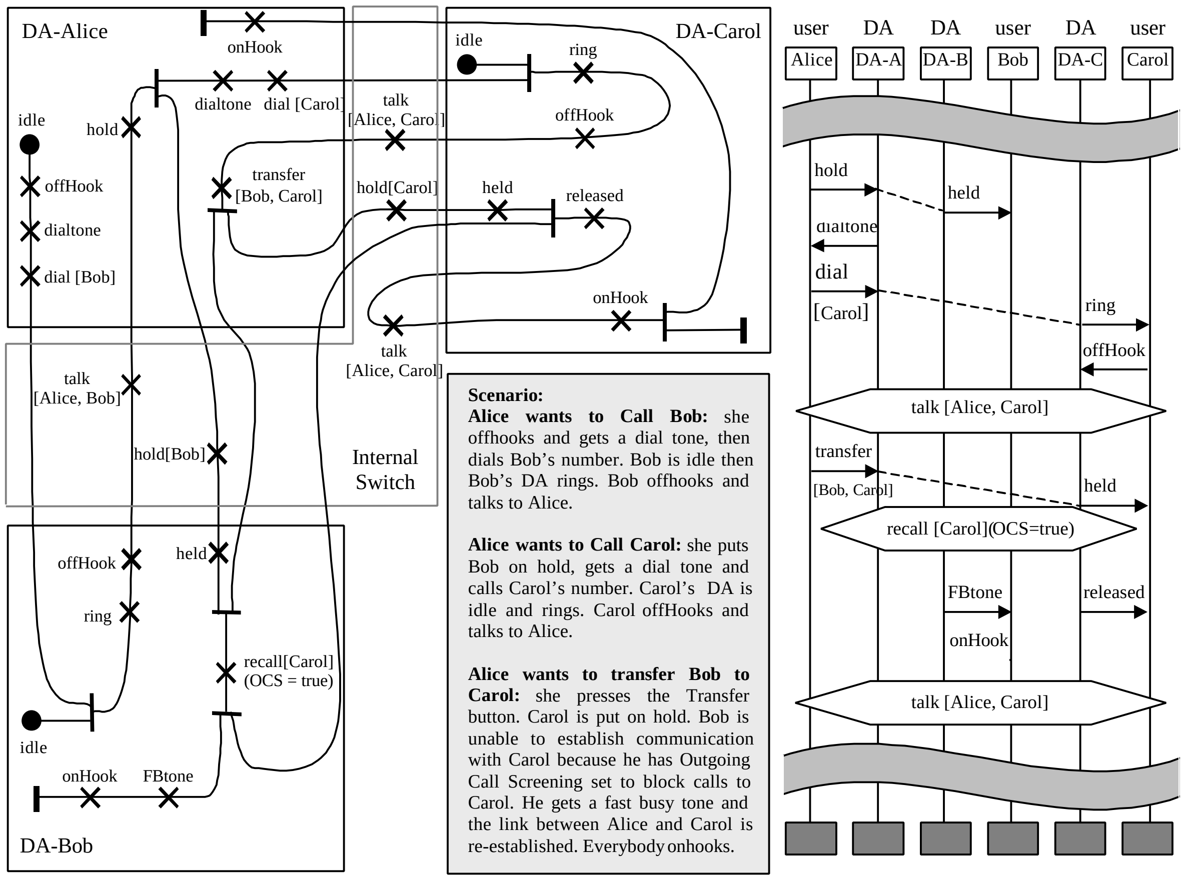
The detection of interactions is not limited to UCM-based testing. Having a LOTOS model enables the use of many complementary techniques, based on several available LOTOS tools. These include goal-oriented execution [15] and the use of observers [9]. Also, we are planning to start experimenting with the CADP tool, which allows extensive reachability analysis and model checking [10].

8 Translation of LOTOS Traces to MSCs , and Animation Tools

The raw results of the testing process are given in the form of LOTOS traces, i.e. commented sequences of LOTOS actions. In order to make these understandable to users not familiar with LOTOS syntax, some tools are being developed.

The first tool is a translator from LOTOS traces into Message Sequence Charts (MSCs). The main difficulty to be addressed here is that LOTOS actions are non-directional and result from multi-way synchronization, while arrows in MSCs represent messages generated by an entity and directed to another entity. This difficulty is overcome by associating directions to gates and by indicating sender and receiver entities (i.e. DAGENT, PAGENT) as parts of the gate name (see examples above). User-prepared configuration tables define the structure of the system and of the messages for the tool. Each action typically includes (in a fixed order that is described in the configuration tables) the following list of experiments: the identifiers of the instances of the sender and receiver entities, the message type, and the parameters of the message. The tool constructs the MSCs by drawing the vertical lines corresponding to the components specified in the configuration tables and then by drawing the arrows representing messages. In fact, LOTOS traces are translated into MSC's machine processable representation (.mpr) [14]. The MSCs are then printed using tools such as Telelogic Tau. Figure 6 shows a flat UCM that illustrates the scenario given in Section 7.2, and a MSC corresponding to part of this UCM. Call establishment and disconnection are not shown in this MSC, also some internal system actions are represented as states. Note that in this example the UCM and the MSC are at the same level of abstraction: in reality, this is seldom the case (Section 2.1). This example is for illustration purposes, and is not part of the Mitel design documentation, which is higher-level, is better structured, and could not be detailed here due to space and confidentiality constraints.

A tool to translate the MSC .mpr notation into LOTOS traces was also developed. This tool is used in order to check whether a LOTOS specification recognizes scenarios produced by hand or by other tools, e.g. SDL tools.



Scenario:
Alice wants to Call Bob: she offhooks and gets a dial tone, then dials Bob's number. Bob is idle then Bob's DA rings. Bob offhooks and talks to Alice.

Alice wants to Call Carol: she puts Bob on hold, gets a dial tone and calls Carol's number. Carol's DA is idle and rings. Carol offHooks and talks to Alice.

Alice wants to transfer Bob to Carol: she presses the Transfer button. Carol is put on hold. Bob is unable to establish communication with Carol because he has Outgoing Call Screening set to block calls to Carol. He gets a fast busy tone and the link between Alice and Carol is re-established. Everybody onhooks.

Figure 6 Partial UCM and partial MSC for the target system.

Another tool that has been developed is a graphic animator for given traces. It shows the system in the form of a conventional structural diagram with boxes and connectors. The user causes message firings by flicks of the mouse. When a message is sent from one component to another, the animator shows the movement of an arrow. Boxes are associated with expressive icons that change depending on the message received, i.e. showing telephones going on-hook, off-hook, and so on.

Prototypes for the tools described so far have been completed. A more advanced graphic animator, a real simulator that enables to see system responses to user actions such as composing numbers, lifting the handset, etc., as icons on the screen (inspired by[16]), is in an advanced stage of design.

The use of such tools requires that LOTOS specifications be written according to specific guidelines.

9 Conclusions and Future Work

The main result of this project is to have shown, to the satisfaction of the industrial partner, that a design methodology for telephony switches based on the combined use of UCM and LOTOS specification methods is feasible and effective in practice. In a companion project (FAST SPEC-TO-TEST), this methodology is being extended to the generation of functional test cases, in TTCN, for testing the implementation of the switches.

A number of research directions suggest themselves in order to complete the methodology. For one, the translation from UCM to LOTOS should be helped. This could be done by user-assisted tools that would construct skeleton specifications corresponding to certain architectures. Alternatively, an intermediate language could be defined. This language would look like UCM but semantically could be closely related to LOTOS. A longer range goal might be defining a new formal language more closely corresponding to UCMs.

The extension of the analysis to a much larger number of features will present new problems. Feature databases, with the capability of configuring systems with different sets of features, will have to be designed. With more features, it will also become critical to have automated tools to propose scenarios to be tested. Nakamura *et al.* [20] address this issue with a filtering approach that extracts interaction-prone scenarios based on the configuration of UCM stubs and plug-ins.

Acknowledgments. This research was funded in part by Communication and Information Technology Ontario, by Mitel and by the Natural Sciences and Engineering Research Council of Canada. D. Pinard contributed to the development of the agent concept and, together with N. Balaba and T. Ware, she authored at Mitel many of the UCMs that were used. Participation of the authors in the FAST SPEC-TO-TEST project, led by R.L. Probert, generated much useful discussion.

References

[1] Amyot,D., Andrade, R., Logrippo, L., Sincennes,J., and Yi, Z. "Formal methods for mobility standards". In: *Proc. of the 1999 IEEE Emerging Technologies Symposium on Wireless Communications and Systems*, Richardson, TX, USA, 1999. http://www.UseCaseMaps.org/pub/ets99.pdf

[2] Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L. "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, 44-53. http://www.UseCaseMaps.org/pub/re99.pdf

[3] Amyot, D., and Logrippo, L. "Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System". To appear in *Computer Communications*, 23(8). http://www.UseCaseMaps.org/pub/cc99.pdf

[4] Bolognesi, T., De Frutos, D., Langerak, R., Latella, D. "Correctness Preserving Transformations for the Early Phases of Software Development". In: Bolognesi, T., van de Lagemaat, J., and Visser, C., *LOTOSPhere: Software Development with LOTOS*. Kluwer, 1995.

[5] Buhr, R.J.A. and Casselman, R.S. *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, 1996. http://www.UseCaseMaps.org/pub/UCM_book95.pdf

[6] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. "Feature-Interaction Visualization and Resolution in an Agent Environment". In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 135-149. http://www.UseCaseMaps.org/pub/fiw98.pdf

[7] Buhr, R.J.A. "Use Case Maps as Architectural Entities for Complex Systems". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998, 1131-1155. http://www.UseCaseMaps.org/pub/tse98final.pdf

[8] Ellesberger, J., Hogrefe, D., and Sarma, A. *SDL - Formal Object-Oriented Language for Communicating Systems.* Prentice-Hall, 1997.

[9] Fu, Q., Harnois, P., Logrippo, L., Sincennes, J. "Feature Interaction Detection: A LOTOS-based approach". To appear in *Computer Networks.*

[10] Garavel, H."OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing". In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, 1998.

[11] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] ISO, Information Processing Systems, Open Systems Interconnection, *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, IS 8807, Geneva, Switzerland, 1989 (E. Brinksma, Ed.).

[13] ITU, *Recommendation Z.100, Specification and Description Language (SDL)*. Geneva, Switzerland, 1994.

[14] ITU, *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU, Geneva, Switzerland, 1996

[15] Kamoun, J. and Logrippo, L. "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model". In K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, Sept. 1998. IOS Press, 172-186.

[16] Kimbler, K., Hagenfeldt, C.-H., Widell, N., Ellsberger, J., and Bergman, G. "SDL Framework for Prototyping and Validation of IN Services". In: R. Dssouli, G.V. Bochmann, Y. Lahav (Eds.). *SDL'99*. Elsevier, 1999, 19-32.

[17] Kimbler, K., and Søbirk, D. "Use Case Driven Analysis of Feature Interactions". In: L.G. Bouma and H. Velthuijsen (Eds.), *Feature Interactions in Telecommunications Systems*, IOS Press, 1994, 167-177.

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

[19] Milner, R. *Communication and Concurrency*. Prentice-Hall, 1989.

[20] Nakamura, M., Kikuno, T., Hassine, J., and Logrippo, L., "Feature Interaction Filtering with Use Case Maps at Requirements Stage". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, May 2000.

[21] Pavón, S. and Llamas, M., "The Testing Functionalities of LOLA". In: J. Quemada, J.A. Mañas, and E. Vázquez (Eds), *Formal Description Techniques, III*, IFIP/North-Holland, 1991, 559-562.

[22] Thomas, M. “Modelling and Analysing User Views of Telecommunications Services”. In: Dini, P., Boutaba, R., and Logrippo, L. (Eds.) *Feature Interactions in Telecommunications and Distributed Systems IV*, IOS Press, 1997.

[23] Turner, K.J. “An architectural description of intelligent network features and their interactions”. In: *Computer Networks and ISDN Systems*, vol. 30, no. 15, September 1998, 1389-1419.

[24] UML Revision Task Force. *OMG Unified Modeling Language Specification*, version 1.3, June 1999. http://uml.shl.com/artifacts.htm

[25] *Use Case Maps Web Page* and *UCM User Group*, 1999, http://www.UseCaseMaps.org

[26] Vissers, C.A., Scollo, G., van Sinderen, M., Brinksma, E. “Specification Styles in Distributed Systems Design and Verification”. In: *Theoretical Computer Science ’89*, 179-206.

[27] Weiss, M., Gray, T., and Diaz, A. “Experiences with a Service Environment for Distributed Multimedia Applications”. In: Dini, P., Boutaba, R., and Logrippo, L. (Eds.) *Feature Interactions in Telecommunications and Distributed Systems IV*, IOS Press, 1997.