Use Case Maps for the Capture and Validation of Distributed Systems Requirements

D. Amyot and L. Logrippo
School of Information Technology and Engineering, University of Ottawa, Canada
{damyot | luigi}@site.uottawa.ca

R.J.A. Buhr
Department of Systems and Computer Engineering, Carleton University, Canada
buhr@sce.carleton.ca

T. Gray
Mitel Corporation
Ottawa, Canada
Tom_Gray@mitel.com

Abstract

Functional scenarios describing system views, uses, or services are a common way of capturing requirements of distributed systems. However, integrating individual scenarios in different ways may result in different kinds of unexpected or undesirable interactions. In this paper, we present an innovative approach based on the combined use of two notations. The first one is a recent visual notation for causal scenarios called Use Case Maps (UCMs), which is used to capture and integrate the requirements. Integrating UCMs together helps avoiding many interactions before any prototype is generated. The second notation is the formal specification language LOTOS*. UCM scenarios are translated into high-level* LOTOS *specifications, which can be used to validate the requirements formally through numerous techniques, including functional testing based on UCMs.* LOTOS *possesses powerful testing concepts and tools that we use for the detection of remaining undesirable interactions. To illustrate these concepts, we use a simple connection example and results from the capture and the validation of several telephony features from the First Feature Interaction Contest.*

1. Introduction

Scenarios describing system views, uses, or services are a common way of capturing requirements of distributed systems. Among other things, they help to uncover hidden requirements and trade-offs, to verify and validate requirements, and to integrate analysis of functional and nonfunctional requirements. In distributed systems, scenarios offer the possibility of expressing functionalities that span many system components. In the telecommunication industry in particular, customer services, also called *features*, are often represented as scenarios in the early stages of the development cycle.

Undesirable interactions between features still represent nowadays a complex problem that telecommunication systems designers must face [5][10][20], 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 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 [8]. 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 aims to facilitate the creation of an avoidant design, and to detect remaining interactions at design time with the help of an executable prototype. Avoidance of interactions between operational requirements is achieved through the visual integration of scenarios expressed with the *Use Case Map* (UCM) notation [7][9]. Detection is done by using a process algebra, the *Language Of Temporal Ordering Specification* (LOTOS) [14] in our case, and formal validation and verification techniques.

LOTOS has been used for years for the specification and validation of telephony systems [11] and for the detection of feature interactions [12][18]. One of the challenges in using this language consists in writing the first system specification from informal requirements. However, once a specification is available, rigorous methods can be used to validate the specification and the requirements.

Use cases, as defined by Jacobson [17], have been utilized for the analysis of interactions in telephony systems [19]. More recently, UCMs, as defined by Buhr, have been used to tackle the problems of feature interactions and resolution of conflicts in multi-agent systems [8]. The UCM notation helps designers with the visualization of problematic situations and their avoidance at a high level of abstraction. However, UCMs do not support formal validation and verification directly.

An approach where UCMs are transformed into LOTOS specifications and test cases has been applied to several examples of distributed systems (a Group Communication

Server [2] and the mobile telephony protocol GPRS [3]). Capturing requirements with UCMs provides much help in writing the first global specification, allowing it to be validated against the requirements through the UCMs, long before any implementation is produced.

With such knowledge and experience available, a methodology that would use the best characteristics 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 interactions in distributed systems seems like a natural evolution. Herein, we use such an approach (Section 2), and we pay a special attention to UCMs for the reader unfamiliar to this notation. Throughout the paper, we illustrate this methodology using a very simple connection example, supported with some results and lessons from the specification and validation of a set of features described in the First Feature Interaction Contest [13]. In Section 3, we present UCMs for selected features. A contribution of this paper is that the integration of the scenarios is first done at the UCM level, and not solely at the LOTOS level as in [2] and [3]. We discuss the synthesis and the validation of the LOTOS specification in the following section (Section 4). When integrating UCM scenarios together, many interactions, which represent the bulk of the problems addressed by many detection techniques, can be avoided (Section 3.4). However, for the remaining undesirable interactions, we use traditional LOTOS techniques and tools (Section 4.3). We discuss some aspects of this methodology in Section 5 and then we 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 iterative and incremental approaches that intend to bridge the gap between (informal) requirements and an abstract system design, in the design phase where a tentative distribution of system behaviours over a topology of components (structure) is being introduced.

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 [2][3]. 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 abstract design.

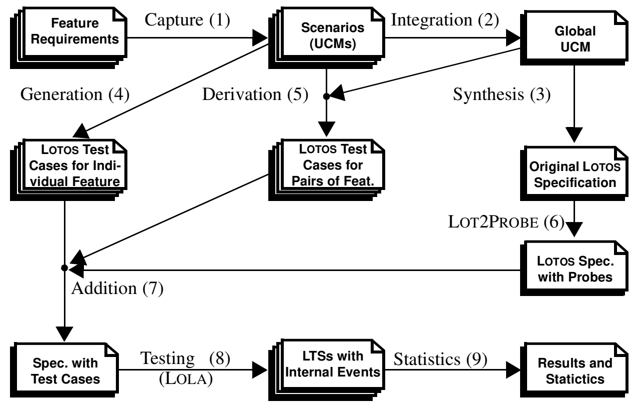


Figure 1. Scenario-based approach

The starting point is a collection of individual features, each of which is captured as a Use Case Map (1). This phase is often called *scenario elicitation* [19]. The responsibilities in the UCMs are bound to components in the underlying structure. UCMs can then be integrated together to produce a global UCM that covers all cases (2). Sequential, alternative, and parallel composition are used as integration operators, as well as more subtle abstraction and composition mechanisms that make use of stubs and plug-ins. The UCM notation allows the generation and evaluation of architectural ways of avoiding undesirable interactions before any prototype is developed. Once the global UCM is available, it can be used to synthesize a LOTOS specification, which becomes the executable prototype enabling formal validation (3).

Concurrently with these steps, validation test cases can be generated from the individual scenarios (4) to ensure that the specification conforms to each feature, when only one is active at a time. We can create further test cases, built on top of the tests for individual features, 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.

Probes can be inserted in the specification to ensure that the whole specification has been covered by the test suite (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 automatically 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, this feature has been incorrectly specified according to the global UCM, or this UCM does not conform to the requirements.

- At least one test case from the set for pairs of features 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. To some extent, the specification conforms to the UCMs (and to the functional requirements), 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.

2.2. Use Case Maps

We utilize the visual notation Use Case Maps (UCMs) for capturing the requirements of reactive and distributed systems. UCMs use behaviour as a concrete, first-class architectural concept. They describe scenarios in terms of *causal relationships* between *responsibilities*. UCMs usually emphasize the most relevant, interesting, and critical functionalities of the system. They can have internal activities as well as external ones. UCMs are abstract (generic), and could include multiple traces. With UCM, scenarios are expressed above the level of messages exchanged between components, hence they are not necessarily bound to a specific underlying structure. UCMs provide a path-centric view of system functionalities and improve the level of reusability of scenarios. Finally, UCMs can handle both acceptance and rejection scenarios, a useful property for the definition of test cases.

The following example illustrates some of the most important concepts of UCMs. For a detailed description of the notation, the reader should refer to [7] and [9].

2.2.1. Overview of the notation with a simple example

Figure 2(d) shows a simple UCM where a user (**U$_1$**) attempts to establish a connection with another user (**U$_2$**) through some network. **U$_1$** first sends a connection request (**req**) to the network. The latter verifies (**vrfy**) whether or not the called party is idle. If she is, then there will be some status update (**idle**) and a ring signal (**ring**) will be activated on **U$_2$**'s side. Otherwise, the network status will be updated differently (**busy**) and a message stating that **U$_2$** is not available (**msg**) will be sent back to **U$_1$**.

A scenario starts with a triggering event or a pre-condition (filled circle labeled **req**) and ends with one or more resulting events or post-conditions (bars), in our case **ring** or **msg**. We call *route* a path that links a cause to an effect. Intermediate responsibilities (**vrfy**, **idle**, **busy**) have been activated along the way. Think of responsibilities as tasks or functions to be performed, or events to occur. In this example, the activities are allocated to abstract components (boxes **U$_1$**, **U$_2$**, and **Network**), which could be seen as objects, processes, agents, databases, or even roles or persons. We call such superposition a *bound map.*

The notation allows for alternative paths (the fork in the figure), concurrent paths, exception paths, timers, stubs/plugins, and synchronous or asynchronous interactions between paths.

The construction of a UCM can be done in many ways. For example, one may start by identifying the responsibilities (Figure 2a), although not necessarily with diagrams like this one. They can then be allocated to scenarios (Figure 2b) or to components (Figure 2c). Components can be discovered along the way. Eventually, the two views are merged to form a bound map (Figure 2d).

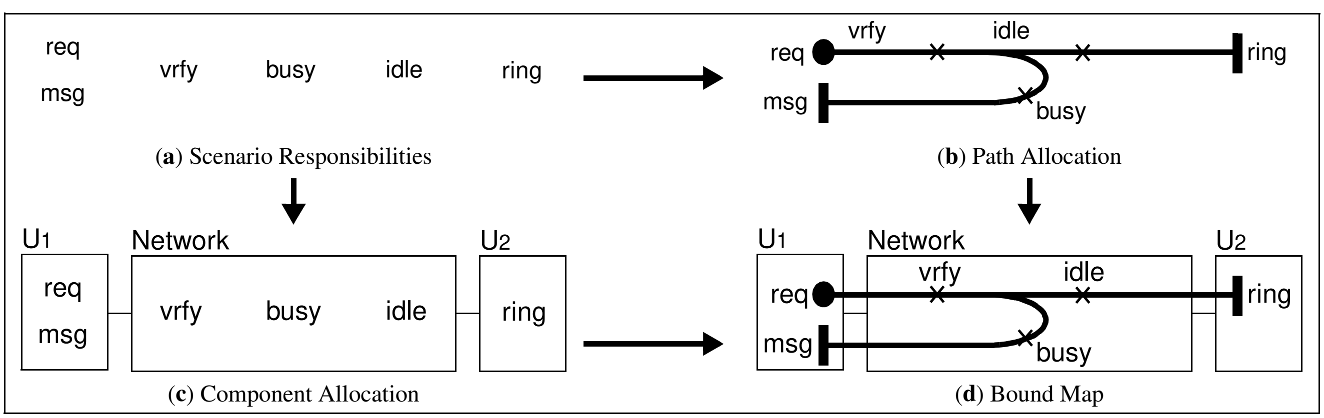


Figure 2. Use Case Maps construction

2.2.2. Evaluation of structures and refinement with message exchanges

The notation supports the reuse of scenarios when the underlying structure is modified or refined. During the service specification or design phases, different potential structures could undergo some evaluation (a.k.a. architectural reasoning). Scenarios described in terms of wired components, such as Message Sequence Charts (MSCs) [16] or UML interaction diagrams, would need to be rebuilt as soon as there is a change in the underlying structure, because the functionalities are tightly bound the structure. UCMs require simpler modifications, consisting only of a new binding between the responsibilities and the components.

A causal relationship, such as the one described by the route <**req**, **vrfy**, **idle**, **ring**> in Figure 2b, can be refined in many ways in terms of exchanges of messages, depending on the components structure, on the availability of communication channels, and on the chosen protocols. In this paper, message exchanges are described by means of MSCs, a standardized notation where vertical lines represent communicating parties, horizontal arrows represent messages, and time increases from top to bottom. Many MSCs could be valid according to a UCM, as long as the intended causal relationships between the responsibilities are satisfied. For instance, several communication channels (lines) are defined between the components of Figure 3(a). They constrain the sequences of messages allowed for the implementation of the causal relations in the scenario.

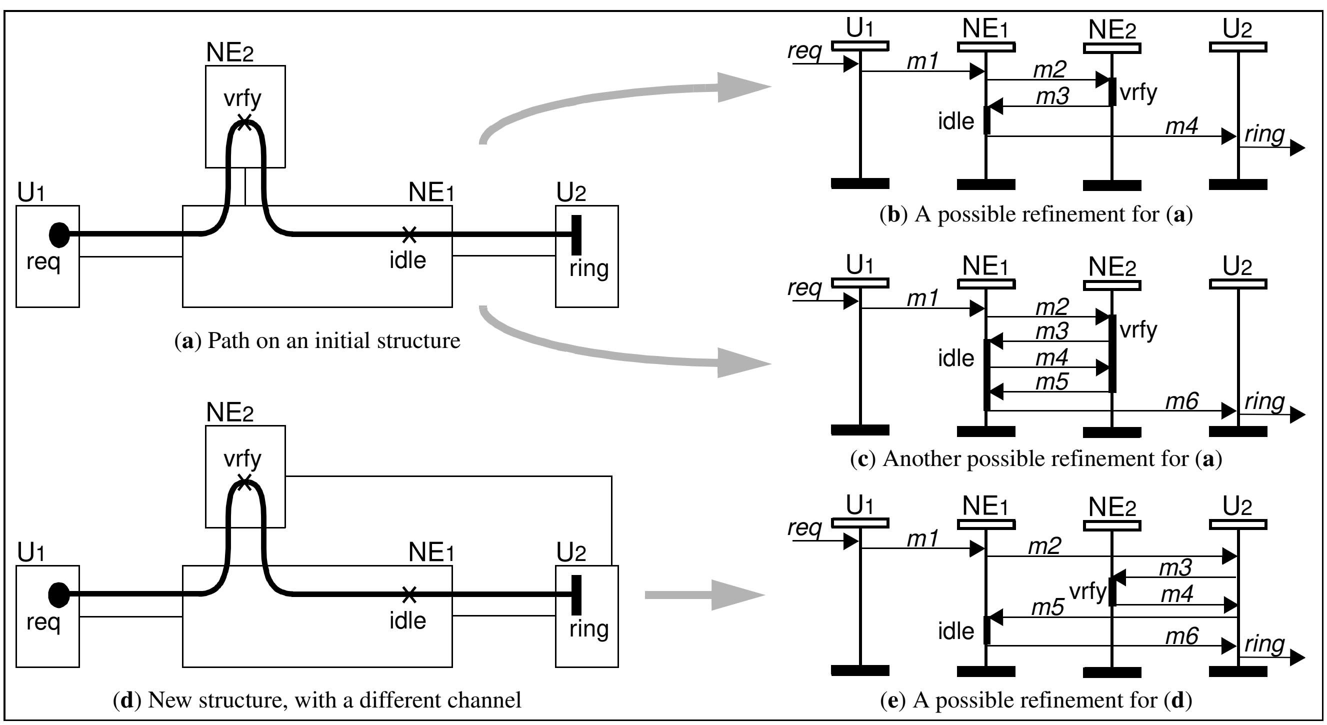
Figure 3(b) presents a MSC where the exchange of necessary messages is minimal. Notice that **U_1** is not allowed to send messages directly to **NE_2**, but messages can be forwarded through **NE_1**. Figure 3(c) illustrates a case where more complex protocols are used between **NE_1** and **NE_2**, and where many messages (perhaps some sort of negotiation) are required for the implementation of the causal relationship between **vrfy** and **idle**.

The structure in Figure 3(d) sees one of its channels in a place different from that of Figure 3(a). The two MSCs discussed previously become invalid because the constraints on the communication channels are not respected anymore (**NE_1** and **NE_2** are no longer allowed to communicate directly). Figure 3(e) suggests a possible refinement for the new structure. Again, we observe the impact of an early emphasis on message exchanges, something that is avoided at the UCM level.

3. Capturing features with Use Case Maps

3.1. Overview of the features

The first FI Detection Contest, organized by Griffeth *et al.* in [13], describes twelve features whose choice has been dictated by the need for them to interact. The network is modeled as a collection of black boxes communicating with each other via defined interfaces. Definitions for POTS and the features are given as informal requirements (in English) and as Chisel diagrams [1].



(**a**) Path on an initial structure

(**b**) A possible refinement for (**a**)

(**c**) Another possible refinement for (**a**)

(**d**) New structure, with a different channel

(**e**) A possible refinement for (**d**)

Figure 3. Causal scenarios and exchanges of messages

Since the focus of our methodology is on the capture of informal requirements, we consider the given Chisel diagrams as such. The interested reader can find further information on the transition from Chisel to UCMs in [4].

In this paper, we focus on only four of the contest features, with a special attention to the first one:

- *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 (PIN).
- *Calling Number Delivery* (CND): allows the called telephone to receive a calling party's Directory Number and the date and time.
- *IN Freephone Billing* (INFB): allows the subscriber to pay for incoming calls.
- *Terminating Call Screening* (TCS): restricts incoming calls from lines that appear on a screening list.

3.2. UCM capture from the requirements

This section discusses step (1) in Figure 1. Figure 4 shows a partial UCM for the INTL feature. Components like the Switch, the users (Originator role and Terminator role), the Service Control Point (SCP), and the Operating System (OS) are all described in the original requirements. Some events in the requirements 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 "**askForPIN**"), and others remain events that the user can trigger (like **off-hook**). Conditions (such as *RestrictedTime*) are italicized, responsibilities are marked with a cross, and event names are associated with start and end points. Responsibilities are bound to their respective network components (boxes). Obviously, some responsibilities will be refined as events or messages between components at a lower level of abstraction, but UCMs defer this kind of decision to a next refinement stage, possibly using another and more appropriate notation.

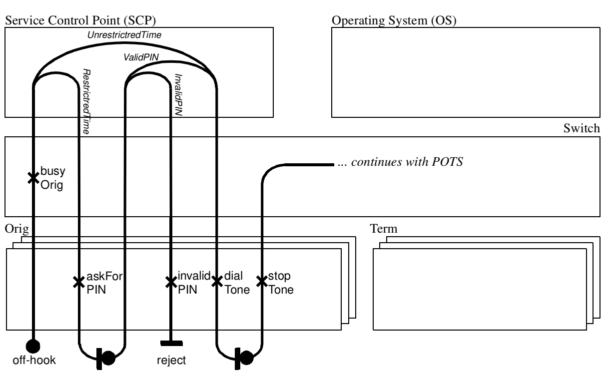


Figure 4. Partial UCM for INTL

Similar scenarios were defined for POTS and for all of our features. The next section discusses their integration in a single map.

3.3. 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 labeled incorrectly, omitted, at different levels of abstraction, or in different orders become hard to integrate. 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. Many such design decisions can be made at this level.

The following root map results from the integration of the twelve features of the contest. This integration was done with the *UCM Navigator* tool [21], a UCM editor developed in our research group. The *root map* (Figure 5) 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 submaps called *plugins*. 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 can also contain their own stubs, recursively.

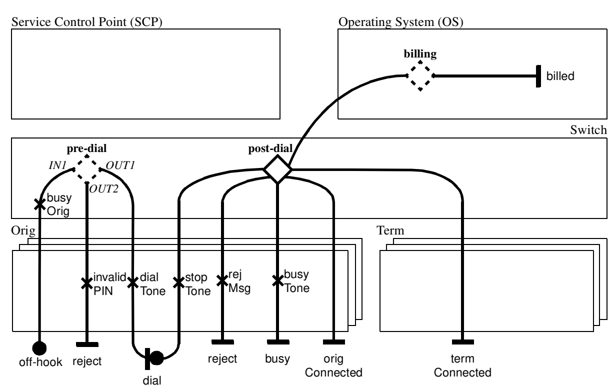


Figure 5. Root map for global UCM

One constructs a complete scenario by selecting appropriate plugins for the stubs. Plugins are bound to stubs by associating the entry and exit points of the stub with the start and end points of the plugin map. For example, the first stub in the root map, **pre-dial**, has one entry point (*IN1*), and two exit points (*OUT1*, *OUT2*). The **pre-dial** stub has two plugins, on of which (INTL) is illustrated in Figure 6. The binding of the INTL plugin is {(**INTL**, *IN1*), (**goDial**, *OUT1*), (**goReject**, *OUT2*)}.

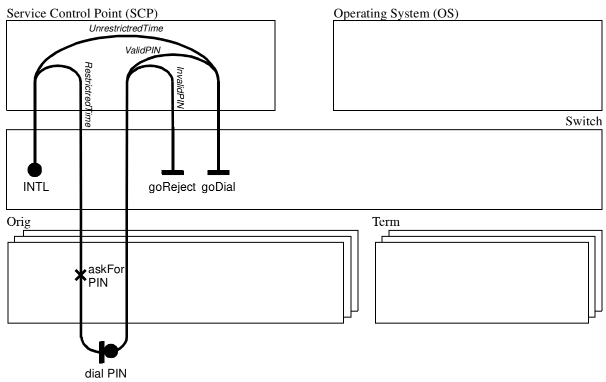


Figure 6. INTL plugin for pre-dial stub in root map

The other (default) plugin for the **pre-dial** stub is an empty path that links *IN1* to *OUT1*. The preconditions attached to these two plugins are disjoint, 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 (**dialTone**), and it overrides POTS according to the intent of INTL.

When an originator 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 4. Due to space limitation, we can only describe these stubs in the following way. The **post-dial** static stub contains only one default plugin in which two other stubs are composed in sequence. The **process-call** stub contains plugins for TCS, INFB, and default behaviour, while the **display** stub contains a CND plugin that can override the default plugin.

3.4. Avoiding feature interactions

We claim that an integration of scenarios at the level of UCMs helps to avoid many 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 in **process-call**) are still possible, depending on the composition/decision mechanism used within the stub. Maps with stubs show how localized the impact of a feature can be. 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 disjoint and complete preconditions (to avoid non-determinism and unspecified behaviour), or that priorities need to be established. Hence, the design decisions are simpler. The integration then becomes a useful step in a design process that includes UCMs. The UCM notation helps us to reason about architectures and behaviour in order to create systems in which undesirable interactions are avoided early in the development cycle, rather than merely detected at a later stage.

4. Formal specification and validation

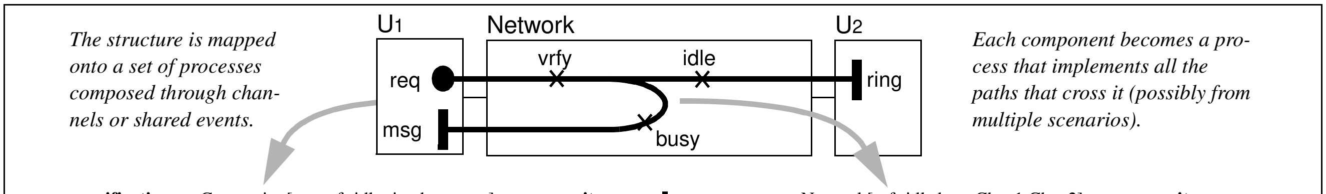
4.1. LOTOS and the synthesis & validation approach in a nutshell

LOTOS is an algebraic specification language standardized by ISO [14]. In LOTOS, the specifier describes a system by defining the temporal relations along the actions that constitute the system's externally observable behaviour. Data abstractions are specified with *Abstract Data Types*.

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 behaviour and structure in a unique executable model. LOTOS models allow the use of many tool-supported validation and verification techniques such as simulation, testing, expansion, equivalence checking, model checking, and goal-oriented execution.

4.1.1. Synthesis of specifications from UCMs

The synthesis of LOTOS specifications, illustrated by our example scenario in Figure 7, allows for the rapid generation of prototypes that represent 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 (Figure 7b). The archi-




```
specification Connection[req,vrfy,idle,ring,busy,msg] : noexit
(* Abstract Data Types *)
behaviour
  hide Chan1, Chan2 in
    ( U1[req,msg,Chan1] ||| U2[ring,Chan2]
    |[ Chan1, Chan2 ]|
    Network[vrfy,idle,busy,Chan1,Chan2]
where
  (* Component processes ... *)
endspec (* Connection *)
```

(**a**) Structure of components

```
process Network[vrfy,idle,busy,Chan1,Chan2] : noexit :=
  Chan1 !U1ToNetwork ! m1; vrfy;
  (
    idle; Chan2 !NetworkToU2 ! m2;
    Network[vrfy,idle,busy,Chan1,Chan2]
    []
    busy; Chan1 !NetworkToU1 ! m3;
    Network[vrfy,idle,busy,Chan1,Chan2]
  )
endproc (* Network *)
```

(**b**) Component behaviour

Figure 7. Synthesis of a LOTOS specification from a UCM

tecture itself is converted to a structure (Figure 7a) where the processes are composed together through shared communication *channels* (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.

For the synthesis process to become automatable, message exchanges, together with appropriate data types, compositions of plugins, and several other detailed design decisions, would need to be formalized at the UCM level. As we consider UCMs to be above such details, the synthesis remains a manual process, and the necessary decisions are therefore taken at the level of the formal specification.

4.1.2. LOTOS testing from UCMs using LOLA

Among all the validation and verification techniques available for LOTOS, we favored functionality-based testing for the validation of the features and the detection of interactions. Simulation is not sufficient because too many global sequences of events are often possible in the system. Since our requirements are expressed mostly operationally, UCMs and test cases are easier to extract than properties, so model checking is hardly usable at this point.

LOLA (LOtos LAboratory) is a state exploration tool with application in simulation and testing of LOTOS specifications [22]. LOLA analyzes 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.

We derive validation test cases 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 select carefully a small and finite set of validation test cases. To do so, we base our strategy on the exploration of UCM paths, similarly to white-box approaches used for sequential programs. Depending on the targeted coverage, 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. In our simple scenario (Figure 7), each of the two routes becomes an abstract sequence that is translated into LOTOS test processes (while considering the observable messages and data types defined during the synthesis).

4.2. Application to the global UCM

4.2.1. Synthesis

The current section relates to step (3) in Figure 1. According to the generic strategy illustrated in Section 4.1.1, the following are some of the basic concepts used to synthesize our LOTOS specification from the global map:

- Components are implemented as processes synchronized on their shared channels/gates.
- Hidden gates are used for responsibilities and messages that are not observable by the user.
- Path segments in one component are integrated together, often as alternatives.
- UCM activities are implemented as gates or as messages exchanges.
- Abstract data types 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).

Several additional rules were defined 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.
- Stub processes receive a list of entry/exit points as input and then output another such list upon termination.

These concepts have been used throughout the construction of the specification. The resulting specification is composed of 39 data types (750 lines of LOTOS code) and 13 process definitions (800 lines).

4.2.2. Test cases for POTS and individual features

Test cases generated from POTS and individual features (step (4) in Figure 1) check that each feature acts properly when being the only one active, and that POTS acts properly in the absence of any active feature. Often, more than one test case will be required to cover one requirement, because initial states and conditions are necessary. POTS has only one *precondition*: whether or not the terminator side is busy. Hence, two test processes, one for each initial configuration, are necessary.

We defined ten test processes for the coverage of INTL, CND, INFB and TCS (600 lines of LOTOS). Each test was created by providing an initial configuration according to the conditions included in the individual UCMs. For instance, we need three configurations to test INTL: *UnrestrictedTime*, *RestrictedTime* $\wedge$ *ValidPIN*, *RestrictedTime* $\wedge$ *InvalidPIN*. They correspond to the three partial UCM routes (abstract sequences) in Figure 4. These tests were applied to the specification, and results were collected (steps (7), (8), and (9) in Figure 1).

Many unique global sequences were generated by the composition of each test with the specification (INTL tests had 61). Each of these sequences could be represented as a unique MSC from the user's point of view. All of them were successful, therefore we do not have any indication that POTS and the individual features are faulty in our system. The validation then continues with the detection of unexpected interactions between pairs of features.

4.3. Detecting unexpected interactions

In theory, if the same type of integration used for merging the individual UCMs is 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 (step (5) in Figure 1). 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.

By covering the UCM paths and all the meaningful combinations of preconditions, we defined a set of 28 test cases (750 lines) that aimed to validate the features according to the intent expressed by the global UCM. With our first specification, all our test cases passed successfully, except the test for the pair INFB-TCS. LOLA returned three traces that led to unexpected deadlocks. One such trace was translated into an equivalent MSC, a form more appropriate for illustration (Figure 8): 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. 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 the TCS plugin was selected, but not INFB. Hence, the person to be billed was the default one, i.e., the originator.

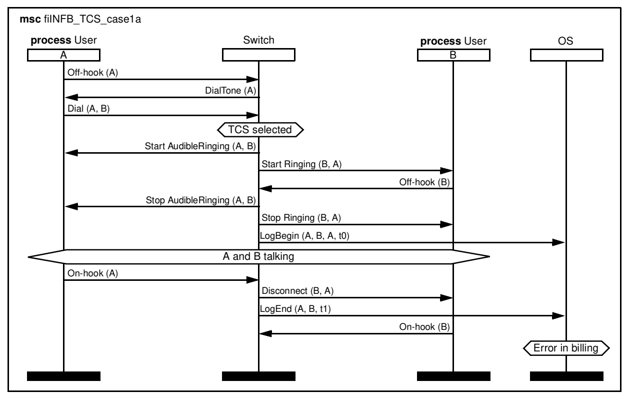


Figure 8. First FI, originator billed, not terminator

The second interaction trace is similar, but the originator on-hooks first. In the third trace, the originator is on the terminator's screening list. The call should be blocked by TCS, but it goes through because INFB was selected while TCS was not.

The choice between the TCS plugin and the INFB plugin in stub **process-call**, 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 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. We specified this new composition, adapted our FI test cases, and in the end all of our test cases (POTS, individual features, and pairs of features) passed successfully. No expected interactions seemed to remain in the global specification.

4.4. Ensuring coverage with probes

Probe insertion is a well-known white-box technique for monitoring software in order to identify portions of code exercised. 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 of the code is unreachable.

We have adapted this approach for LOTOS specifications (step (6) in Figure 1). We inserted 55 probes (one for each basic behaviour expression) in the specification of the system. Out of these, 5 were not covered by the whole test suite, but for good reasons (these cases were related to features not yet fully specified). 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.

5. Discussion

Adding new features has a direct impact on the global UCM, the specification, and the test suite. In our experience, the integration of two new features (the second phase of the FI contest) to the first ten 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 new stub and a new output path had to be added to a plugin. The impact appears to be proportional to how coupled the features are in a map. The more a map is modularized (by using stubs), the less it is likely that major modifications will be necessary. Since the LOTOS specification reflects the global UCM, the conclusions are basically the same. We added a few new gates and appropriate data structures, and new processes for the new plugins.

The addition of a feature has a profound impact on the test suite. The number of pairs among n features is $n*(n+1)/2$ (if we assume that a feature may interact with itself), and for each pair the number of test cases grows exponentially with the number of distinct conditions to be covered. 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. However, UCMs can help to determine some unnecessary combinations of conditions by following the paths and their associated conditions. For instance, when we have *UnrestrictedTime* in INTL (Figure 4), whether the PIN is valid or not has no impact, hence one of these two combinations can be dropped.

Our approach has several advantages over conventional (telecommunication) requirements engineering with MSCs and SDL [15]. These two techniques require an early commitment to components and messages for scenarios and features. Moreover, SDL needs states to be explicitly defined in the specification. LOTOS does not have such constraints, and it even allows the specification of UCMs without components (e.g., Figure 2b)

From a tool perspective, testing with LOLA seems to be an efficient solution for the validation of prototypes and the detection of feature interactions. The compilation of this 2800-line specification and the execution of all the test cases take about 30 seconds on a low-end PC (Cyrix P150, Win95). 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.

6. Conclusions and future work

This paper presents an approach for the avoidance and detection of interactions between operational requirements at design time. These requirements, telephony features in our study, are captured as UCM scenarios, integrated in one global map with stubs and plugins, and then transformed into a LOTOS specification, which provides formal semantics to semi-formal notation. Test cases are generated from the UCMs, and we use them to validate the integration and to detect unexpected interactions.

UCMs describe systems at a useful level of abstraction. We showed how, during their integration, some interactions can be avoided by insuring disjoint and complete preconditions and by composing plugins in stubs according to the intent of the features. Further design decisions are necessary when synthesizing the specification, although the burden of the integration is mostly taken care of at the UCM level. Test selection strategies 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. Several such interactions were detected. They were caused by the composition of plugins in a stub, and this is where the problem was resolved. 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.

We plan to use this methodology on a new telecommunication product that involves several hundred features. Here are several research issues and work items we plan to address:

- The intent of a UCM scenario would be better described by indicating which events or paths should obliged, permitted, or forbidden. Forbidden paths would also allow for a better way of generating rejection test cases.
- Use of other LOTOS validation techniques for further FI detection.
- Generation of UCM-based abstract test suites in TTCN.

Acknowledgements

This work has been supported by *Communications and Information Technology Ontario* (CITO). We would like to thank Dorin Petriu for capturing the features as UCMs. Andrew Miga and Chris Witham supported us with the UCM Navigator tool. Finally, we are thankful to the agent group at Carleton U., to Mitel, and to the LOTOS group at U. of Ottawa for useful discussions.

7. References

[1] Aho, A., S. Gallagher, N. Griffeth, C. Scheel, and D. Swayne, "Sculptor with Chisel: Requirements Engineering for Communications Services". In [20], pp. 45-63. http://www-db.research.bell-labs.com/user/nancyg/sculptor.ps

[2] Amyot, D., L. Logrippo, and R.J.A. Buhr, "Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage". In: *CFIP 97, Ingénierie des protocoles*, Hermes, Paris, 1997, pp. 159-174. http://www.csi.uottawa.ca/~damyot/cfip97/cfip97.pdf

[3] Amyot, D., N. Hart, L. Logrippo, and P. Forhan, "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

[4] Amyot, D., *Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features*. CITO report #1430, Ottawa, Canada, 1998. http://www.csi.uottawa.ca/~damyot/FI/

[5] Bouma, L.G., and H. Velthuijsen (eds), *Second International Workshop on Feature Interactions in Telecommunications Systems*, IOS Press, Amsterdam, Netherlands, 1994.

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

[7] Buhr, R.J.A., and R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM_book95.pdf

[8] Buhr, R.J.A., D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski, "Feature-Interaction Visualization and Resolution in an Agent Environment". In: [20], pp. 135-149. http://www.UseCaseMaps.org/UseCaseMaps/pub/fiw98.pdf

[9] Buhr, R.J.A., "Use Case Maps as Architectural Entities for Complex Systems". In: *Transactions on Software Engineering*, IEEE, December 1998, pp. 1131-1155. http://www.UseCaseMaps.org/UseCaseMaps/pub/tse98final.pdf

[10] Cameron, E.J., N. Griffeth, Y.-J. Linand, M.E. Nilson, W.K. Schnure, and H. Velthuijsen, "A Feature Interaction Benchmark for IN and Beyond". In [5], pp. 1-23. http://www-db.research.bell-labs.com/user/nancyg/benchmark.ps

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

[12] Faci, M., and L. Logrippo, "Specifying Features and Analysing their Interactions in a LOTOS Environment". In: [5], pp. 136-151. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Papers/Fits94.CameraReady.ps.gz

[13] Griffeth, N.D., O. Tadashi, J.-C. Grégoire, and R. Blumenthal, "First Feature Interaction Detection Contest". In [20], pp. 327-359. http://www.tts.lth.se:80/FIW98/contest.html

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

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

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

[17] Jacobson, I., M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, ACM Press, 1992.

[18] Kamoun, J., and L. Logrippo, "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model". In [20], pp. 172-186.

[19] Kimbler, K. and Søbirk, D., "Use case driven analysis of feature interactions". In [5], pp. 167-177.

[20] Kimbler, K. and W. Bouma (eds.), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems*, IOS Press, Amsterdam, Netherlands, 1998.

[21] 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/UseCaseMaps/pub/am_thesis.pdf

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