Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System

Daniel Amyot and Luigi Logrippo

School of Information Technology and Engineering (SITE)
150 Louis-Pasteur, University of Ottawa,
Ontario, Canada, K1N 6N5
+1 613 562 5800 (ext. 6704)
damyot@site.uottawa.ca, luigi@eiti.uottawa.ca

ABSTRACT — SPEC-VALUE, a rigorous scenario-driven approach for the description and validation of complex system functionalities at the early stages of design, is presented. It is based on two notations. The first notation, called Use Case Maps (UCMs), is used to capture functional requirements. UCMs can help reasoning about system-wide functionalities at a high level of abstraction before a prototype is generated. The second notation is the formal specification language LOTOS. UCM scenarios are translated into LOTOS specifications, which animate UCMs with the help of tools. LOTOS-based techniques, especially specification-level testing, can be used to validate designs. It is shown how SPEC-VALUE can help to produce better-quality designs and standards and to improve human understanding with reduced time and costs. A real-life case study is provided: the Group Call service of the mobile data system General Packet Radio Services (GPRS).

KEY WORDS: Causal Scenarios, LOTOS, Telecommunications Standards, Use Case Maps, Validation Testing.

1. INTRODUCTION

1.1 The Design and Standardization Challenges

Numerous industries and standardization bodies (ITU, ISO, ANSI, ETSI, TIA, etc.) are constantly at work to design new telecommunications products and new standards for such products, involving increasingly complex functionalities. These services require increasingly complex architectures and protocols. In the early stages of many conventional design processes used in industry and in standardization bodies, many features, services, and functionalities are described using informal operational descriptions, tables and visual notations such as *Message Sequence Charts* (MSCs) [26]. As these descriptions evolve, they quickly become error-prone and difficult to manage. The need of precisely documenting all stages of the design process, which is very important in the industrial environment, becomes critical in the standardization process, where there is international scrutiny for which the stages are formalized and must undergo formal review and approval [3].

The following issues should be addressed:

- While designing systems and services in the initial stages, the discussion must focus on a level of detail that reflects the level of knowledge (about data, messages, components, etc.) available at the time. Descriptions that include irrelevant details tend to obscure the main idea behind a feature/service/functionality, especially when the latter needs further modifications or refinements. Several levels of detail and abstraction, similar to *viewpoints* in *Open Distributed Processing* (ODP) [22] and *planes* in *Intelligent Networks* (IN) [25], are often mixed in a single description.
- A simple visual notation, which abstracts from messages while focusing on the tasks to perform and their cause-to-effect (or *causal*) relations, can help focusing on the general control flow while providing for more maintainable and reusable scenario descriptions. The Message Sequence Charts (MSC) notation is very commonly used, but its focus is on message exchanges, which come into consideration later at the detailed design stages. Such a focus can be inappropriate while defining the functionalities in the initial stages of the design, when details related to messages and system components might be unknown [4].
- There are possibly ambiguities, inconsistencies or undesirable interactions inside or between service descriptions, or between levels of abstraction of a given service. These remain difficult to detect with conventional inspection methods, and often remain hidden until errors are discovered after implementation, at which point corrections can be very costly and system interoperability can be jeopardized.

1.2 On Scenarios and FDTs in a Design Approach

The process of going from informal functional or operational requirements to a high-level formal specification is a research subject where much work has been done [3][6][14]. However, many challenges, such as the ones presented in the previous section, still remain. *Formal Description Techniques* (FDTs), such as LOTOS [21] and SDL [24], were created in order to formally express functional requirements, and hence to answer some of these challenges. In particular, FDTs are well suited for the precise definition of telecommunication systems. Although they help avoid ambiguities and inconsistencies, FDTs often require an inappropriate level of detail and completeness in the preliminary stages of standards definitions. Furthermore, they are not easy to learn, and this slows down their penetration in industry.

Several techniques can be used to address the issues related to design and standardization processes (Section 1.1). Over the last few years, there has been a strong interest, in both academia and industry, in the use of *scenarios* for requirements engineering and system design [43]. The introduction of *use cases* [27] in the object-oriented world confirmed this trend. The exact definition of a scenario may vary depending on semantics and notations, but most definitions include the notion of a partial description of system usage as seen by its users [35]. In this paper, we consider the terms “scenario” and “use case” as synonyms, but the literature often distinguishes them by defining a scenario to be a specific realization of a use case [40][34]. Many methodologies are now available and they often have a high degree of acceptance because of the intuitive and linear nature of scenarios [20][40][43]. However, many different meanings have been associated to the word “scenario”. They are related to traces (of internal/external events), to message exchanges between components, to interaction sequences between a system and its user, to a more or less generic collection of such traces, etc. Numerous notations are also used to describe scenarios: informal pic-

tures [27], natural language or structured text [34][40], grammars or automata [19], tables [13], and message exchange diagrams similar to MSCs [27][34][38][40], to mention but a few. The approaches available differ on many aspects, depending on the definition and the notation used.

A more rigorous approach, based on scenarios and supported by a formal description technique, would allow the design process to focus on the main functional aspects of systems, and hence better to cope with some of the complex problems related to the design, documentation, validation, and maintenance of systems and standards. In Section 2, we present such an approach where informal requirements are partially captured as causal scenarios through the use of a visual notation called *Use Case Map* (UCM) [10][11]. These designs are also documented with tables describing the activities to be performed. In order to detect logical errors, inconsistencies, ambiguities, and undesirable interactions between the scenarios, we use LOTOS to specify the integration of the scenarios and their distribution over a topology of components. The choice of these two notations is justified in Section 2.1 and Section 2.3 introduces the main UCM concepts with a simplified GPRS operation. The reader unfamiliar with LOTOS is invited to read Section 2.4 for an overview of the operators.

The goal of this approach is to produce better quality standards and to improve the overall understanding of distributed systems, while decreasing the required time, costs, and efforts. We illustrate this approach on a real-life example from a mobile telephony standard: the Point-to-Multipoint Group Call service of General Packet Radio Services. One of its functionalities is illustrated as a Use Case Map (Section 3). We show how UCMs helped in the construction of our GPRS specification (Section 4) and in its validation (Section 5). A discussion that includes a comparison with similar techniques is given in Section 6, followed by our conclusions. Due to the large number of acronyms used in this paper, a glossary of acronyms is included in Appendix A.

1.3 General Packet Radio Services (GPRS)

GPRS [16] is a mobile telephony service at an advanced stage of standardization at the time this paper is written. It allows its subscribers to send and receive data in an end-to-end packet transfer mode. Built on top of the concepts and technologies of *Global System for Mobile Communications* (GSM) [28], a connection-oriented service for mobile telephony, GPRS provides connectionless packet transfer within the *Public Land Mobile Network* (PLMN) in interworking with external networks (X.25 and TCP/IP).

There are two main categories of GPRS services: *Point-To-Point* (PTP) and *Point-To-Multipoint* (PTM), based on existing and standardized network protocols. Typical applications for PTP services include retrieval services (Web), messaging services (mailbox), real time conversations (Telnet), and short tele-actions (credit-card validation). Typical applications for PTM services include unidirectional distribution services (newsgroups), bidirectional dispatching services (for a taxi fleet), and multi-directional conferencing services, without store and forward, between multiple users. The PTM services have several capabilities, including geographical routing to restrict distribution and scheduled delivery.

In this paper, we focus on the *PTM-Group Call* (PTM-G). This service allows transmissions of messages to specific groups of users in specific geographical areas (see Figure 1). At any point in time, the network has the knowledge of the actual group call participants present within the given area.

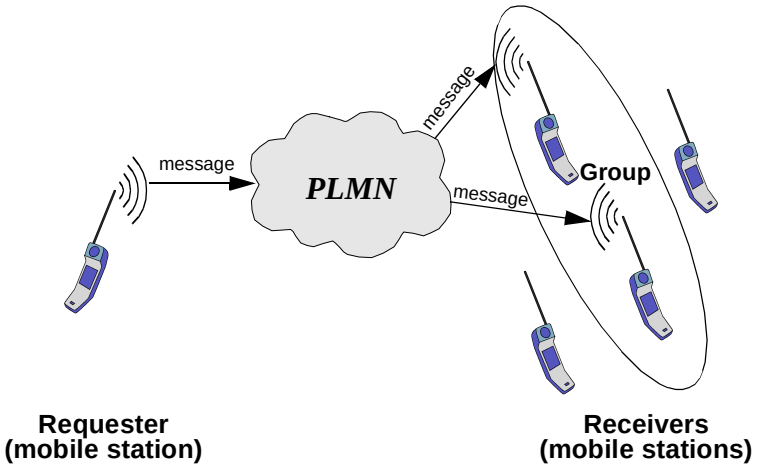


Figure 1. *Illustration of GPRS Group Call*

2. DESIGN APPROACH BASED ON SCENARIOS

2.1 Motivation

In the first stages of design processes, we anticipate instabilities in the requirements as well as volatility of scenarios and potential component topologies (structures). Hence, an iterative and incremental process (in spiral form), which allows rapid prototyping and test cases generation directly from the same scenarios, seems appropriate.

We believe that the usage of Use Case Maps in a scenario-based approach represents a judicious choice for the description of communicating and distributed systems in a design process. UCMs represent scenarios as causal paths cutting across organizational structures of components. They fit well in 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. This is exactly the design phase that is of interest to us in this paper (see Section 2.3.1). Furthermore, UCMs can help validating the system design against the requirements.

Pragmatically, our choice of the UCM methodology is justified by the enthusiastic acceptance in a few companies that were introduced to it in recent years. Such acceptance makes us think that UCMs do capture important aspects of the design process and that eventually they will be used widely. One of the strengths of UCMs is their capability to illustrate scenarios without having to describe the messages exchanged between the system components. In later design stages, such messages can be derived from UCMs once the necessary information is available.

For the formal representation of our scenarios, we opted for LOTOS, the Language of Temporal Ordering Specification, an algebraic specification language standardized by ISO [21]. In LOTOS, the specifier describes a system by defining the temporal relations among the actions that constitute the system's externally observable behaviour. The main influences for the behaviour part of LOTOS were CCS [29] and CSP [18]. LOTOS behaviour expressions are built from elementary actions by using operators such as *action prefix, choice, parallel composition, multiway synchroni-*

zation, hiding, process instantiation, and a few others. Data abstractions are specified with *Abstract Data Types* (ADT). 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 [3]. Our choice of LOTOS is motivated by several factors:

- The language is capable of expressing behaviours at several stages of design or levels of abstraction, including the initial ones where it is not yet known what the components of the system are, what are their states, and what are the messages exchanged between them (this is normally the situation at the early stages of design).
- UCM scenario paths can be fairly easily translated into LOTOS [1].
- LOTOS specifications are executable prototypes that can be formally analysed and validated against the intended functionalities of the individual scenarios.
- The language is mature in the sense that it is an established international standard, around which much useful theory and a number of useful tools have been developed.

Because UCMs are not yet as popular and well-known as LOTOS, they will be presented thoroughly in Section 2.3.

2.2 The Approach: SPEC-VALUE

Figure 2 introduces the approach used in this project, the *Specification-Validation Approach with LOTOS and UCMs* (or *SPEC-VALUE*). Its main cycle is first concerned with the description of the structure ① and of the scenarios ②, which can be done independently. A structure contains the abstract system components of interest (mostly software, but also hardware), as well as some of their relationships (containment, communication links, etc.). Services and functionalities are captured (scenario elicitation) as Use Case Maps. These UCMs represent scenarios emphasizing the causal relationships among the responsibilities that compose services and large-grain functionalities. The responsibilities in the UCMs are then allocated to the components in the selected underlying structure ③. Each component will have to perform the responsibilities allocated to it. Next, we combine the scenarios (manually) to synthesize a LOTOS specification ④, which becomes the executable prototype enabling formal validation ⑥.

Concurrently with these steps, validation test cases can be generated from the individual scenarios ⑤ to ensure that the specification conforms to each intended functionality. The test cases are described in the same language as the specification, i.e. LOTOS. These tests check the integration of the functionalities, which is currently done manually in SPEC-VALUE. They also check that the integrated behaviour emerging from the collaboration among the components in the system structure corresponds to the intended behaviour expressed by the UCMs. Because UCMs are close to the requirements, verifying that the LOTOS specification conforms to the UCMs is a way of validating the high-level design against the requirements. Probes can be inserted in the specification to measure the structural coverage of the specification by the test suite.

Once the specification has been tested against all the test cases, results and statistics ⑥ can be obtained automatically from the resulting execution trees (*Labelled Transition Systems,* or LTSs). One of the following verdicts will occur:

- At least one test case derived from a UCM has failed. A logical error was detected in the functionality exercised by this test case. The faulty functionality has been incorrectly specified according to its UCM, or this functionality was incorrectly integrated with the others (this may be caused by an undesirable interaction or conflict between the functionalities involved).
- 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 ⑦, or the specification covers a case that should not be considered.
- The test suite has passed successfully and all probes have been covered. The specification conforms, according to the test selection strategy, to the UCMs (and hence to the functional requirements). No undesirable interaction between the functionalities was detected. We then have a good level of confidence in the UCMs, in the LOTOS specification, and in the test suite.

Following the verdict, modifications may be required to the UCMs, to the test cases ⑦, to the specification, or even to the requirements ⑧. In fact, the approach of Figure 2 is iterative. It is also incremental as new functionalities may be integrated at a later time.

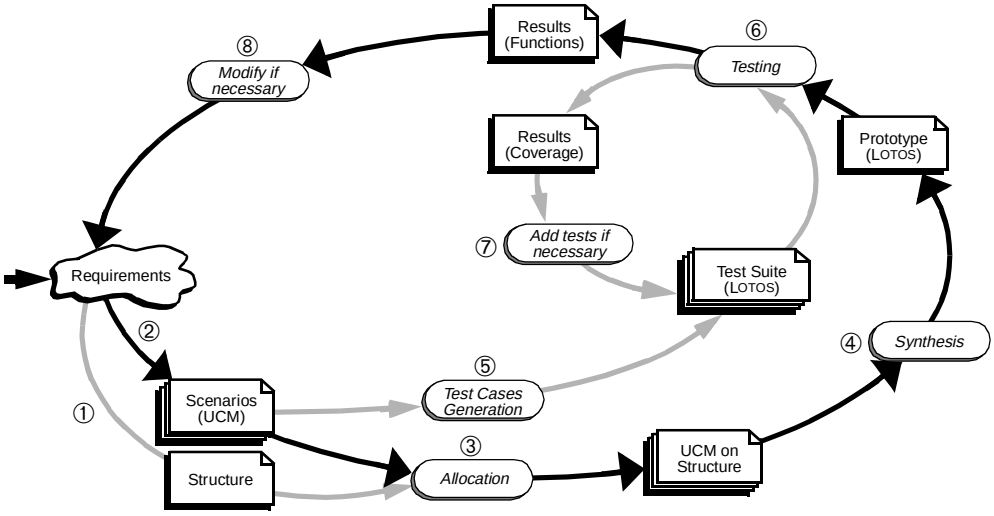


Figure 2. *Specification-Validation Approach with Lotos and UCMs (SPEC-VALUE)*

We claim that SPEC-VALUE has several advantages:

- **Separation of the functionalities from the underlying structure**: since scenarios are formalized at a level of abstraction higher than message exchanges, different underlying structures or architectures can be evaluated, even before the generation of a prototype. The scenarios then become highly reusable entities, for example they can be used on different equipment and for different products. Senior managers and senior designers can keep control over the general logic of the design without having to know the characteristics of the latest equipment. This separation helps to cope also with the incremental addition of new functionalities that require modifications to the structure.
- **Fast prototyping**: once the structure and the scenarios are selected and documented, and once the responsibilities have been allocated to their respective components, a prototype can be generated rapidly. This is mainly due to the ease with which LOTOS constructs can formalize UCM

constructs. Formal prototyping adds rigor to scenario-based requirements and engineering with UCMs because UCMs are semiformal and non-executable.

- **Test cases generation**: scenarios guide the generation of test cases, hence allowing the validation of the prototype against the UCMs and the informal functional requirements. The test suite can itself be validated using structural coverage criteria on the model. It can be reused as a regression test suite in the subsequent steps of the development process.
- **Design documentation**: the documentation of requirements and designs is done as we go along the development cycle. It is also adapted to the expressive needs of the different people involved in the design process. The part related to scenarios should be understandable by marketing people and service operators. These people do not have to know every technical detail described in the formal specification (such as message exchanges), since such details may be important only for engineers, implementors, or testers. UCMs allow different specialists to become involved in discussions at different levels while a sharing common language and, hopefully, understanding.

2.3 Use Case Maps (UCM)

The visual notation Use Case Maps is used for capturing the operational requirements of communicating 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 the UCM notation, 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 validation test cases.

To illustrate these concepts, we will use several simplified versions of one operation of the GPRS Group Call (see Section 3.1). The *Join Call* operation consists, for a mobile station, of attempting to join an existing group call. The UCM notation will be introduced through the examples. For a detailed description of this notation, the reader should refer to [10] and [12].

2.3.1 Overview of the Notation with a Simplified Join Call Operation

Figure 3d shows a UCM where a requesting mobile station (**Req**) attempts to join, through the PLMN, a group call where a receiving mobile station (**Rec**) is already a participant. **Req** first initiates a connection request (join) to the PLMN. The latter verifies (vrfy) whether or not the requester is allowed to become a participant. If so, then the PLMN will need to update (updjoin) its list of members for that particular group call. Then, an indication (ind) will be activated on the receiver's side, so it can become aware that a new party joined the call. If the requester is not allowed to join the call (e.g. **Req** may not be in the right geographical zone), then the PLMN status will be updated differently (updrej) and a message stating that **Rec** is not available (err) will be sent back to **Req**.

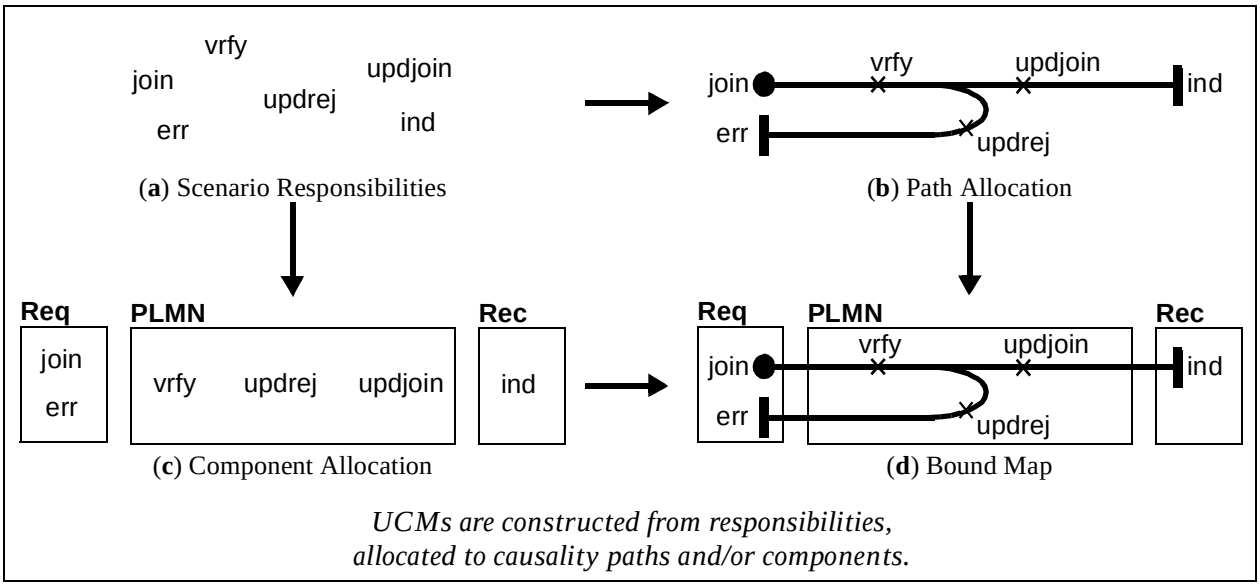


Figure 3. *Use Case Maps Construction*

A scenario starts with a triggering event or a precondition (filled circle labelled join) and ends with one or more resulting events or postconditions (bars), in our case ind or err. We call *route* a path that links a cause to an effect. Intermediate responsibilities (vrfy, updjoin, updrej) 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 **Req**, **Rec**, and **PLMN**), 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 *OR-Fork* in the Figure 3b), concurrent paths, exception paths, timers, stubs/plug-ins, 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 3a), although not necessarily with diagrams like this one. They can then be allocated to scenarios (Figure 3b) or to components (Figure 3c). Components can be discovered along the way. Eventually, the two views are merged to form a bound map (Figure 3d). Unbound maps (Figure 3b) express scenarios when decisions about the underlying structure remain to be taken. Even at this level, causal relationships and system behaviour can be reasoned about both informally and formally [1].

2.3.2 Causal Versus Temporal Relations

An early focus on causal relations between responsibilities helps to avoid several misunderstandings related to temporal relations. These problems seldom occur in sequential systems, but they often do when concurrency is involved, such as in communicating and distributed systems. For example, Figure 4 shows a different UCM scenario where the join call request is accepted. An acknowledgment is sent back concurrently (expressed with the *AND-Fork* after the initial path segment) with the update of internal databases followed by an indication on the receiver side. We clearly see that both updjoin and ack are caused by join, although they are independent from one

another. A global sequence of events such as <join, updjoin, ack, ind> is possible in this system, but it is only one of the many possible global scenarios, and it should not be interpreted as "updjoin causes ack". The causal dependence between events should be documented in the early stages of the design process, before this information gets lost among the details of the behaviour of individual components. UCMs are very helpful in this context.

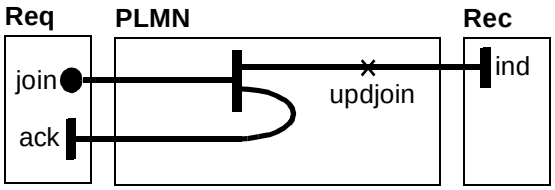


Figure 4. *Scenario with Concurrency: Acceptance of a Join Call Request, with Acknowledgment*

2.3.3 Evaluation of Structures

The notation supports the reuse of scenarios when the underlying structure is modified or refined. For example, Figure 5c contains a new structure with two functional entities: **Ctrl,** the PLMN controller (not necessarily a realistic one), and **Group**, the active process responsible for the management of a specific group call. Our original scenario (Figure 5a) is bound differently in Figure 5b, where the resulting event (err) ends in **Req**, and in Figure 5c, where err remains local to **Group** (for some reason, e.g. this **PLMN** does not provide acknowledgements). During the service specification or design phases, different potential structures could undergo some evaluation (architectural reasoning). Scenarios described in terms of communicating components, such as Message Sequence Charts or UML (*Unified Modeling Language*) interaction diagrams [40], would need to be rebuilt as soon as there is a change in the underlying structure, because the functionalities are tightly bound to how the structure looks. UCMs require simpler modifications, consisting only of a new a binding between the responsibilities and the components.

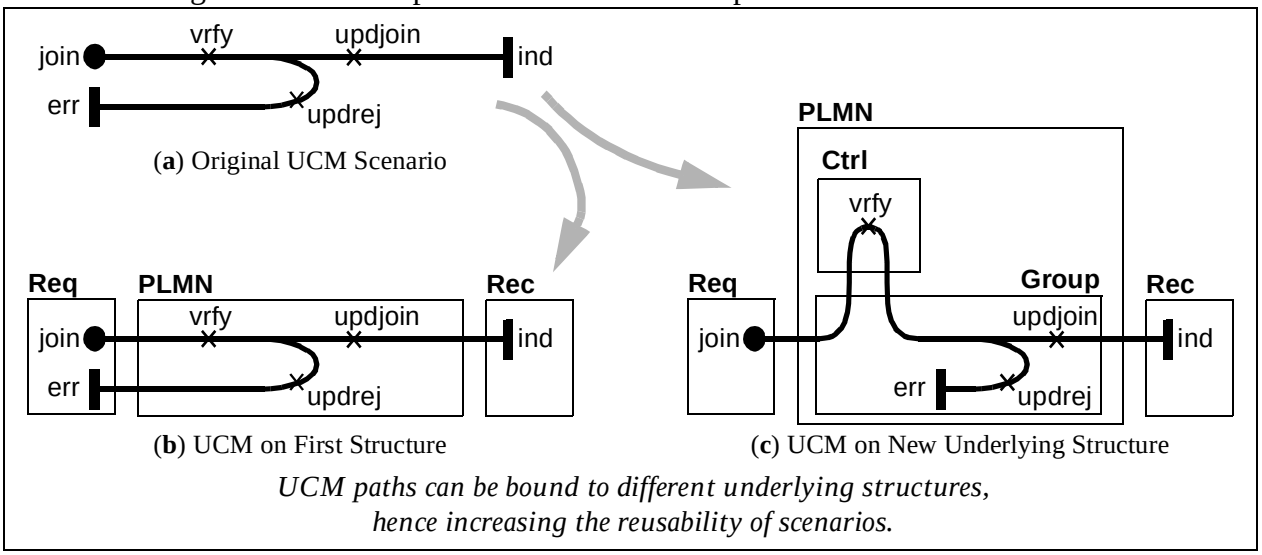


Figure 5. *A Causal Scenario Bound to Different Structures*

2.3.4 Refinement with Message Exchanges

A causal relationship, such as the one described by the route <join, vrfy, updjoin, ind> in Figure 3b, can be refined in many ways in terms of exchanges of messages, depending on the component structure, on the availability of communication channels[1], 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 [26]. 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) link the components of Figure 6a. They constrain the sequences of messages allowed for the implementation of the causal relations in scenarios. Figure 6b presents an MSC where the exchange of messages is minimal. Notice that **Req** is not allowed to send messages directly to **Ctrl**, but messages can be forwarded through **Group**. Figure 6c illustrates a case where more complex protocols are used between **Group** and **Ctrl**, and where many messages (perhaps some sort of negotiation) are required for the implementation of the causal relationship between vrfy and updjoin.

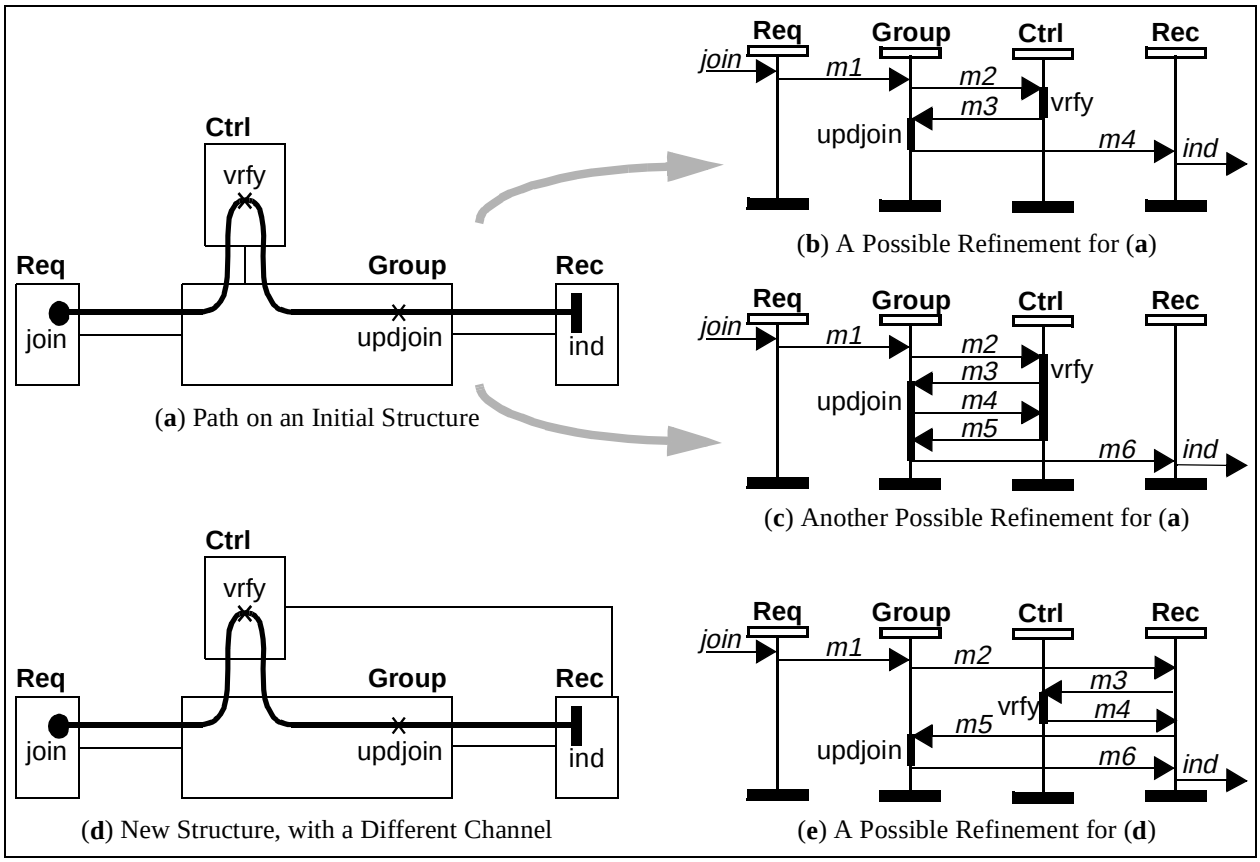


Figure 6. *Causal Scenarios and Exchanges of Messages*

The structure in Figure 6d sees one of its channels in a place that is different from that of Figure 6a. The two MSCs discussed previously become invalid because the constraints on the

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

communication channels no longer hold (**Group** and **Ctrl** are no longer allowed to communicate directly). Figure 6e suggests a possible refinement for the new structure. Again, we observe the impact of premature decisions on message exchanges, something that is avoided at the UCM design level.

2.4 LOTOS Operators

This section provides an overview of the main LOTOS operators to the reader not familiar with this specification language. LOTOS specifies system behaviour with *actions* (e.g. updjoin or vrfy) and *behaviour expressions*. Actions with value offers occur on *gates*, which are similar to ports used for synchronization. There is an internal (invisible) action written **i**, which is never involved in synchronizations. There are three basic behaviour expressions, and more complex expressions can be formed as shown in Table 1, where a is an action, B_i are behaviour expressions, g_i are gates, v_i are values, and P is a predicate.

	Name	*Behaviour Expression*	*Comment*
Basic Behaviour Expressions	Inaction	**stop**	Cannot engage in any interaction (deadlock).
	Successful Termination	**exit**(v_1,...,v_n)	Terminates successfully. Return values may optionally be specified.
	Process Instantiation	ProcName [g_1,...,g_n]	Creates an instance of a process ProcName.
Basic Operators	Action Prefix	a; B	Prefixes a behaviour expression B with an action a.
	Choice	B_1 [] B_2	Offers a choice between two behaviour expressions.
	Enabling	B_1 >> B_2	Sequences two behaviour expressions. B_1 has to **exit** for B_2 to be executed. Values may be passed through the construct: B_1 >> **accept** parameters **in** B_2
	Disabling	B_1 [> B_2	B_1 can be disrupted by B_2 during normal functioning.
Composition	Parallel Composition	B_1 |[g_1,...,g_n]| B_2	B_1 and B_2 behave independently, except for the gates g_1,...,g_n where B_1 and B_2 must synchronize.
	Interleaving	B_1 ||| B_2	B_1 and B_2 behave independently (the synchronization set is empty).
	Full Synchronization	B_1 || B_2	B_1 and B_2 are synchronized on all their gates.
Other Operators	Hiding	**hide** g_1,...,g_n **in** B	Hides actions g_1,...,g_n, which become internal and can no longer synchronize with the environment.
	Guarded Behaviour	[P] -> B	B can be executed if P is true.
	Local Definition	**let** x:s = E **in** B	Substitutes a value expression (E) by a value identifier (x) of sort s in B.
	Process Definition	**process** ProcName [g_1,...,g_n] (parameters) : funct := B **endproc**	Creates a process definition with formal gates and parameters. The functionality funct indicates whether the process can terminate successfully (**exit**, optionally with values) or not (**noexit**). Can be instantiated as a basic behaviour expression.
	Comment	*(* This is a comment *)*	Comment skipped by the parsers.

Table 1. *Summary of LOTOS Syntax*

In LOTOS, data can be associated with actions in two ways: !value, which means *value offer*, and ?variable:type, meaning *value query*. These can be combined in actions, for example:

request !myGroupId ?service:serviceSort

denotes an action where on gate request, the current value of the variable myGroupId is offered, and a value for service (of type service_sort) is queried simultaneously. Offers and queries are called *experiments*. Selection predicates can be optionally added to value queries, as in:

request ?n:number [n>3]

meaning that the acceptable values for n are greater than 3. This example demonstrates the abstract nature of the language, since it allows to express in a single action system events that could be quite complex to implement.

3. SCENARIOS FOR GPRS GROUP CALL (PTM-G)

3.1 Informal Requirements and Assumptions

Six operations are defined in [16] for the implementation of the PTM-G service: *Initiate Call*, to create a group call; *Terminate Call*, to delete a group call; *Call Status*, to get the attributes of a group call; *Join Call*, to join an existing group call; *Leave Call*, to quit a joined group call; *Data Transfer*, to send messages and data.

In order to generate the Call Terminate and Leave Call operations invoked by the network, we define three artificial operations that we can trigger at will. Two of them are located in the underlying services: *Attach GPRS* and *Detach GPRS*. The third one, *Change Zone*, emulates the routing operation triggered by the physical layer.

In this paper, we focus on only one operation, namely *Initiate Call*, although general results for the whole set of functionalities will also be given. Upon an Initiate Call request (*Req_Init*), two outcomes are possible: the acceptance (*Ack_Init*) or the rejection (*Err_Init*). The main reasons for rejecting an Initiate Call request include an incorrect geographical zone, an invalid International Mobile Group Identity (*IMGI*), or an insufficient privilege access. In case of acceptance, the indication *Ind_Init* is multicast to the receivers only if the Initiate Call Notification (*call_not*) attribute defined for the call says so.

In order for this operation to be accessible, its *preconditions* need to be satisfied. A requester can only initiate a group call if it is attached to GPRS and if the group is already set up in the PLMN's databases.

Beside the usual member identification number (*M_ID*) and the IMGI, the different parameters provided with this request are the Data Transfer Mode (*DTM*), the quality of service (*QoS*), the geographical area (*GeoZone*), the *join_leave* indication (to inform receivers when someone joins or leaves the call), and the Initiate Call Notification (*call_not*). We also included a last parameter named *send_to_all*. Although this parameter is not defined in [16], it seems necessary as it allows the multicast of a join/leave indication to all members in the call. In the draft standard, the initiator is the only party allowed to receive this indication.

3.2 Structure

Our approach allows designers to determine the system structure independently of scenarios, i.e. before, during, or after the specification of the scenarios. As no concrete structure or architecture was imposed in the preliminary version of the standard, we decided to use an abstract structure (using logical entities) independent of the physical components of GPRS (step ① in Figure 2). Once the scenarios are validated with this structure, they can be mapped onto a more concrete architecture, potentially with regard to an existing system of a particular service provider. However, this step is postponed to a subsequent stage of the development cycle, possibly the detailed design, which is outside the scope of this work. As shown in Figure 5, other abstract and concrete structures could also be evaluated with the same scenarios.

Figure 7 presents our GPRS abstract structure. It shows several kinds of components represented with Buhr's notation [10]: active processes (e.g. OS task) as parallelograms, passive objects (e.g. databases) as rounded rectangles, and containers (teams) as rectangles. Arrows represent communication channels (unidirectional or bidirectional). The components in dotted lines are dynamically instantiated when required. Stacks of processes show that multiple concurrent instances may coexist.

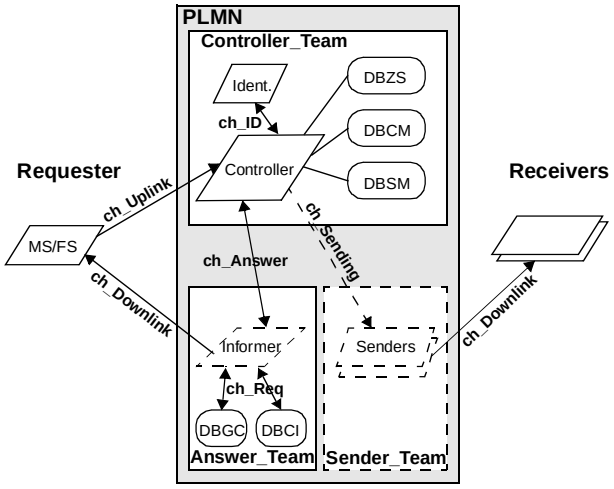


Figure 7. *GPRS Abstract Structure*

We have identified three teams in the **PLMN**, each with a specific role. The **Controller Team** manages the mechanisms of control and reception of requests. The **Answer Team** sends responses back to the **Requester**. The **Sender Team** sends indications, resulting from a request, to the other participants called **Receivers**. Processes **Informer** and **Senders** aim to relieve the main **Controller** from the direct communication of messages to requesters and receivers. **Ident** is a simple process that provides the controller with logical identifiers for available channels. A client can accumulate both the roles of requester and receiver as a Mobile Station, or even as a Fixed Station (**MS/FS**) since GPRS can support connections with non-mobile clients.

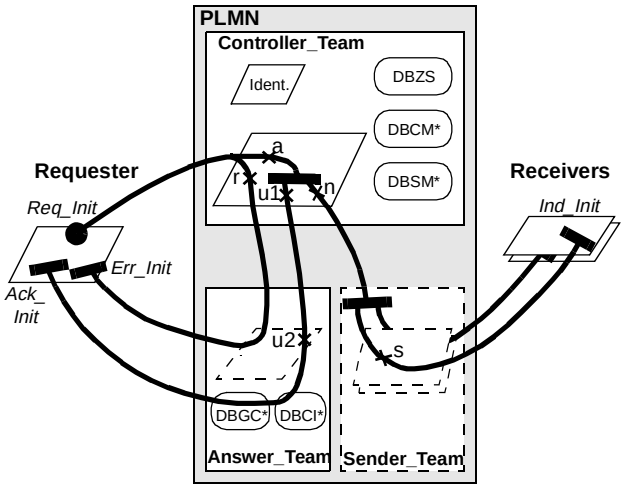
Figure 7 also shows the presence of several databases that contain the information required by our scenarios: **DBZS** for the localization of stations, **DBCM** for the list of members who joined a group call, **DBSM** for member characteristics, **DBGC** for the list of *Call-ID* of each group, and

DBCI for the parameters associated to each group call. Again, please refer to the glossary of acronyms (Appendix A) when necessary.

3.3 Scenarios

We described nine scenarios for the PTM-G service: six for the regular operations and three for the artificial ones (step ② in Figure 2). We obtained the first six UCMs fairly easily since GPRS services are described rather operationally, although very informally, in the draft standard [16]. Our scenarios usually start with a single triggering event, leading to one or possibly many resulting events.

The UCM of our Initiate Call example, which captures the causal relationships expressed in the informal requirements, is shown in Figure 8a. Channels have been removed from the picture to make it simpler (refer to Figure 7 for their description). This scenario uses one *OR-Fork* to express a choice between responsibilities a and r, and two *AND-Forks* to introduce concurrent paths. For instance, u1 and n can perform concurrently after a.



(**a**) Initiate Call UCM

Resp.	Type	Parameters	Alloc.	Comments
Req_ Init	Request	M_ID, IMGI, DTM, QoS, GeoZone, join_leave, send_to_all, call_not	Requester	Request creation of a group call, with requester attached.
Err_ Init	Error	IMGI, M-ID, cause	Requester	Request rejected (wrong GeoZone, wrong IMGI,...)
Ack_ Init	Ack.	IMGI, C-ID, Cipher_Key	Requester	Acknowledgement of call creation.
Ind_ Init	Indic.	IMGI, C-ID	Receivers	Indication of call creation.
r	Cond.		Controller	Request refused.
a	Cond.		Controller	Request accepted.
u1	Internal		Controller	Update DBCM, DBSM.
u2	Internal		Informer	Update DBGC, DBCI.
n	Cond.		Controller	Call Notification needed.
s	Internal		Senders	Sending notification.

(**b**) Description of Responsibilities

Figure 8. *UCM and Responsibilities Information for "Initiate Call" Operation*

Additional information on the responsibilities is provided in Figure 8b. In the first stages of design, it is in general possible to provide only a few details about their type (request, error, acknowledgement, indication, condition, or internal activity), parameters, allocation to a component (step ③ in Figure 2), and additional informal comments. Conditions can be refined more formally once the data types and data structures are known.

This UCM also has the preconditions previously introduced in Section 3.1. For Req_Init to be triggered, the **Requester** is required to be attached to the **PLMN**, and the databases need to contain the information describing the group to be called.

The stars beside the names of several database objects indicate that these databases are updated along the way by the process to which they are connected (see Figure 7). For instance, in this scenario, **DBCM** is updated by the **Controller** whereas **DBZS** remains as is.

UCMs can convey a lot of information in a compact and visual from. To support this claim, we explain two of the many end-to-end routes, described as temporal sequences, which can be extracted from the UCM of Figure 8:

- <Req_Init, a, u1, u2, n, Ack_Init, s, Ind_Init>: Normal scenario. The **Requester**, located in the right geographical zone and attached to GPRS, requests the initiation of a call to a specific group. The **Controller** accepts the request and updates **DBCM** and **DBSM** accordingly. The **Informer** also updates two databases, namely **DBGC** and **DBCI**. The **Controller** then observes that call notifications are needed. The acknowledgment of the Initiate Call request is propagated back to the **Requester**. The **Senders** prepare the notifications. There is one sender process for each group member. To keep things simple, we assume only one member in our group, i.e. the requester itself. This results in a call initiation notification in the **Receivers**.
- <Req_Init, r, Err_Init>: Error scenario. The **Requester** is attached to GPRS and it requests the creation of a group call. This time, the **Requester** is not in the appropriate geographical zone, hence the Initiate Call request is refused in **Controller**. The rejection is propagated back to the **Requester**.

Decisions on how the causal relationships are to be implemented with message exchanges are delayed until the specification is constructed.

4. LOTOS SPECIFICATION

4.1 Synthesis of Specifications from UCMs

The synthesis of LOTOS specifications (step ④ in Figure 2), illustrated with our Join Call scenario in Figure 3d, allows for the rapid generation of prototypes that represent UCM scenarios. The behaviour of each component is translated into a LOTOS process (Figure 9b) that preserves the internal causality relationships between the responsibilities and events that are part of path segments crossing this component. The structure itself is converted (Figure 9a) to a set of processes composed 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.

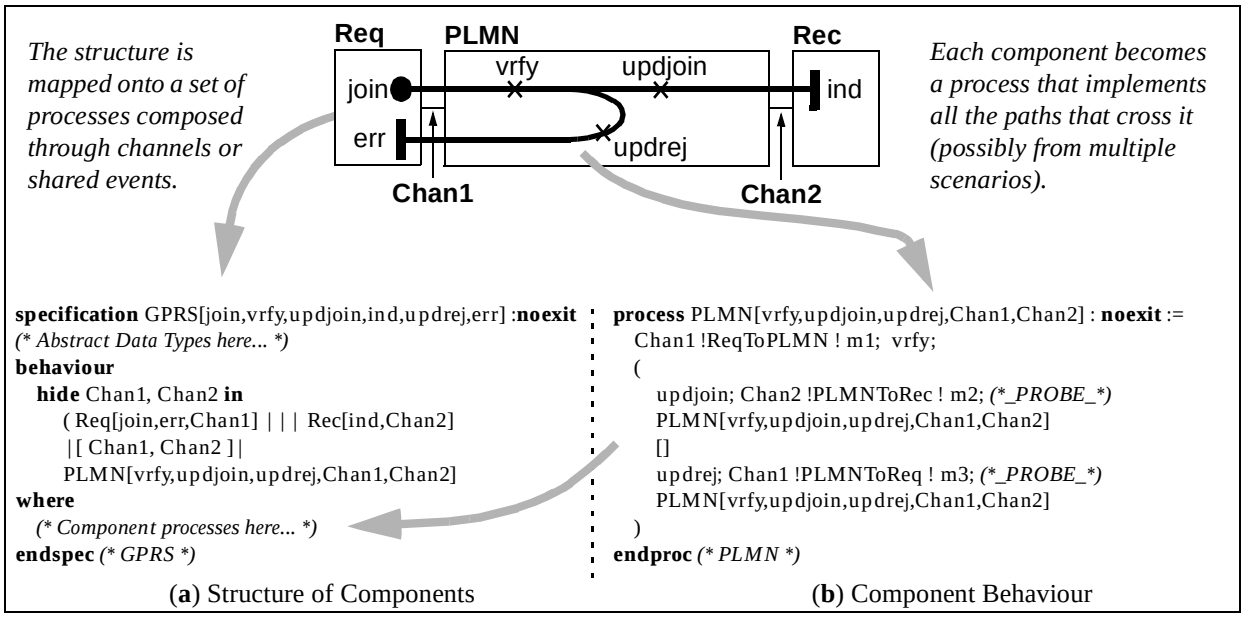


Figure 9. *Synthesis of a LOTOS Specification from a UCM*

We have defined guidelines that can help synthesizing specifications from UCMs. The design decisions that are necessary in this synthesis require creativity and make the automation a difficult task. Our guidelines are summarized as follow:

- Start points end points are usually represented by LOTOS gates in the prototype (e.g. gates join, vrfy, and ind). They can then be controlled and observed during the validation. If the component has a predetermined interface to the external world, then start and end points could be transformed into experiments (values) attached to the interface gate.
- Gates representing responsibilities and channels that are not observable by users are hidden (e.g. gates **Chan1** and **Chan2**).
- Components are implemented as processes synchronized on their shared channels/gates. For instance, the UCM component **PLMN** becomes the LOTOS process **PLMN** (Figure 9b), synchronized with **Req** on **Chan1** and with **Rec** on **Chan2**. LOTOS is abstract enough to represent any such component with only one construct, i.e. the process.
- The structure is specified mostly in a resource-oriented style [42], with multiway synchronization (|[...]|) and interleaving (| | |) operators.
- Containment of components is maintained. If the PLMN is refined with sub-components (as in Figure 5c), then these new processes are defined within the **PLMN** process.
- If multiple path segments (possibly from different scenarios) cross one component, they are integrated together in the LOTOS process, often as alternatives (e.g. **Req** contains two path segments).
- Elementary processes are specified mostly in a state-oriented style [42], with choice ([]) and action prefix (;) operators, and with guarded behaviours ([...]->).
- UCM activities are implemented as gates (this is the case in our example), sometimes with additional message exchanges (to ensure causality across components).

- Abstract data types are used to represent databases, operations, and conditions (LOTOS guard expressions).
- Symmetry is enforced in synchronized actions: actions in one process must be mirrored in the other synchronized processes, unless locally hidden. For instance, the operations on **Chan1** in **PLMN** have to be reflected in **Req**.

As explained in Section 2.4, LOTOS events are composed of gates with, optionally, experiments. Different conventions can be used to provide meaning to experiments in an event. For instance, experiments can represent messages or plain data, but they can also indicate some process identifier (when there are multiple concurrent instances of the same process definition) or the direction of a message between two components (LOTOS synchronizations are directionless). For instance, the event Chan2 !PLMNToRec !m2 found in **PLMN** has two experiments, the first one for the direction of the message and the second for the message itself. **Chan2** connects the **PLMN** to **Rec** (these two processes are synchronized on **Chan2**), hence for any synchronization to occur, **Rec** has to offer a compatible event in its process definition (e.g. Chan2 !PLMNToRec ?msg:message_sort).

Some special comments *(*_PROBE_*)* indicate locations where probes are to be inserted for measuring the structural coverage of the specification by the test cases. As comments, they have no impact on the behaviour of the specification, but they enable tools to translate them into hidden gates with unique identifiers (the probes) at a later time. Section 5.3 will present this technique.

4.2 GPRS Specification

4.2.1 Structure of the Specification

The structure of the specification in Figure 10 is derived from the abstract component structure found in Figure 7 following the guidelines we have just presented. Each component (process, object, and team) is mapped to a corresponding LOTOS process (represented as boxes), except for **DBZS**, **DBCM**, and **DBSM**, which we decided would better represented as parameters for process **Controller**.

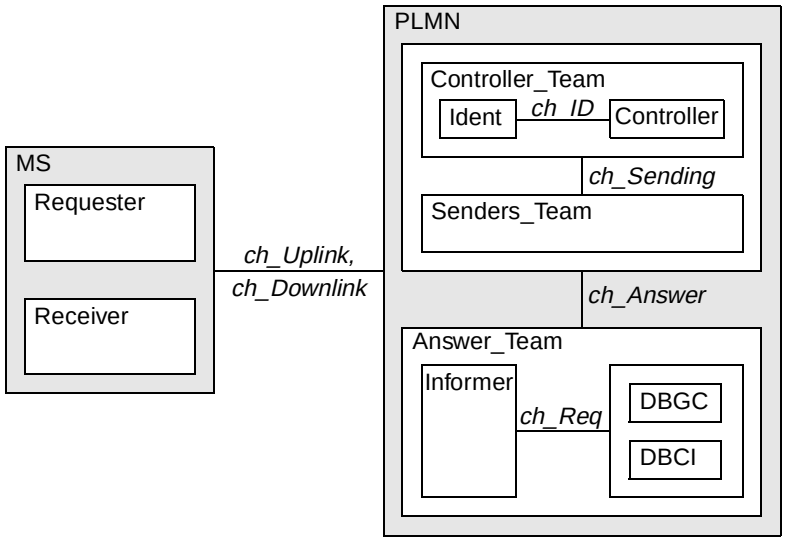


Figure 10. *Structure of the LOTOS Specification*

Containment relationships are also maintained (e.g. **Informer** is defined within **Answer_Team**, itself defined within **PLMN**). Channels become gates (represented as lines with italicized labels) on which processes synchronize. Several abstract data types, mainly enumerations and lists of complex data structures, were defined to specify our databases, variables, and parameters.

Note that some groupings of components in the specification structure differ from those of the abstract component structure. This is because LOTOS processes are composed using *binary* operators. However, the resulting groupings and possible ways of establishing communications are semantically equivalent.

Figure 11 presents two sample processes that represent partial structural groupings in Figure 10. In Figure 11a, we interleave the two possible roles for a mobile station, namely Requester and Receiver (boxes that are not linked in Figure 10). We also provide an identifier (n) for the resulting MS process. This allows us to instantiate multiple mobile stations that are unique and that can realistically assume both roles at the same time.

Figure 11b presents the first level of decomposition of the PLMN, with the necessary synchronizations between the three teams involved. The parentheses surrounding Controller_Team and Senders_Team correspond to the unnamed box in Figure 10. Notice how the process hides the communication channels that the mobile stations cannot access directly. This encapsulation mechanism is useful for defining precise interfaces, and it promotes the use of black-box testing for validation. All the other components in the structure are specified in a similar way.

```
process MS[ch_Uplink, ch_Downlink] (n: M_ID) : noexit:=
  Requester[ch_Uplink,ch_Downlink] (n)
  |||
  Receiver[ch_Downlink](n)
endproc  (* process MS *)
```

(**a**) MS, with Interleaved Sub-Processes

```
process PLMN[ch_Uplink, ch_Downlink]: noexit :=
  hide ch_Answer, ch_Sending, ch_ID in
    (
      (
        Controller_Team[ch_Uplink, ch_Answer, ch_Sending, ch_ID]
        |[ch_Sending]|
        Senders_Team[ch_Downlink, ch_Sending]
      )
      |[ch_Answer]|
      Answer_Team[ch_Downlink, ch_Answer]
    )
endproc (* process PLMN *)
```

(**b**) PLMN, with Synchronized Sub-Processes

Figure 11. *Two LOTOS Processes Representing Component Structures*

4.2.2 Construction of the Processes

According to the guidelines of Section 4.1, we synthesize the behaviour of each component from the responsibilities that are allocated to them. We need to ensure that the causality relationships defined in the nine UCMs are preserved (this is to be tested at validation time). For each component, we therefore consider all the paths crossing it. Causal relationships across components are refined as exchanges of messages, according to design decisions about protocols and channels to use.

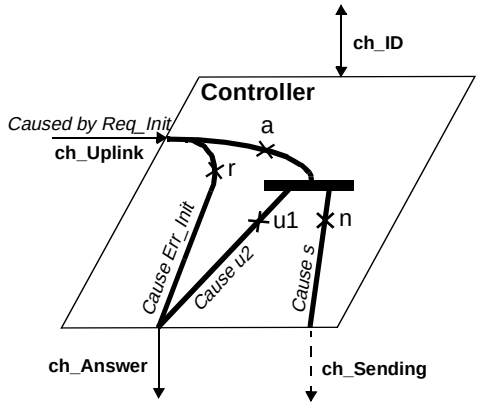
Some large components are specified using sub-processes for coping with the length and inherent complexity. For instance, the process Controller (Figure 12) receives a request from some mobile station and calls the sub-process relevant to this request. There are nine types of request, one for each of PTM-G's functionalities, combined as alternatives.

```
process Controller [ch_Uplink, ch_Answer, ch_Sending, ch_ID] (      (* Databases as parameters. Format is Value_Identifier:Sort *)
                DBZS: GeoMemberList, DBCM: CallMemberList, DBSM: StatusMemberList): noexit:=

  ch_Uplink ?n: M_ID ?req: Req ?msg: Msg;                  (* Get request (req) and encoded parameters (msg) from a MS (n) *)
    (
      [req eq Req_Init] ->                                 (* Guarded behaviour: Is request an Initiate Call? *)
        ContInit[ch_Uplink, ch_Answer, ch_Sending, ch_ID] (DBZS, DBCM, DBSM, n, GetIMGI(msg), GetDTM(msg),
                                                GetQoS(msg), GetGeo(msg), GetJoin(msg), GetSend(msg), GetCallnot(msg))
      []
      [req eq Req_Join] ->                                 (* Guarded behaviour: Is request a Join Call? *)
        ContJoin[ch_Uplink, ch_Answer, ch_Sending, ch_ID] (DBZS, DBCM, DBSM, n, GetIMGI(msg), GetC_ID(msg))
      []
        ...                                                (* Seven remaining functionalities described below... *)
    )
endproc (* process Controller *)
```

Figure 12. *Process Controller Can Instantiate Sub-Processes*

The sub-process responsible for our Initiate Call example (ContInit) is constructed from the responsibilities and causal relationships that are allocated to the **Controller** component in the UCM of Figure 8a. Figure 13a emphasises the relevant path segments and the causality chain. Figure 8b indicates that a, r, and n are conditions, while u1 is some internal activity to be performed. With such information in hand, an informal interpretation of ContInit's behaviour can be described. Figure 13b presents this interpretation with LOTOS-like pseudo-code, but of course this is only one possible translation of Figure 13a. In fact, such interpretation is a transitory step that may be skipped. The real specification of ContInit is included in Appendix B. In this process, there is no reference to Req_Init because this part has already been taken care of by the calling process Controller (Figure 12).



(**a**) Path Segments of Call Initiation in Controller

```
ContInit =
  Caused by Req_Init on ch_Uplink; (
    [a]-> (
      (
        Perform u1; Cause u2 on ch_Answer
        | | |
        [n]-> Cause s on ch_Sending
      )
      Reinstantiate Controller
    )
    []
    [r]-> (
      Cause Err_Init on ch_Answer
      Reinstantiate Controller
    )
  )
```

(**b**) Informal Interpretation of ContInit

Figure 13. *Construction of Sub-Process ContInit*

The reader acquainted with LOTOS will also observe the presence of ContSend in Appendix B. It is used by ContInit as a mean to send indications (by causing s via ch_Sending) to the list of members subscribed to the group call. ContSend iterates over the provided list; this is how we represent loops in LOTOS. ContSend is not local to ContInit because other processes that need to send indications reuse it.

Eight other sub-processes similar to *ContInit* were created for the other types of request in the controller, some of which composed of their own sub-processes. The synthesis resulted in a LOTOS specification with 1140 commented lines of code for the data types and 1400 lines for the behaviour part (i.e. the processes, including those related to the component structure).

5. VALIDATION

5.1 LOTOS Testing and UCMs

A good overview of verification and validation techniques available for LOTOS can be found in [7]. In this experiment, we were not able to use a formal approach for the generation of test cases. The main reason is that our interest resides primarily in validation testing, where the main source of information for test derivation are informal requirements, rather than in testing for conformance with respect to an existing formal specification [23]. Functionality requirements are not given formally in the standard. Hence, methods based on automata models or on canonical testers were not appropriate. We took the UCMs as representations of the requirements and we implemented a pragmatic type of testing for the validation of the services and for the detection of logical errors, ambiguities, inconsistencies, and undesirable interactions.

We derive validation test cases from UCMs (step ⑤ in Figure 2). 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 an idea similar to the white-box approaches used for sequential programs [32]. 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. 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 UCMs. However, this list is in no way complete as other UCM constructs exist, and other patterns based on combinations of constructs can also be defined.

As a testing tool, we use LOLA (LOtos LAboratory), which is a state exploration tool for the simulation and testing of LOTOS specifications [32]. Test cases are LOTOS processes derived from scenarios, and they can be composed with the specification to detect possible errors (step ⑥ in Figure 2). LOLA analyses the test terminations for all possible evolutions, also called *test runs*. If the number of test runs is too large or even infinite, LOLA can use equivalence relations and coverage heuristics to check a representative subset of the possible evolutions. The successful termination of a test run consists of reaching a state where the termination event (e.g. Success) is offered. A test run does not terminate if a deadlock or internal livelock is reached.

For each selected abstract sequence of events/responsibilities (UCM routes), *acceptance* test cases (whose expected verdict is *Must pass*) and possibly *rejection* test cases (whose expected verdict is *Reject*) are generated. Acceptance test cases are well known in the formal theory of *conformance testing* [26], where the goal is to test the conformance of an implementation to its formal specification. However, conformance testing does not test for robustness, which means rejection of certain test cases. This aspect is important for validation testing where we test the specification itself, according to expected functionalities and functional requirements expressed in the UCMs. Therefore, we combine rejection test cases with acceptance test cases. Rejection test cases should not be too prescriptive as to what should be rejected. Automated methods to derive them are still under development.

Our sample Join Call scenario is reused again in Figure 14 in order to illustrate the derivation of a set of test cases with the goal of covering all paths in the UCM. The most suited testing pattern is "Alternative-All paths", as the alternative (OR-Fork) is the only construct found in the UCM that is relevant to test selection. Each selected route then becomes an abstract sequence (also called *test purpose*) that will be translated into a LOTOS test process, while considering the observable messages and data types defined during the synthesis. In this example, the assumption is that the communication channels are hidden (see Figure 9a), so they cannot be tested.

The two rejection test cases are generated from the abstract sequence where a mutation is applied on the last event. This simple strategy aims to detect unexpected results for a specific input. For instance, the specification under test could offer both ind and err after the sequence join; vrfy; updjoin; and this problem could not be detected by the acceptance test case Test1A alone. Other more appropriate strategies can be used for defining rejection test cases for more complex specifications.

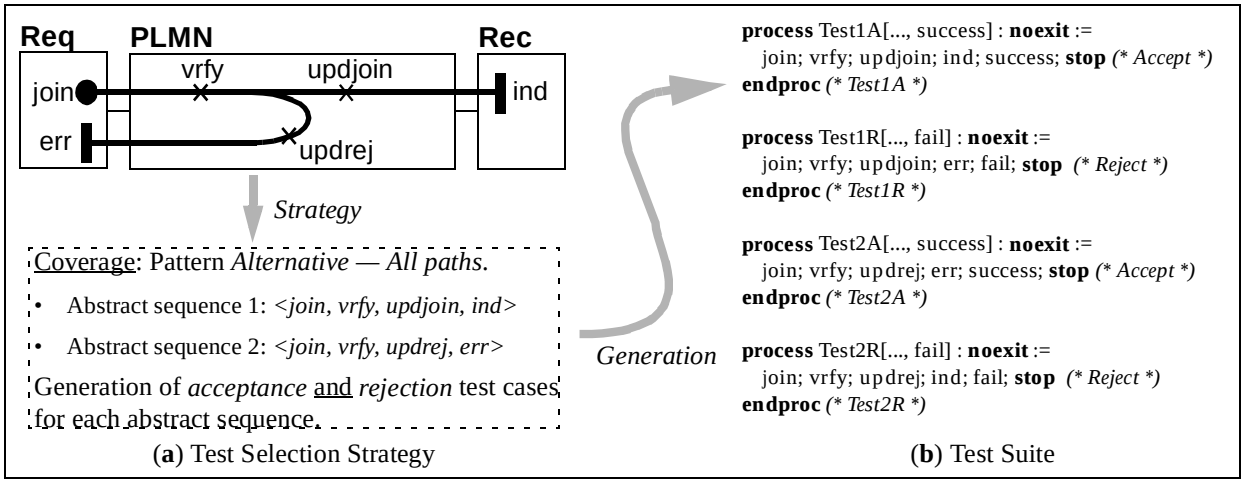


Figure 14. *Derivation of Validation Test Cases from UCMs*

5.2 Testing of GPRS Group Call

Our goal is to validate the specification, synthesized in Section 4.2, against the functional requirements by using a test suite derived from the UCMs. Because the specification involves databases and possibly multiple mobile stations, we first discuss the initial configuration we chose for our system under test. Then we present some test sequences and the results of their execution.

5.2.1 Test Configuration

For complex specifications that involve data and dynamic instantiation of processes, it is best to carefully prepare system configurations that will be used as contexts for the execution of the test suite. Their goal is to enable the satisfaction of the preconditions of all the UCM paths selected as test purposes (i.e. the abstract sequences). A configuration can be built once these test purposes are known, after the selection strategies and testing patterns have been applied to the UCMs.

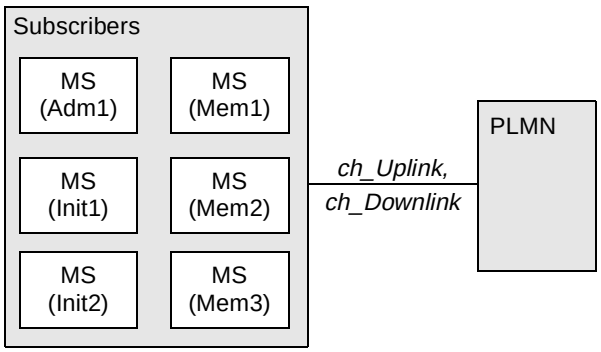


Figure 15. *Test Configuration*

Examination of our nine UCMs revealed that a single configuration with six mobile stations is sufficient for allowing all our test purposes to be fulfilled. They are regrouped in the process Subscribers in Figure 15 and they are instantiated with the following access privileges:

- *Adm1* is an administrator who has administrative privileges over groups and membership.
- *Init1* and *Init2* are initiators who are allowed to initiate and terminate group calls.
- *Mem1*, *Mem2*, *Mem3* are plain group members, who have no other special privileges but to receive and send messages.

These six mobile stations are all initially detached from GPRS. The five databases (in **Informer**: DBCI and DBGC; in **Controller**: DBCM, DBZS, and DBSM) are initialized with sufficient information to eventually satisfy all the preconditions associated to the UCMs.

Our test processes are to be composed with this system, i.e. the subscribers and the PLMN. However, it is also possible to test the PLMN alone for robustness. By removing the subscribers, we also remove several constraints (e.g. synchronizations on ch_Uplink and ch_Downlink) on how the PLMN can react. It is then possible to observe, to some extent, whether or not the PLMN can compensate for faulty behaviour from mobile stations.

5.2.2 Test Derivation and Execution

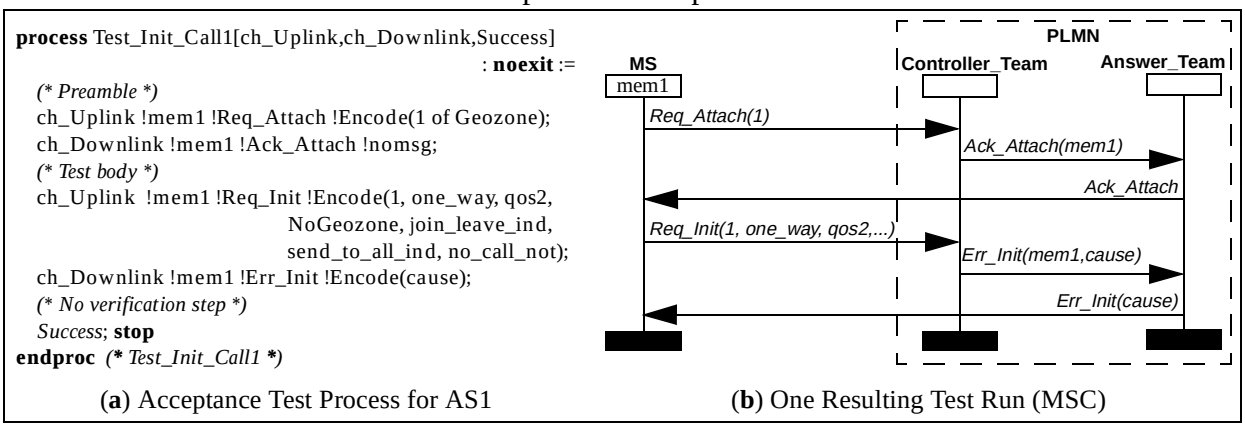
We first choose a test selection strategy for the Initiate Call scenario. As discussed in Section 5.1, we apply the testing patterns “Alternative — All Paths” for OR-Forks and “Concurrent — One combination” for AND-Forks in the UCM of Figure 8a. This means that we want to cover at least all branches in alternatives and at least one specific temporal sequence of events for concurrent paths. This can be done with the two abstract sequences introduced in Section 3.3 (AS1=<Req_Init, r, Err_Init> and AS2=<Req_Init, a, u1, u2, n, Ack_Init, s, Ind_Init>), plus another one

for the case where condition n is not satisfied (no call notification sent to receivers): AS3=<Req_Init, a, u1, u2, Ack_Init>.

Since conditions a and r are complex (see Section B), we can have several test cases covering the same abstract sequence, but with different data values. We defined three acceptance test cases for AS1, each of which corresponds to one of the reasons why an Initiate Call request may be refused. We derived only one acceptance test case for AS2. Although we chose to make limited use of rejection test cases in our experiment, we defined both one acceptance and one rejection test case for AS3. Consequently, the *test group* for Initiate Call contains six test cases.

A typical test case includes a preamble, a test body (which refines the test purpose), and optional verification steps. Preambles are sequences of events that satisfy the preconditions of a scenario under test. For instance, to test the Initiate Call operation, we must first have a sequence of events ensuring that the requester gets attached to the GPRS network. Verification steps are sequences of events that check some aspect of the system's current state. At the end of a test for Initiate Call, the Call Status operation could be invoked to verify that the group call was correctly initiated in the PLMN.

In Figure 16a, process Test_Init_Call1 is one of our three black-box acceptance test cases generated from AS1. It checks that the Initiate Call request is rejected when the mobile station does not have sufficient privileges. According to the configuration of Figure 15, the database **DBSM** initially contains information stating that *mem1* is not an initiator. As a preamble, *mem1* has to attach to GPRS. We included no verification step for this simple test.



(**a**) Acceptance Test Process for AS1 (**b**) One Resulting Test Run (MSC)

Figure 16. *Process Test_Init_Call1 and An Execution Trace (MSC)*

LOLA checks that all possible executions of this test terminate successfully. One possible temporal trace is illustrated as an MSC in Figure 16b, where some components and internal messages within the PLMN (e.g. between **Controller_Team** and **Answer_Team**) are shown.

A more complex test case, defined for AS2 (with an additional group member), is presented in Figure 17a. This time, verification steps check that the call initiation resulted in the appropriate indications and database updates. This black-box test does not have direct access to the internal content of the PLMN databases, but it makes use of the Call Status request to increase our confidence in the accuracy of the verdict.

Due to interleaving of events and to internal non-determinism in the specification, LOLA generated a total of 2594 test runs for Test_Init_Call4. Figure 18 illustrates one of these executions. Again, every message sent and received within the PLMN is locally hidden while the messages

between the subscribers and the PLMN are observable and controllable by the test process. The trace generated by LOLA contains more internal messages between the sub-components of **Controller_Team** and **Answer_Team**, but they are not included here in order to simplify and shorten the resulting MSC.

```
process Test_Init_Call4[ch_Uplink, ch_Downlink, Success]
                        : noexit :=
  (* Preamble *)
  ch_Uplink !init1 !Req_Attach !Encode(1 of Geozone);
  ch_Downlink !init1 !Ack_Attach !nomsg;
  ch_Uplink !mem1 !Req_Attach !Encode(1 of Geozone);
  ch_Downlink !mem1 !Ack_Attach !nomsg;
  (* Test body *)
  ch_Uplink !init1 !Req_Init !Encode(1, n_way, qos1, ...,
                                     call_not_ind);
  ch_Downlink !init1 !Ack_C_Init !Encode(1, succ(noc_id),
                                    cipher_key(succ(noc_id)));
  ch_Downlink !mem1 !Ind_Init !Encode(1, succ(noc_id));
  ch_Downlink !init1 !Ind_Init !Encode(1, succ(noc_id));
  (* Verification steps*)
  ch_Uplink !mem1 !Req_Status ! Encode(1, noc_id,
                                       Insert(sta_callids,{}));
  ch_Downlink !mem1 !Ack_Status !Encode(1,
                                    Insert(succ(noc_id),{}));
  Success; stop
endproc (* Test_Init_Call4 *)
```

(**a**) Acceptance Test Process 4 for AS2

```
process Test_Init_Call6[ch_Uplink, ch_Downlink, Reject]
                        : noexit :=
  (* Preamble *)
  ch_Uplink !init1 !Req_Attach !Encode(1 of Geozone);
  ch_Downlink !init1 !Ack_Attach !nomsg;
  (* Test body *)
  ch_Uplink !init1 !Req_Init !Encode(1, n_way, qos1, ...,
                                     no_call_not);
  ch_Downlink !init1 !Ack_C_Init !Encode(1, succ(noc_id),
                                    cipher_key(succ(noc_id)));
  (* Next event should be rejected by the system *)
  ch_Downlink !init1 !Ind_Init !Encode(1, succ(noc_id));
  (* No verification step *)
  Reject; stop
endproc (* Test_Init_Call6 *)
```

(**b**) Rejection Test Process 6 for AS3

Figure 17. *Processes Test_Init_Call4 and Test_Init_Call6*

While all the acceptance test cases for Initiate Call resulted in a *Must Pass* verdict (as expected), LOLA pronounced a *Reject* verdict for the rejection test case of Figure 17b. This last test is unable to reach the Reject event, as expected, because the initiator (*init1*) is expecting a call initiation indication (Ind_Init) while the PLMN was required not to send any (**no_call_not**). If the Reject event had been reached by some test run, then the verdict would have been different (*May Pass* or *Must Pass*) and this would have proved that the PLMN is not specified properly.

5.2.3 Testing Results

We applied similar test selection strategies to the nine UCMs in our system. This resulted in nine test groups, one for each UCM, for a total of 36 test cases and 800 lines of LOTOS code. As expected, our test cases led to incorrect traces that were used to diagnose logical errors in the specification. Most of them were due to the following problems:

- Conditions (guards) that were incomplete or ambiguous. Some test cases resulted in a *Reject* verdict when selected values were not covered by any of the conditions. Others resulted in a *May Pass* verdict when selected values satisfied simultaneously two conditions that were expected to be mutually exclusive.
- Infeasible synchronizations between processes, caused by incompatible data types or value offers, especially in the context of a multiway rendezvous.
- Incorrect definition of abstract data types. ADTs are not tested as such by the scenario-based approach, and hence their definition is prone to errors.

After several iterations in SPEC-VALUE, which included appropriate modifications to the UCMs, specification, and tests (step ⑧ in Figure 2), LOLA successfully executed all the 36 test cases in 3 minutes on a Sparc Ultra 1. Each test covered hundreds or thousands of test runs. In our experience, this level of performance is pragmatic enough for SPEC-VALUE to be used in an iterative and incremental design process where numerous modifications, additions, debugging sessions, and executions of regression test suites need to be supported.

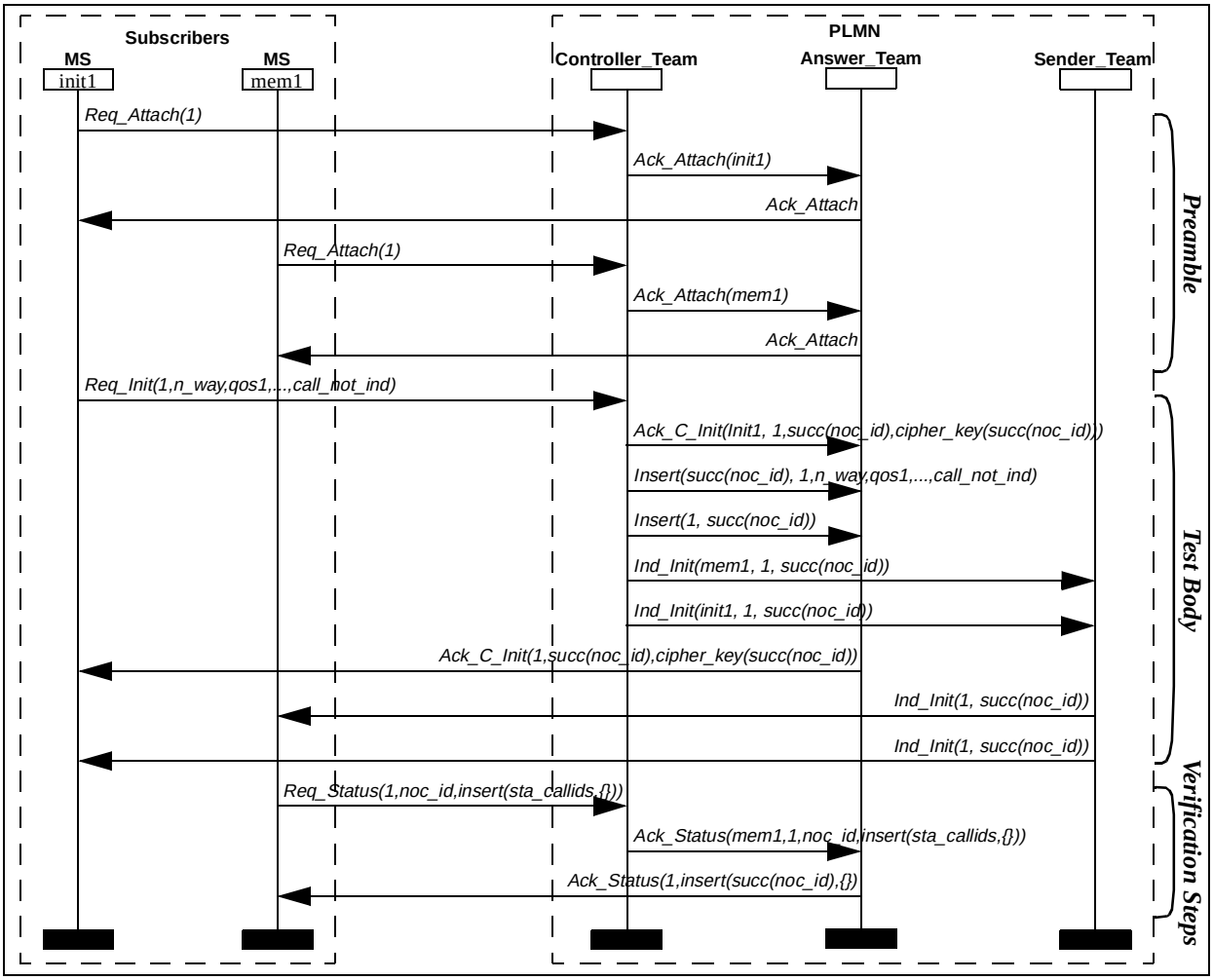


Figure 18. *An Execution Trace (MSC) for Process Test_Init_Call4*

Following the testing of the PLMN in composition with six subscribers, the PLMN was retested with the same test suite, only this time without any subscribers. The testing of the PLMN in such an unconstrained environment aims to ensure the robustness of PLMN and to check that its design does not depend on the assumption that the subscribers are well-behaved. Since the PLMN was already debugged, and since we had already augmented its behaviour with several features for fault tolerance, we noticed no difference in the test verdicts. This improved our confidence in the accuracy of the PLMN, although security problems and others may still exist.

5.3 Coverage Measurement

5.3.1 Structural Coverage with Probes

Once inconsistency errors between the LOTOS specification and the UCM-based tests have been detected are corrected, we consider the *functional coverage* of the requirements to be complete under the hypothesis made in the test selection strategies. In order to verify that the test suite has covered each responsibility syntactically present in the specification, we need a measure of *structural coverage*.

Probe insertion is a well-known white-box technique for monitoring software and measuring structural coverage in order to identify portions of code exercised [33]. 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. We insert probes at strategic places in the behaviour expressions of the LOTOS specification (before **stop**, **exit**, and process instantiations) that are candidates for being leaves in their respective labelled transition systems. They have the property of covering causal sequences of events in the specification. Our probes are LOTOS internal events with unique identifiers, hence they neither affect the observable behaviour of the specification nor the test verdicts.

Problems associated with probes that are not visited by a validation test suite usually fall into one of the following categories:

- **Incorrect specification**. In particular, there could be unreachable code caused by events that cannot synchronize or by guards that cannot be satisfied.
- **Incorrect test case**. Premature deadlocks are usually detected before probes are inserted, during the verification of the functional coverage.
- **Incomplete test suite**. Caused by an untested part of the specification (e.g. a functionality of the specification that is not part of the original requirements). New test cases may be necessary (step ⑦ in Figure 2).
- For our scenario-based approach, there could be some **discrepancy between a UCM and the specification** caused by ADTs, guards, and choice operators.

Code inspection and step-by-step execution of the specification can help diagnosing the source of the problem highlighted by a missed probe.

5.3.2 GPRS Results

We inserted 71 probes in the specification for the PLMN and 30 others for the MS process. The comments *(*_PROBE_*)* in Figure 9b and *(*_PROBE_PLMN_*)* in Appendix B are instances of probes inserted in specifications. An in-house translator automatically creates a new and equivalent specification where these comments are transformed into hidden gates with unique identifiers.

Since we had added features for improving the robustness of the PLMN, we expected to have seven probes unvisited as a result of events that should not happen in the normal use of the system. Because of obscure and ambiguous points in the requirements, we had to make some design decisions at the specification level. The coverage of the PLMN process alone highlighted an unvisited probe corresponding to a portion of the LOTOS code (in the process DBGC) that was useless. It was removed from the specification. The remaining 62 probes were visited as expected. Similarly, we expected three probes in the MS definition not to be reached by our test suite (they were part of additional code for robustness). Indeed, the remaining 27 probes were covered as planned.

The test suite could be augmented with new tests for the coverage of the 10 unvisited probes (step ⑧ in Figure 2). Through step-by-step execution of PLMN and MS taken individually, we were able to visit these probes, hence proving that the code is not unreachable. These traces can easily be transformed into test sequences, derived solely from the specification. However, since these tests would have nothing to do with the requirements (as they relate more to our design decisions for improving robustness), we do not include them in our validation test suite.

This example, where we first checked the probes in PLMN and then the ones in MS, shows that we do not have to cover all the probes at once to get meaningful results. Since probes do not affect the observable behaviour of the specification, we can use a compositional coverage of the structure. Probes can be covered independently, and one could even do this one probe at a time. This helps avoiding the state explosion problem caused by the presence of additional internal events in compositions.

6. DISCUSSION

6.1 Problems Detected in the PTM-G Service

The application of SPEC-VALUE to PTM-G raised many questions about the standard [16]. Multiple errors, inconsistencies, and ambiguities were unveiled in many steps of our approach, especially in the requirement capture, formal specification of causal scenarios, validation testing of the prototype, and measure of structural coverage. Some of the problems that were uncovered, for the part of the system considered in this paper, include:

- Sending of indications: for the successful execution of operations Join Call and Leave Call, it is not clear whether an indication is sent to all participants or not. We decided to allow for both cases in our specification by adding a parameter to the relevant requests.
- Rejection cause: when a rejection is provided (e.g. for a Call Initiation request), the end-user receives an ambiguous answer that can be interpreted in many ways, making it difficult to diagnose the reason of the rejection.
- Restrictions of joining calls: in a Join Call operation, there is no restriction concerning the calls which a subscriber could ask to join. This potential flaw in the design raises many security and privacy issues. We decided to constrain the use of this operation in our prototype.

In most cases, the descriptions are operational and supported by informal figures in the standard. Nevertheless, they represent only partial scenarios, and no system view of the functionalities

is provided. Improving these descriptions with UCMs would represent a good step towards avoiding problems similar to those enumerated above.

6.2 Advantages of SPEC-VALUE

At the end of Section 2.2, we claimed that this approach has four main advantages. Without repeating the explanations of the first three, we indicate the sections of this paper in which they are best illustrated. Sections 2.3.3, 2.3.4, and 3.3 discuss the separation of the functionalities from the underlying structure. The fast prototyping of systems and the generation of test cases are covered in Sections 4 and 5 respectively.

The fourth advantage, design documentation, is illustrated by several figures. UCMs (Figure 8) convey important functional information about the system. Through experiences shared with many industrial partners and university students, we know that the end-to-end system behaviour (the *big picture*) is easier to understand with a Use Case Map than with a set of interaction diagrams that describe detailed design (e.g. MSCs like Figure 18). This is particularly true for non-technical people. Designers and testers can refer to more detailed descriptions of the system in the form of formal specifications (Appendix B) or of execution traces generated by the use of tools (MSCs).

On a number of occasions, we gave our documentation to beginners and they found that they could understand it easily. For instance, the first version of the Group Call example presented above was mainly done by an undergraduate student (P. Forhan), who initially was not familiar with any of GPRS, LOTOS and its tools, UCMs, or this approach. Nevertheless, it took him less than 5 months to gain an understanding of the Group Call service and to produce useful documentation, concise and descriptive scenarios, a validated specification, and a test suite in which we have a high level of confidence. He based his work on an earlier study of a Group Communication Server [2], for which the UCM scenarios were generic enough to be reused for the design of the PTM-G service. Since the structures of these two systems are not alike, this reuse of scenarios would have been more difficult with Message Sequence Charts. Other anecdotal evidence that UCM paths are reusable can be found in [12].

SPEC-VALUE promotes *traceability* among the models it uses, which also improves the quality of the documentation. UCMs represent functional requirements. Through the UCMs, responsibilities, components, and scenarios can be traced to one another. In our prototype, there is clear correspondence between processes and the component structure, and between UCM responsibilities and LOTOS events. Traceability is also possible in the validation steps. For instance, abstract sequences of events can be linked to their respective UCMs through the selection strategies. Traceability relationships also exist between test cases and abstract sequences, and between test runs and test cases.

Another benefit of this approach is that it permits independent teams to work simultaneously on different system functionalities. Similarly, tests can be derived concurrently with the specifications. According to current software engineering practices, this is often desirable [32].

Functionalities can also be added incrementally to the specification. In this GPRS case study, we started to integrate and validate the operations that were independent of the others, followed by the operations whose dependencies were already specified (in order: *Attach GPRS*, then *Detach GPRS*, then *Initiate Call*, then *Call Status*, then *Join Call*, *Data Transfer* and *Change Zone*, then

Leave Call and *Terminate Call*). With UCMs, adding new functionalities when the structure is stable is no more difficult than doing it with scenarios based on components and message exchanges. However, when a new functionality requires modifications to the structure, the impact appears to be less strong when UCMs are involved. The scenario paths can be easily adapted to the new structure by reallocating the responsibilities (as in Figure 5), hence providing a traceable link to the components/processes in the specification whose behaviour needs to be adapted. Doing this remodeling requires more efforts with scenarios based on MSCs or the like. Again, we have observed this when specifying this system (we made several modifications to the initial GPRS abstract structure we considered), but also in other application of SPEC-VALUE [2][5].

A last noticeable benefit is the minimization of the modeling effort, as UCMs are used for guiding both the synthesis of the specification and the generation of test cases.

6.3 Comparison with Other Approaches

We relate this work to the following five methodologies based on scenarios and/or LOTOS:

- **Timethreads-LOTOS**: SPEC-VALUE builds on previous work on the formalization of *Timethreads* (a former version of UCMs) in LOTOS [1]. The Timethreads-LOTOS technique addresses the specification of UCM paths only. The system end-to-end functionalities are not distributed over component structures, hence resulting specifications are simpler and more abstract. Many issues related to the implementation of causality chains across components are not addressed. Our new approach addresses these issues, and it also allows for alternative structures to be evaluated and for validated prototypes and test suites to be generated.
- **LOTOSphere**: The LOTOSphere project [7] was an international effort to formulate an integrated methodology for the development of communications software. LOTOSphere based itself on LOTOS and correctness-preserving algebraic transformations. Requirements were to be written in LOTOS constraint-oriented style, which subsequently would be transformed into state-oriented and resource-oriented styles for implementation [42]. Other transformations of the specifications would provide test cases. The obvious advantage of this approach is that one could hope that these transformations would be implemented in tools, thus guaranteeing conformance throughout the whole process. Unfortunately, it turned out that constraint-oriented specifications are difficult to write, especially because most designers seem to be more used to think in terms of scenarios than in terms of constraints. The algebraic transformation approach could not be practically applied to realistic specifications, and algebraic transformation tools were not made available. We believe that SPEC-VALUE, even if limited to the first stages of design, has a better chance to be applied in industry.
- **SDL and MSCs**: This is one of the most popular approaches, for which industrial support exist and powerful tools are available [3]. MSCs represent scenarios visually in terms of sequences of messages exchanged between system components. They guide the generation of component behaviour, which can be formally specified and validated with the *Specification and Description Language* (SDL) [24]. Multiple methodologies based on the combined use of SDL and MSCs (or similar types of scenarios) exist. For instance, Eberlein proposes the *RATS service development methodology* [15], which covers requirements engineering tasks from use cases to the first service specification in SDL. Regnell proposes the *Usage Oriented Requirements Engineering* approach [34], where use cases are represented in structured text and scenarios are for-

malized in a MSC-like notation (this is an extension of Objectory's *Use Case Driven Analysis* [27]). However, components and messages need to be identified early in the design process for both MSCs and SDL. SDL also requires the definition of states for the components. Premature decisions often need to be taken on the sole basis of informal requirements. Moreover, the system view of functionalities and causal relationships between activities tend to be hidden behind clouds of details, especially as the scale of the system increases.

- **Unified Modeling Language**: UML 1.3 includes sophisticated diagram notations [40], among which we find *use case diagrams* [27], which show relationships among prose descriptions of behaviour. These diagrams present the interactions between actors and the system in a black-box fashion, and they do not help much in visualizing system behaviour or causality relationships. *Interaction diagrams* (similar to MSCs) and *statechart diagrams* do focus on behaviour, but at a detailed design level that include actors (our components), states, and messages. Unfortunately, these component-based diagrams suffer from problems similar to those of SDL and MSCs. *Activity diagrams* can illustrate causality between events, but at a level closer to procedures than to systems (and activities are usually unrelated to components, except through the limited *swimlines*). Furthermore, UML is a modeling notation and is not too concerned with formal validation of prototypes and formal generation of test suites, while SPEC-VALUE focuses on these two aspects.
- **ROOA**: In their *Rigorous Object-Oriented Analysis* (ROOA) method [30], Moreira and Clark integrate formal techniques with object-oriented analysis methods (i.e. OMT [36]) in order to generate executable prototypes (in LOTOS) and validate them against the requirements. ROOA considers the system as a set of concurrent objects, modelled by LOTOS processes. The method starts by identifying the object model, then scenarios are created to model the system dynamics. Scenarios are, again, sequences of interactions between the objects, just like the SDL/MSC approach. Validation is done through simulation, testing, and symbolic execution. However, regrouping OMT classes into system components for the specification (i.e. going from a static model to a more dynamic and functional one) remains difficult. Unlike SPEC-VALUE, ROOA does not really develop any strategy for the validation or the derivation of test cases.

Our design approach does not attempt to replace other methodologies. For instance, UML covers many steps and concerns of the software development cycle that are not addressed here. However, integrating the ideas and techniques developed in this paper with existing methodologies may prove to be a good investment.

6.4 Research Directions

The outcome of this experiment has attracted the attention of several telecommunication companies. We have already started a joint project (with a large Canadian telecommunication company) where this approach is applied to phases I and II of the *Wireless Intelligent Network* (WIN), a forthcoming North-American standard for mobile telephony of the *Telecommunications Industry Association* (TIA) [39]. ETSI and TIA have similar standardization processes, where each phase (introduction of new features and functionalities) is based on three major stages. In stage one, the general functionality of the system is being studied. The focus of stage two is at the level of MSCs, while level three details protocols and procedures. UCMs fit best in stage one, and the use of a LOTOS prototype paves the way towards stage two, where MSCs start being constructed. In [17],

Ghribi *et al.* report on how LOTOS was used with success in the specification and validation of other parts of the GPRS standard at different stages.

This and other similar experiments have initiated an industrial uptake of UCMs and of our approach. Another Canadian telecommunication company is currently using UCMs for describing and prototyping self-modifying agent systems [11]. Following another experiment on the application of this approach to the complex problem of detecting and avoiding undesirable interactions between telephony services [5], this company has started to use UCMs for describing a new generation of IP-based PBXs containing several hundred features. They plan to prototype parts of their system with LOTOS and to validate it using UCM-based testing.

Apart from exploring these applications, we intend to extend our work in several directions, including:

- Use of message exchange patterns. We would like to provide designers with a collection of pre-defined message exchanges or negotiations in order to formalize the refinement of inter-component causality relationships during the synthesis step.
- Formalization of test purpose derivation from UCMs. As stated at the beginning of Section 5, our test selection strategies and testing patterns are still informal, although they seem pragmatic. By removing the component structure, individual UCMs can easily be translated into LOTOS [1], thus enabling the generation of canonical testers. According to LOTOS testing theory [9], we can derive sequences and sub-trees from this canonical tester (called *reductions*), and those would become the test purposes.
- Extensions to UCMs in order to express mandatory and forbidden causal sequences. As of now, only the former ones are captured as UCMs. Forbidden sequences could influence design decisions made during synthesis, and they could be used as a basis for the generation of rejection test cases.

Therefore, the future of the combined UCM-LOTOS methodology is promising. A user group has been initiated [38] to support UCMs in their evolution. Already, companies and researchers discuss the best way of integrating them with the UML methodology [8]. Tools have started to appear and a graphical UCM editor (UCM Navigator) is already available for several platforms [28]. The UCMs internal format is based on hypergraphs and the file format is in XML (*eXtensible Markup Language*). Unfortunately, tools to support translation to LOTOS are not available yet.

7. CONCLUSION

We have demonstrated an iterative and incremental design approach, based on a visual scenario notation (Use Case Maps) and a formal description technique (LOTOS). In this approach, causal scenarios, described as UCMs, guide the synthesis of prototypes and the generation of validation test cases specified in LOTOS. The main features of UCMs were illustrated with a simplified PTM-G operation (Join Call) and a real operation (Initiate Call). Although we touch lightly upon this aspect in this paper, the approach helped us unveil several ambiguous and incomplete descriptions in the stage 1 standard document of the GPRS Group Call service [14].

Based on past and current projects involving an artificial Group Communication Server [2], avoidance and detection of feature interactions [5], and TIA's forthcoming Wireless Intelligent Network standard [39], we observed several interesting points. Scenarios described as UCMs

focus on causality instead of message exchanges, and consequently they can be developed independently of the underlying structure. Such scenarios tend to be highly reusable, sometimes even among different systems. Design documentation is improved by the inclusion of concise and descriptive scenarios and of formal specifications, and by enhanced traceability between the different models and constructs involved in our approach. According to guiding rules, design decisions, and test selection strategies, prototypes and test cases are generated from the scenarios. Prototypes and test suites can further be validated through probes inserted in specifications and structural coverage measurement.

We believe that SPEC-VALUE can facilitate the early stages of requirements and systems engineering within industrial design processes. It can be useful also in the later stages, because our scenarios are useful throughout, as global documentation and as functional test cases. We intend to improve the approach in many ways, including the use of message exchange patterns for specifying inter-component causality relationships, the formalization of test purpose derivation from UCMs, and extensions to UCMs in order to guide the generation of rejection test cases.

8. ACKNOWLEDGMENTS

We are indebted towards Pascal Forhan for his work on the Group Call UCMs and LOTOS specification. We thank the University of Ottawa LOTOS research group (especially Jacques Sincennes, Brahim Ghribi, Laurent Andriantsiferana, and Neil Hart) for their usual yet very appreciated collaboration, and Ray Buhr for his work on Use Case Maps. We kindly acknowledge FCAR, NSERC, CITO, Motorola, Nortel, and Mitel for their financial support.

9. REFERENCES

[1] Amyot, D., Bordeleau, F., Buhr, R.J A., and Logrippo, L. (1995) “Formal support for design techniques: A Timethreads-LOTOS approach”. In: von Bochman, G., Dssouli, R., and Rafiq, O. (Eds.), *FORTE VIII, 8th International Conference on Formal Description Techniques*, Chapman & Hall, 57-72. http://www.csi.uottawa.ca/~damyot/phd/forte95/forte95.pdf

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

[3] Amyot, D., Andrade, A., Logrippo, L., Sincennes, J., and Yi, Z. (1999) “Formal Methods for Mobility Standards”. *IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems*, Dallas, USA, April 1999.

[4] Amyot, D. and Andrade, R. (1999) “Description of Wireless Intelligent Network Services with Use Case Maps”. In: *SBRC’99, 17th Brazilian Symposium on Computer Networks*, Salvador, Brazil, May 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/sbrc99.pdf

[5] Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L. (1999) “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/UseCaseMaps/pub/re99.pdf

[6] Ardis, M.A., Chaves, J.A., Jagadeesan, L. J., Mataga, P., Puchol, C., Staskauskas, M.G., and Olnhausen, J.V. (1996) “A Framework for Evaluating Specification Methods for Reactive Systems — Experience Report”. In: *IEEE Transactions on Software Engineering*, 22 (6), 378-389.

[7] Bolognesi, T., van de Lagemaat, J., and Vissers, C. (1995) *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, The Netherlands.

[8] Bordeleau, F. and Buhr, R.J.A. (1997) “The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines”. *Conference on the Engineering of Computer-Based Systems*, Monterey, USA, March 1997. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM-ROOM.pdf

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

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

[11] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S. (1998) "High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example". In: H.S. Nwana and D.T. Ndumu (Eds), *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998, 277-295. http://www.UseCaseMaps.org/UseCaseMaps/pub/4paam98.pdf.

[12] Buhr, R.J.A. (1998) "Use Case Maps as Architectural Entities for Complex Systems". In: [20]. http://www.UseCaseMaps.org/UseCaseMaps/pub/tse98final.pdf

[13] Cockburn, A. (1997) "Using Goal-Based Use Cases". In: *JOOP*, November/December 1997, 56-62.

[14] Courtiat, J.-P., Dembinski, P., Holzmann, G.J., Logrippo, L., Rudin, H. and Zave, P. (1996) "Formal methods after 15 years: Status and trends — A paper based on contributions of the panelists at the FORmal TEchnique '95 Conference, Montreal, October 1995". In: *Conputer Networks and ISDN Systems*, 28, Elsevier Science B.V., 1845-1855.

[15] Eberlein, A. and Halsall, F. (1997) "Telecommunications Service Development: A Design Methodology and its Intelligent Support". In: *Journal of Engineering Applications of Artificial Intelligence*, 10, (6).

[16] ETSI (1996), Digital Cellular Telecommunications System (Phase 2+); *General Packet Radio Service (GPRS); Service Description Stage 1 (GSM 02.60), Version 2.0.0* (November 1996).

[17] Ghribi, B. and Logrippo, L. (1999) " Prototyping and Formal Requirement Validation of GPRS: A Mobile Data Packet Radio Service for GSM" . In: *Seventh International Conference on Dependable Computing for Critical Applications (IFIP/IEEE)*, San Jose, CA, 99- 118.

[18] Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Prentice-Hall International, U.K.

[19] Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. and Chen, C. (1994) "Formal Approach to Scenario Analysis". *IEEE Software*, March 1994, 33-40.

[20] IEEE (1998), *Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998.

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

[22] ISO/ITU-T (1995), *Open Distributed Processing, Reference Model*, ISO 10746, ITU Recommendation X.901-904, Geneva.

[23] ISO/EIC (1996), Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing" (FMCT). ISO JTC1/SC21/WG7, ITU-T SG 10/Q.8, CD-13245-1, Geneva.

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

[25] ITU (1995), *Q.1200 General Series, Intelligent Networks Recommendation Structure*. Geneva.

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

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

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

[29] Milner, R. (1989) *Communication and Concurrency.* Addison-Wesley, Reading, Massachusetts,USA.

[30] Moreira, A.M.D. and Clark, R.G. (1996) "Adding Rigour to Object-Oriented Analysis". In: *Software Engineering Journal*, 11(5), 270-280.

[31] Mouly, M. and Pautet, M.-B (1992) *The GSM System for Mobile Communications.* Cell. & Sys.

[32] Pressman, R. S. (1997) *Software Engineering — A Practitioner's Approach*. Fourth edition. McGraw-Hill, USA.

[33] Probert, R.L. (1982) "Optimal Insertion of Software Probes in Well-Delimited Programs", *IEEE Transactions on Software Engineering*, Vol 8, No 1, January 1982, 34-42.

[34] Regnell, B., Kimbler, K., and Wesslén, A. (1995) "Improving the Use Case Driven Approach to Requirements Engineering". In: *Proceedings of Second International Symposium on Requirements Engineering*, York, U.K., March 1995, 40-47.

[35] Regnell, B., and Runeson, P. (1998) "Combining Scenario-based Requirements with Static Verification and Dynamic Testing". In: E. Dubois, A. L. Opdahl, and K. Pohl (Eds.), *Proceedings of the Fourth International Workshop on Requirements Engineering - Foundations for Software Quality (REFSQ'98)*, Pisa, Italy, June 1998.

[36] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991) *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, USA.

[37] Quemada, J., Azcorra, A., and Pavón, S. (1995) "The LOTOSPhere design methodology". In [3], 29-58.

[38] Somé, S., Dssouli, R., and Vaucher J. (1996) "Toward an Automation of Requirements Engineering using Scenarios". In: *Journal of Computing and Information*, 2(1), 1110-1132.

[39] TIA/EIA (1998) *Wireless Intelligent Networks (WIN)*. TR-45.2.2.4, PN-3661 Ballot Version, May 1998.

[40] UML Revision Task Force (1999) *OMG Unified Modeling Language Specification, version 1.3*, June 1999. http://uml.systemhouse.mci.com/

[41] *Use Case Maps Web Page and User Group* (1999). http://www.UseCaseMaps.org

[42] Vissers, C.A., Scollo, G., van Sinderen, M., Brinksma, E. (1991) "Specification Styles in Distributed Systems Design and Verification", *Theoretical Computer Science '89*, pp. 179-206.

[43] Weidenhaupt, K., Pohl, K., Jarke, Matthias, and Haumer, P. (1998) "Scenarios in System Development: Current Practice". In: *IEEE Software*, March/April 1998, 34-45.

APPENDIX A. GLOSSARY OF ACRONYMS

Acronym	Description	Acronym	Description
ADT	*Abstract Data Type*	ITU	*International Telecommunications Union*
ANSI	*American National Standard Institute*	LOTOS	*Language Of Temporal Ordering Specification*
AS	*Abstract Sequence*	LTS	*Labelled Transition System*
CCS	*Calculus of Communicating Systems*	MSC	*Message Sequence Chart*
CSP	*Communicating Sequential Processes*	MS/FS	*Mobile Station / Fixed Station*
DBCI	*Database of Call Information*	ODP	*Open Distributed Processing*
DBCM	*Database of Call Members*	OMT	*Object Modeling Technique*
DBGC	*Database of Group Calls*	PBX	*Private Branch eXchange*
DBSM	*Database of Status Members*	PLMN	*Public Land Mobile Network*
DBZS	*Database of geographical Zone Stations*	PTM-G	*Point To Multipoint - Group Call*
ETSI	*European Telecom. Standards Institute*	PTP	*Point To Point*
FDT	*Formal Description Technique*	ROOA	*Rigorous Object-Oriented Analysis*
GSM	*Global System for Mobile Communications*	SDL	*Specification and Description Language*
GPRS	*General Packet Radio Services*	TIA	*Telecommunications Industry Association*
IMGI	*International Mobile Group Identity*	UCM	*Use Case Map*
IN	*Intelligent Network*	UML	*Unified Modelling Language*
IP	*Internet Protocol*	WIN	*Wireless Intelligent Network*
ISO	*International Organization for Standardization*	XML	*eXtensible Markup Language*

APPENDIX B. SPECIFICATION OF PROCESSES CONT_INIT AND CONT_SEND

```
process ContInit[ch_Uplink, ch_Answer, ch_Sending, ch_ID] ( DBZS:GeoMemberList, DBCM:CallMemberList,
                                                            DBSM:StatusMemberList, n:M_ID, imgi:IMGI, dtm:DTM, qos:QoS,
                                                            zg:GeoZone, join:Join_leave, send:Send_to_all, callnot:Call_not): noexit :=
  (* Process called by Controller, where an Initiate Call request was already received on ch_Uplink *)
  (* First, check the UCM OR-Fork. Is Requester allowed to initiate a call? *)
  let S:StatusMember = RowSM(n, DBSM), Z:GeoMember = RowZS(zg, DBZS) in  (
    [ ((MemSta(S) eq init) or (MemSta(S) eq cont))                          (* Requester has rigths to initiate a call... *)
      and (imgi Isin IMGISta(S))                                            (* ... and the group is valid... *)
      and ((zg eq NoGeozone) or (n Isin M_IDZS(Z))) ] -> (                  (* ... and the geographical zone is valid. *)
      (* Condition "a" satisfied. Cause "u1" and "n" in parallel. *)
        ch_ID?cid:C_ID;                                                     (* New channel identifier needed for responsibilities "u1" and "n" *)
        (
          (
            (* Responsibility "u1": update DBCM and DBSM.  *)
            let newDBCM:CallMemberList = Insert(CM(cid, n, Insert(n, {} of M_IDlist)),DBCM),
                newDBSM:StatusMemberList = Insert(SM(M_IDSta(S), MemSta(S), IMGISta(S), Insert(cid, C_IDSta(S)), AttSta(S)),
                                                  Remove(S, DBSM))
            in
              (* Next three messages will cause "u2" in the Answer_Team via ch_Answer *)
              ch_Answer!n!Ack_C_Init!Encode(imgi, cid, Cipher_key(cid));     (* Info for acknowledgment. *)
              ch_Answer!Insert!cid!Encode(imgi, dtm, qos, zg, join, send, callnot);   (* Info for DBCI. *)
              ch_Answer!Insert!imgi!cid;                                     (* Info for DBGC. *)
              exit(newDBCM, newDBSM)
          )
          |||
          (
            (* Check condition "n": Do we need to notify the Receivers?  *)
            [callnot eq Call_not_ind] -> (      (* Condition "n" satisfied. Cause "s" in Sender_Team  via ch_Sending, one for each Receivers. *)
              ContSend[ch_Sending](M_IDZS(Z), Ind_Init, Encode(imgi, cid))
              >> (*_PROBE_PLMN_*)
              exit(any CallMemberList, any StatusMemberList)
            )
            []
            [callnot eq No_call_not] -> ( (*_PROBE_PLMN_*)                  (* Condition "n" not satisfied. Do nothing. *)
              exit(any CallMemberList, any StatusMemberList)
            )
          )
        )
        >> accept newDBCM:CallMemberList, newDBSM:StatusMemberList in
          (* Reinstantiates the Controller once "u1" and "n" have ended. *) (*_PROBE_PLMN_*)
          Controller[ch_Uplink, ch_Answer, ch_Sending, ch_ID](DBZS, newDBCM, newDBSM )
    )
    []
    [ ((zg ne NoGeozone) and (n Notin M_IDZS(Z)))                           (* The geographical zone is not valid... *)
      or (imgi Notin IMGISta(S))                                            (* ... or the group is not valid... *)
      or ((MemSta(S) ne init) and (MemSta(S) ne cont)) ] -> (               (* ... or the requester does not have rigths to initiate a call. *)
      (* Condition "r" satisfied. Cause "Err_Rejinit" in Requester via ch_Answer. *)
      ch_Answer!n!Err_Rejinit!Encode(Cause);
      (* Reinstantiate Controller without any modifications to the databases *) (*_PROBE_PLMN_*)
      Controller[ch_Uplink, ch_Answer, ch_Sending, ch_ID](DBZS, DBCM, DBSM)
    )
  )
endproc  (* process ContInit *)

process ContSend[ch_Sending](midl: M_IDlist, ind: Ind, msg: Msg): exit:=    (* Loop that sends an indication to a list of members. *)
  [Top(midl) eq (Noelement of M_ID)] -> (*_PROBE_PLMN_*)                    (* No more indication to send. Exit this loop. *)
    exit
  []
  [Top(midl) ne (NoElement of M_ID)] ->                                     (* Cause “s” in Sender_Team (indication to first member in the list). *)
    ch_Sending!Top(midl)!ind!msg; (*_PROBE_PLMN_*)
    ContSend[ch_Sending](Remove(Top(midl), midl), ind, msg)                 (* Iterate with the rest of the list. *)
endproc (* process ContSend *)
```