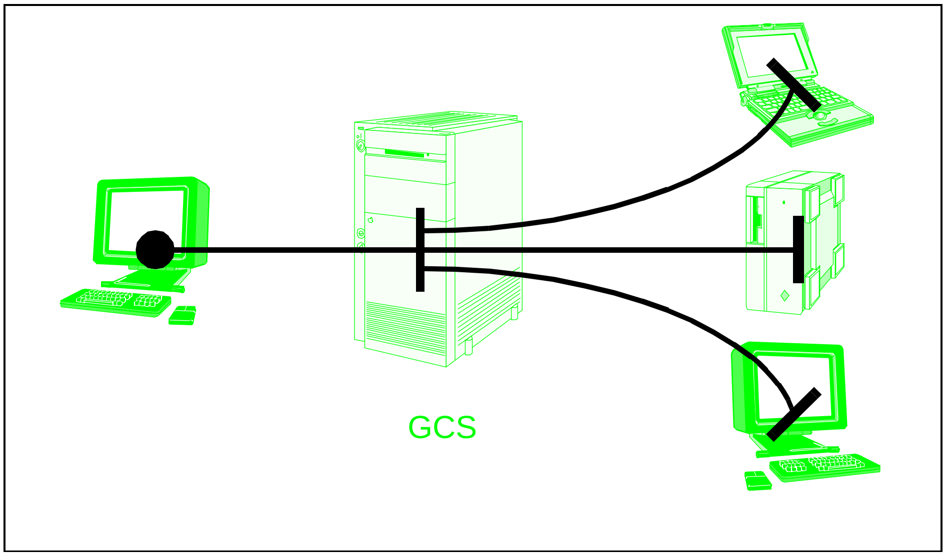
Group Communication Server:

A Scenario-Based Design Exercise



Daniel Amyot
damyot@csi.uottawa.ca

SITE, University of Ottawa
June 1998

List of Figures

List of Figures

List of Figures

List of Tables

List of Tables

Glossary

Acronym	Definition	
ACE	Adaptive Communication Environment	1
ADL	Architecture Description Language	89
ADT	Abstract Data Type	48
BBE	Basic Behaviour Expression	75
BE	Behaviour Expression	75
CID	Channel identifier (ADT)	17
CITO	Communications and Information Technology Ontario	1
CMM	Capability Maturity Model	3
CORBA	Common Object Request Broker Architecture	3
CTMF	Conformance Testing Methodology and Framework	56
FC	Functional Coverage	90
FDT	Formal Description Techniques	2
FIFO	First In First Out	26
FM-CMM	Formal Methods CMM	3
FMCT	Formal Methods in Conformance Testing	56
FSM	Finite State Machines	56
FTP	File Transfert Protocol	22
GCS	Group Communication Server	1
GID	Group identifier (ADT)	16
HTTP	HyperText Transfert Protocol	21
IP	Internet Protocol	17
IRC	Internet Relay Chat	14
ITU	International Telecommunication Union	5
LIFO	Last In First Out	52
LOTOS	Language Of Temporal Ordering Specification	2
LSET	External Spawning - External Threads (distribution model)	23
LSLT	Local Spawning - Local Threads	23
LTS	Labeled Transition System	57
MGCS	Manager of GCS	15
MID	Member identifier (ADT)	16
MSC	Message Sequence Chart	5
RAD	Rapid Application Development	4
SSET	Shared Spawning - External Threads	23
SUT	Specification Under Test	56
TMDL	Timethread Map Description Language	44
TRIO	Telecommunication Research Institute of Ontario	1
UCM	Use Case Maps	2
XELUDO	Environnement LOTOS de l’Université d’Ottawa (tool)	48

Glossary

Group Communication Server: A Scenario-Based Design Exercise

Daniel Amyot, University of Ottawa
damyot@csi.uottawa.ca

TRIOsoft project, June 1998

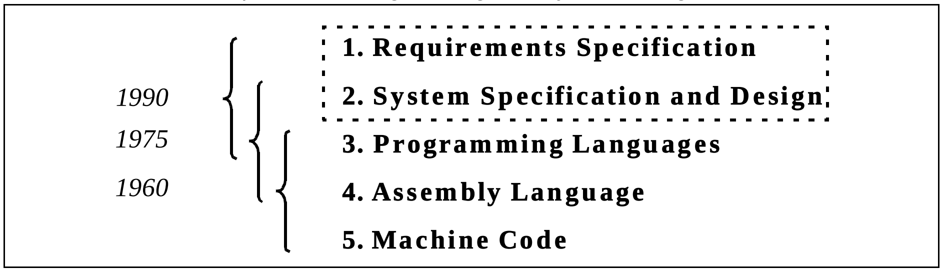
Chapter 1 Introduction

1.1 Motivation

This work started as part of a project where a multidisciplinary team, composed of researchers from three universities of Ontario, established a collaborative case study under the auspices of the *Telecommunication Research Institute of Ontario* (TRIO), which has recently become *Communications and Information Technology Ontario* (CITO). The main goal of this project was to demonstrate the usefulness of the theories, methods, and tools, developed by each group of the team, via a unique task: the design and implementation of a *Group Communication Server* (**GCS**). Our intent was to show how our research could fit in a telecommunication software development process that includes the specification, design, verification, validation, component-based implementation (using a new and promising component-based framework: the *Adaptive Communication Environment* — **ACE** [Schmidt, 1994]), testing, and performance measures of a non-trivial example, namely the GCS.

Along the way, another project involving agent systems superseded the first one [Buhr, 1997a]. Fortunately, the approach on which we have been working is still relevant to this new research direction. We focus on the preliminary steps of *scenario-based* requirements engineering and on the generation and validation of a first prototype design, independently of the target implementation technology, whether it is procedural or based on components or agents. We believe there is a natural evolution of software design methodologies towards requirements engineering and high-level design, where the errors are the most costly for software producers. This trend was illustrated (see Figure 1) by Piotr Dembinski at FORTE 95 [Courtiat *et al.*, 1996]. Our approach aims the rapid generation and the validation of design prototypes from informal operational requirements (dotted rectangle in the figure). We believe this would represent a major step in understanding software systems and enhancing their quality, while reducing their cost and time to market.

FIGURE 1. Evolution Towards Requirements Engineering and System Design



Several techniques are to be used to achieve this goal. We will use **LOTOS** [ISO, 1988] to describe the specification obtained from high-level scenarios (*Use Case Maps* — **UCM**) [Buhr and Casselman, 1995, and Buhr, 1997b]. The design will also be documented with tables describing the activities, and with Parnas' logic tables [Parnas *et al.*, 1994] when appropriate. We will use the tools developed or used within our research groups for verification, validation, scenario-based validation testing, and coverage measurement.

1.2 General Research Questions

Here is a collection of questions related to the use of scenarios and components, included in order to give an idea of the research context within which we produced this document. This report does not intend to answer them all, but some elements of answer are to be given.

Development process and reuse

The process of going from informal functional or operational requirements to high-level formal specification is a research area where many people have been working in the last two decades. However, everything has not been said yet, and many challenges still remain. *Formal Description Techniques* (**FDT**), such as LOTOS, were created in order to formally express functional requirements, most of the time operationally. In particular, they are well suited for the precise definition of telecommunication systems.

It is a well-known fact that one cannot specify a whole system in one single attempt with the level of details that such languages require. System design is an iterative and incremental process, and these languages are not always intrinsically efficient or user-friendly with respect to such an approach. For instance, if new functionalities were to be added to a system specified in LOTOS, much experience with the language would be needed to achieve this goal gracefully.

Code reuse is a reality for many programming languages. However, people tend to "reinvent the wheel" each time they write a specification. Many communication tasks and structural organizations (let us call them *specification patterns*) are rewritten over and over, without any real reuse. Although the specification languages might not all have efficient and flexible modularity features (if any), libraries of components (including both behaviour and data) can usually be defined and reused to some extent. So we should not blame it all on the languages. Reuse, even at the specification level, has to be part of every software development process.

As Ed Brinksma mentioned in his invited talk at FORTE'96: if we were to fit the current FDTs into a CMM (*Capability Maturity Model* [Paulk *et al.*, 1993][Herbsleb *et al.*, 1997]) adapted for formal-methods (FM-CMM), we would still be at the first level (referred to as *initial* or *anarchy!*). In fact, a concrete formal specifications maturity model had been recently suggested in [Fraser and Vaishnavi, 1998]. The sole use of an FDT (such as LOTOS) belongs indeed to the first level of maturity in that particular model. We believe that the approach presented in this report can help reaching the third level (out of five) on that scale, and this is very exciting to us.

From scenario-based requirements to high-level specifications

In this project, we concentrate on specification synthesis from *functional* scenarios. Non-functional requirements, such as performance, timing constraints and robustness, are not explicitly considered, although some are implicit in the chosen structure or architecture in qualitative terms. Many issues related to the use of scenarios concern:

- the completeness of scenarios (requirements coverage);
- the consistency of scenarios (they should not contradict each other);
- the granularity of scenarios (at the same abstraction level);
- the satisfaction of informal requirements;
- the traceability between the models;
- the scalability of the synthesis approach in an iterative process.

How do we then create a functional specification from a set of scenarios? Many synthesis methods, especially for protocols [Probert and Saleh, 1991], already exist. However, they are generally concerned with scenarios based on components, not on *end-to-end causal relationships* between components.

From specifications to component-based implementations

We would like to use our experience in scenario-based requirement analysis to construct the functional specification and test cases. We also want to use component-based distributed platform for the implementation. The main challenge here is the definition of a development process that integrates scenarios (UCMs, MSCs, prose descriptions, etc.) and components on top of a distributed platforms (such as ACE or CORBA, in OO languages such as C++ or Java) in a consistent and seamless way.

In our specific case, some relevant questions are: would the definition of an ACE-oriented style of LOTOS specifications (if at all possible) or more generic *specification patterns* help us coping with this problem? Would a component-based specification narrow the gap between the functional specification and the (component-based) implementation? Would this be resulting in a higher degree of confidence? Would such systems require less exhaustive testing? Should we try to extract formal LOTOS-oriented descriptions of ACE components and services?

Specification environment

One of the major reuse problems is the lack of good specification environments that integrate these two types of feature: incremental specification and components. A syntax-based editor and a compiler are necessary tools, but they are definitely not sufficient to help specifying a system. What are the tools and techniques needed to achieve reuse at specification level?

Programmers now use very popular and productive development tools called RAD (*Rapid Application Development*) such as Visual Basic, Visual Age for Java, or Delphi. Should we consider having such tools for specification languages? Would their complexity become less apparent, more hidden? Could they integrate, as components, well-known patterns for specification, if they exist at all?

All these questions are relevant to the definition of a design process that integrates scenarios and components. This document, however, does not attempt to answer then all. We mainly focus on a design process that allows the generation and validation of high-level and component-based specifications from scenario-based requirements, without any attempt at this point to get to component-based implementations.

1.3 Goals

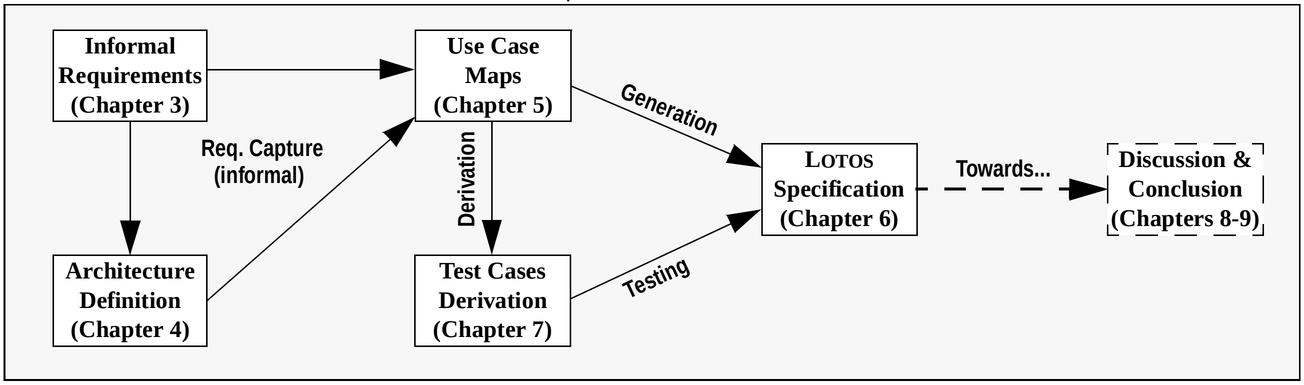
The current work aims to provide experience and elements of answers to the complex research questions enumerated in the previous section. Here are several specific goals for this report:

- precise definition, through scenarios, of a non-trivial application used as a common case study;
- high-level design and documentation of the GCS;
- validation and verification of the GCS specification;
- enumeration of major issues in order to continue the work towards component-oriented specifications and implementations.

1.4 Structure of this document

The construction and validation of a functional specification of a GCS is the main output of this report. Figure 2 illustrates the document structure, which is closely related a pragmatic and rigorous design process undertaken. We first give, in Chapter 2, an overview of the approach taken to produce and validate our specification. We then present an informal description of such a server in Chapter 3. We discuss the structure in Chapter 4, followed by a presentation of the operational scenarios (UCMs and tables). Chapter 6 discusses the LOTOS specification generated from the scenarios. We cover how we validated the GCS specification in Chapter 7, the most complex chapter of this report. We finally discuss several issues and give our conclusions in the last two chapters.

FIGURE 2. Structure of the Report



Chapter 2 Background and Overview of the Approach

2.1 Scenarios

Over the last few years, we observed a strong interest, from both academia and industry, in the use of scenarios for system design. The venue of *use cases* [Jacobson *et al*, 1993] in the OO world confirmed this trend. Many methodologies are now available. However, many different meanings were associated to the word "scenario". They are related to traces (of internal/external events), message exchanges between components, 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 them: grammars, automata, messages exchange diagrams similar to MSCs (*Message Sequence Chart*) in Z.120 [ITU, 1996]. The approaches available thus differ on many aspects, depending on the definition and the notation used.

2.1.1 Scenarios for Requirements Engineering

The use of scenarios for requirement engineering bears advantages and drawbacks. A non-exhaustive list of the most relevant ones follows:

Good Points

- Scenarios are intuitive and close to the requirements. Designers and clients can understand them. They are particularly well-suited for operational descriptions of reactive systems.
- They can be introduced, in a seamless way, in iterative and incremental design processes.
- They can abstract from the underlying system structure.
- They are most useful for documentation and communication.
- They guide the requirements-based tests generation.

Not So Good Points

- Design approaches based on scenarios are recent and seldom possess a high level of maturity. Scalability and maintainability represent notably important issues.
- Completeness and consistency of a set of scenarios are hard to check, especially when the latter are not described at a uniform abstraction level.
- The synthesis of automata for components, from a collection of scenarios, remains a complex problem.
- The use of scenarios leads to the usual problems related to traceability with other models used in the design process.

2.1.2 Scenario Definitions

Many definitions of the term "scenario" exist, and it would be impossible to enumerate them all. We selected six important critters that could however categorize many of these definitions:

- **Hiding**: Scenarios could describe partial descriptions of system behaviour w.r.t their *environment* only, or it could include *internal information* as well.
- **Multiplicity**: We can either have one *single* trace only or possibly *multiple* traces per scenario.
- **Ordering**: Scenarios represent a collection of events ordered according to *time* or to *causality.*
- **Abstraction**: An *abstract* scenario is generic, with formal parameters, while a *concrete* scenario concentrates on one specific instance, with concrete values.
- **Component-orientation**: Scenarios can be described in terms of communication events between system components, or else independently from components, in a pure functional style (end-to-end).
- **Acceptance**: Scenarios usually describe ways to use the system to accomplish some function the user desires or *accepts*. Some notations also present uses that have to be forbidden or *rejected*.

In the next section, we present our scenario notation and show how it stands with respect to these critters.

2.2 Use Case Maps

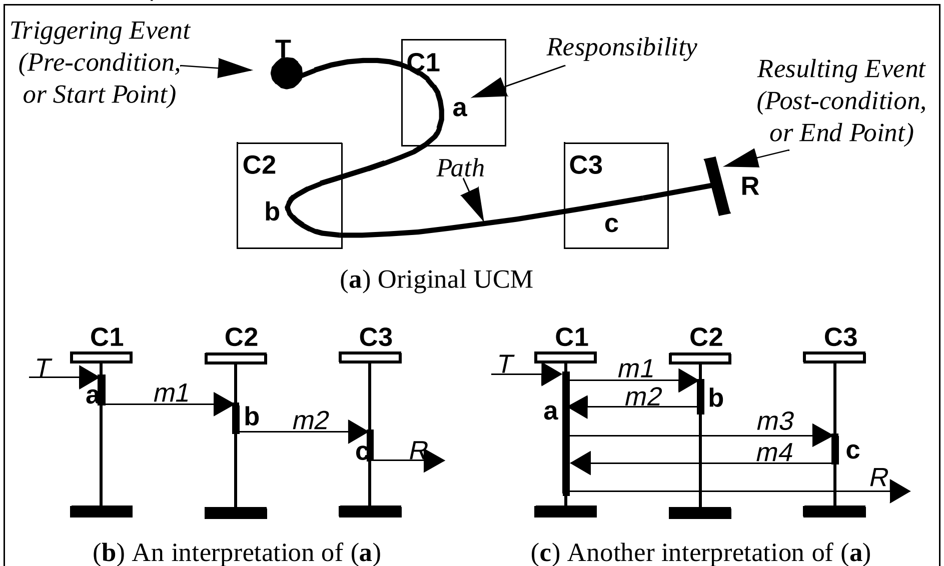
Use Case Maps (UCMs—previously called *Timethreads*) are a visual notation we utilize for capturing the requirements of reactive systems. They describe scenarios in terms of *causal relationships* between *responsibilities*. They can have internal activities as well as external ones. Usually, UCMs are abstract (generic), and could include multiple traces. With UCM, scenarios are expressed above the level of messages exchanged between components, hence they are not necessarily bound to a specific structure. Finally, UCMs handle both acceptance and rejection scenarios.

The following diagrams will illustrate some of the most important concepts of UCMs. For a detailed description of the notation, refer to [Buhr and Casselman, 1995] and [Buhr, 1997a].

Figure 3(a) shows a very simple UCM that contains only one *route*, linking a cause to an effect. A scenario starts with a triggering event of a pre-condition (filled circle) **T** and ends with one or more resulting events or post-conditions (bar) **R**. Intermediate responsibilities (**a**, **b**, **c**) have been activated along the way. Think of responsibilities as being tasks or functions to be performed, or events to occur. In this picture, the activities are allocated to abstract components ($\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$), which could be seen as objects, processes, agents, databases, or any kind of concrete components. We call such superposition a *bound map* (and respectively an *unbound map* when no component is shown). The notation also allows for alternative and concurrent paths, and for interactions between paths. It will be further developed along with the example in the next chapter.

A causal relationship can be refined in many ways in terms of exchanges of messages, depending on the components structure, on the availability of communication channels, and on the chosen protocols. For instance, two MSCs (Figure 3(b) and Figure 3(c)) could be considered as valid implementations of the UCM. The left alternative represents the most straightforward interpretation while the right one indicates that activity **a**, located in $\mathbf{C_1}$, has total control over activities **b** and **c**.

FIGURE 3. Use Case Maps Notation



(**a**) Original UCM

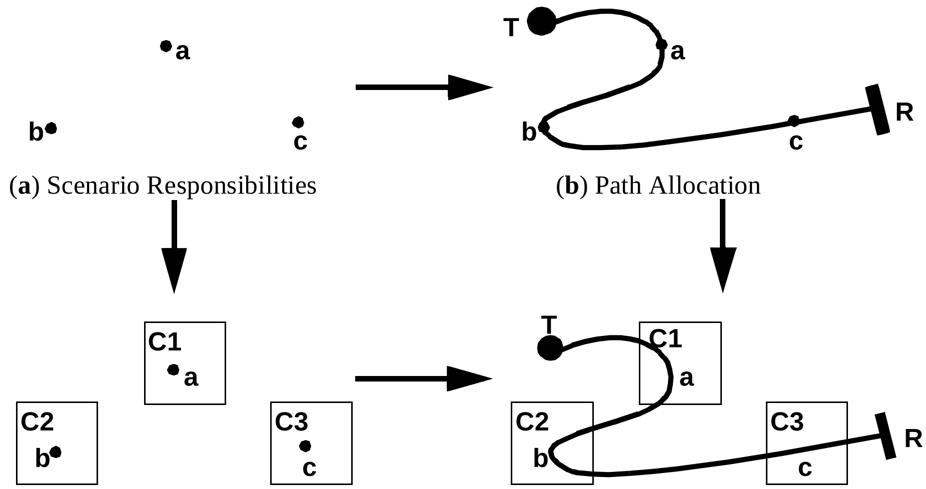
(**b**) An interpretation of (**a**)

(**c**) Another interpretation of (**a**)

UCMs describe scenarios in terms of causality, above the level of messages.

As shown in Figure 4, the construction of a bound UCM can be done in many ways. Usually, one starts with the activities that are to be performed by the system (a). They can then be allocated to scenarios (b) or to components (c). Eventually, the two views are merged to form a bound UCM (d).

FIGURE 4. Use Case Maps Construction



(**a**) Scenario Responsibilities

(**b**) Path Allocation

(**c**) Component Allocation

(**d**) Bound Map

UCMs are constructed from responsibilities, allocated to causality paths and/or components.

Through the binding of responsibilities to components, different target structures can be evaluated for the scenarios. In Figure 5, the same scenario is bound to two alternative structures (a) and (c). In this diagram, the components structures are similar, but their communication links are not the same, leading to different refinements. In (d), the causal relationship between responsibilities **b** and **c** cannot be expressed directly as a message between C_2 and C_4 because there is not any channel that links these two components. However, C_1 could be used to forward a message.

FIGURE 5. Evaluation of Structures with UCMs

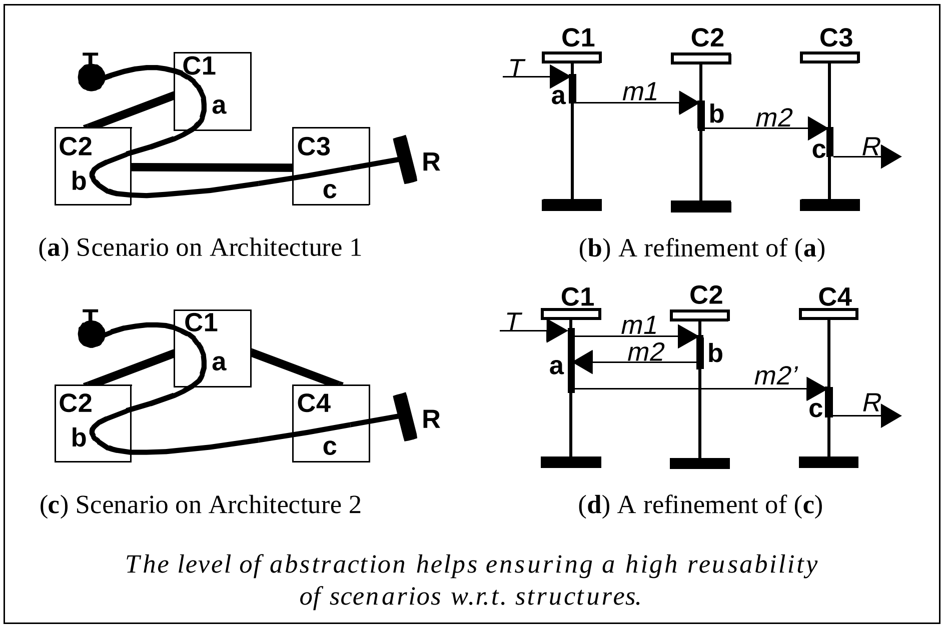
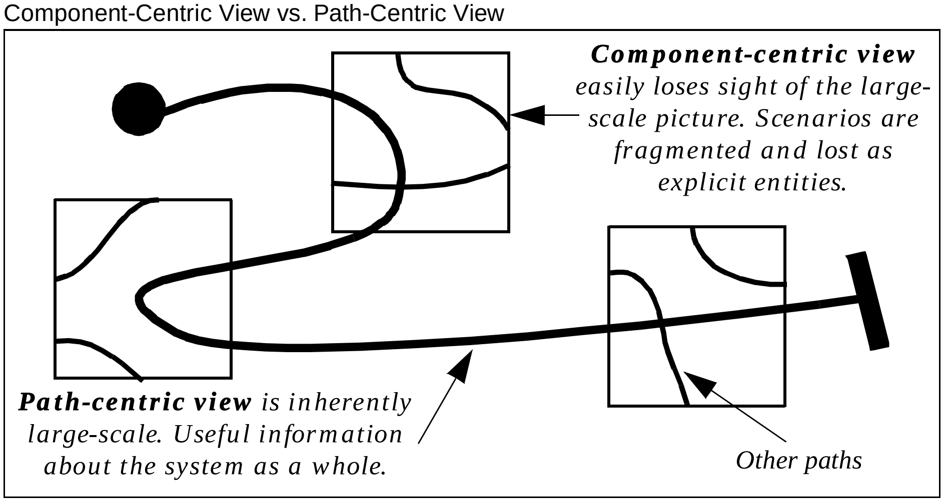


Figure 6 emphasizes the differences between two important views of a system. A *path-centric view* is inherently large-scale and provides useful information about scenarios and the system as a whole. *Component-centric view* easily loses sight of the large-scale picture. Scenarios are then fragmented and lost as explicit entities. The former view eases the thinking process while the latter view helps in the synthesis of component behaviour.

FIGURE 6. Component-Centric View vs. Path-Centric View



2.3 LOTOS

Formal methods, in particular process algebras, proved their usefulness in capturing descriptions of complex, concurrent, and communicating systems. LOTOS (Language Of Temporal Ordering Specification) is an algebraic specification language and a standardized Formal Description Technique [ISO, 1988]. Using LOTOS, the specifier describes a system by defining the temporal relations along the interactions that constitutes the system's externally observable behaviour.

LOTOS is powerful at describing and prototyping communicating systems at many levels of abstraction through the uses of *processes*, *hiding* and multiway, non-deterministic *synchronization*. LOTOS is suitable for the integration of behaviour and structure in a unique executable model. These models allow the use of many validation and verification techniques such as step-by-step execution (simulation), random walks, testing, expansion, model checking, and goal-oriented execution. Many tools can be utilized for the automation of these techniques, and several development processes are available [Bolognesi *et al.*, 1995].

The main LOTOS constructors are recalled in Table 1, where a is an action, B_i are behaviour expressions, g_i are gates, and P is a predicate.

TABLE 1. Main LOTOS Operators

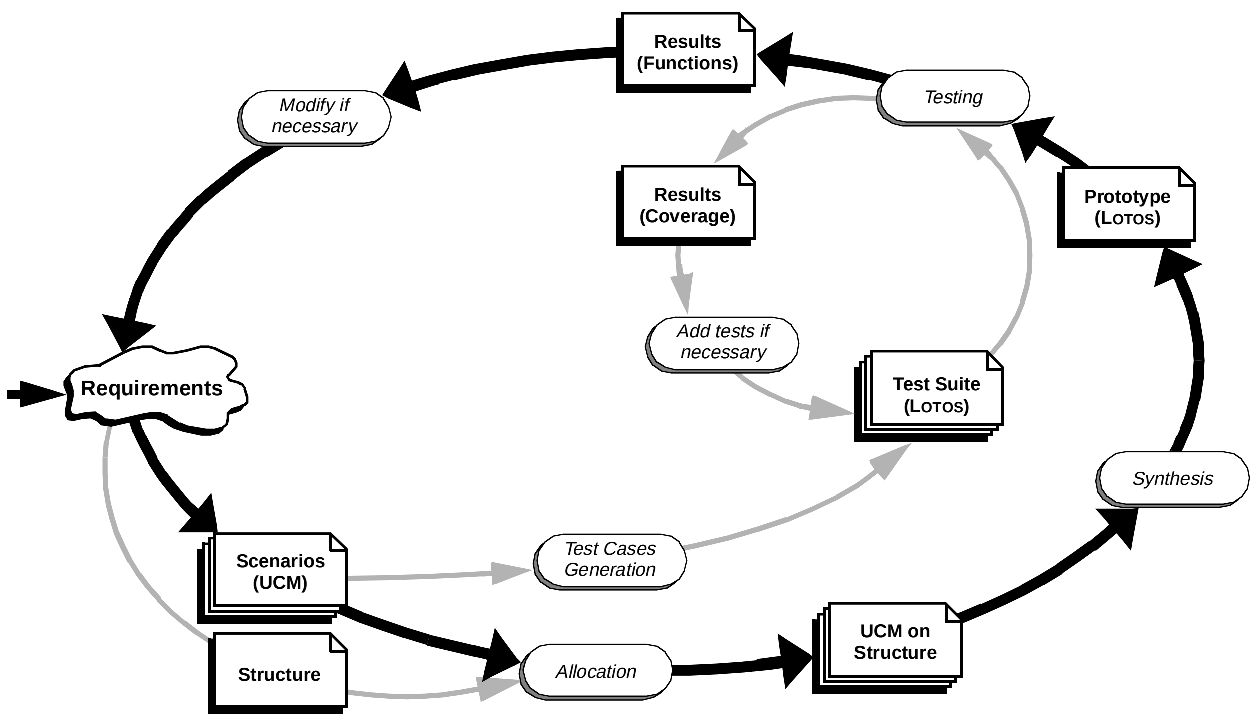
	Name	*Behaviour Expression*	*Comment*
Basic Behaviour Expressions	Inaction	**stop**	Cannot engage in any interaction (deadlock).
	Successful Termination	**exit**	Indicates that a process has successfully performed all its actions.
	Process Instantiation	ProcName [g_1, ..., g_n]	Creates an instance of a process.
Basic Operators	Action Prefix	a**;** B	Used to prefix a behaviour expression B with an action a. There exists a special action, called **i**, that a process can execute independently.
	Choice	B_1 **[]** B_2	Allows the user to define different alternatives for a given process.
Enabling and Disabling	Enabling	B_1 >> B_2	Used to sequence two behaviour expressions. B_1 has to **exit** for B_2 to be executed.
	Disabling	B_1 **[>** B_2	Used to express situations where B_1 can be interrupted by B_2 during normal functioning.
Composition	Parallel Composition	B_1 **|[**g1, ..., gn**]|** B_2	Composition in which 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	Composition in which B_1 and B_2 behave independently (the synchronization set is empty).
	Full Synchronization	B_1 || B_2	Composition in which B_1 and B_2 are synchronized on all their gates.
Other Operators	Hiding	**hide** g_1, ..., g_n **in** B	Used to hide actions (g_1, ..., g_n) which are internal to a system. These actions cannot 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 variable identifier (x) of sort s in B.

2.4 The Approach

We believe that the usage of UCMs in a scenario-oriented approach represents a judicious choice for the description of reactive and communicating systems. They fit well in the design approach that we propose in Figure 7, where we intend to bridge the gap between informal requirements and the first system design. This approach improves the maturity of a design process based on formal specifications.

Requirements are usually dynamic; they change and are adapted over time. This is why we promote an iterative and incremental process (in spiral) that allows rapid prototyping and test cases generation directly from scenarios. Figure 7 introduces an approach where the main cycle is concerned with the description of the scenarios and the structure (can be done independently). They are then merged in order to (manually) synthesize a LOTOS specification, our prototype. Concurrently, test cases can be generated from these scenarios and then be used to test the specification. The results we obtain from those tests allow us to see whether or not additional test cases are necessary in order to achieve the desired specification coverage. We can then observe that the prototype corresponds to the requirements.

FIGURE 7. Scenario-Based Approach Used in this Report



We observed several advantages to this rigorous approach:

- **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 with more flexibility. The scenarios then become highly reusable entities. They can be used again to test the implementation.

- **Fast prototyping**: once the structure and the scenarios are selected and documented, a prototype can then be generated rapidly.
- **Test cases generation**: scenarios ease the generation of test cases that relate directly to the operational requirements. The test suite can itself be validated using structural coverage criteria on the model.
- **Design documentation**: the documentation is done as we go along the design cycle. Very often, designers document their design only when they have to; we believe this approach encourages designers to methodically produce useful documentation.

This design process is to be illustrated and discussed further in the remaining chapters.

2.5 Chapter Summary

Scenarios useful entities but they can be complex. We provided the reader with several criteria to categorize the multiple scenario definitions, and we gave the main advantages and drawbacks of their use.

Although the semantics of the notation is still informal to some extent, Use Case Maps let designers focus on system functionalities and reusable end-to-end scenarios, without being constrained to early by an underlying structure or by exchanges of messages between components.

LOTOS is useful for the formal description and validation of communicating, distributed, and reactive systems. It integrates behaviour and structure in a way that is suitable for the generation of executable models from a collection of scenarios (UCMs).

The approach we propose aims to rapidly produce prototypes from scenarios and structures. The generation of a validated test suite, used to check the model with respect to the informal requirements, is another goal of this iterative and incremental design process.

Chapter 3 GCS Informal Description

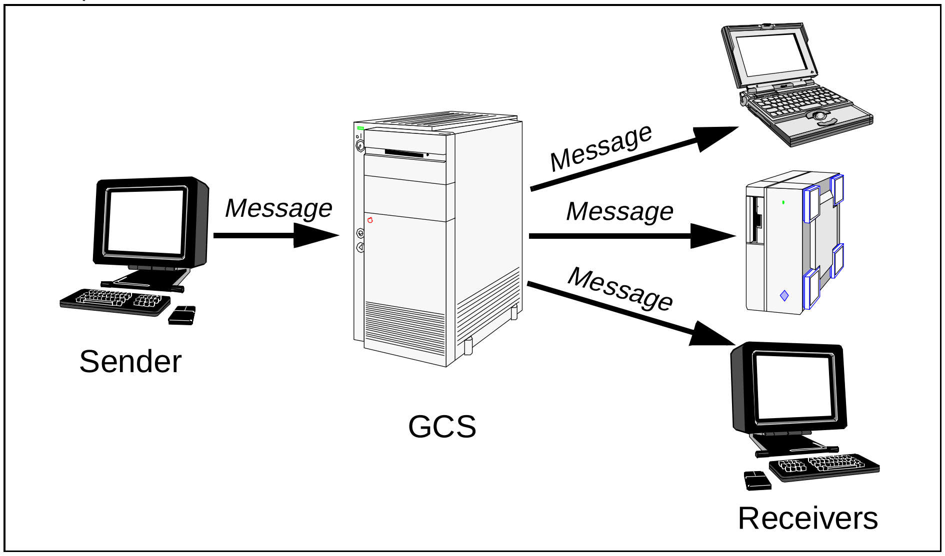
This chapter introduces a high-level and informal description of the requirements. Each functionality is then explained using plain English.

3.1 Usage

A *Group Communication Server* (GCS) allows the multicasting of messages to members of a group (Figure 8). Groups are created and destroyed as the need commands. A GCS offers the core services required for the implementation of systems such as:

- *Mailing list* servers (e.g., Majordomo, Listserv)
- *Internet Relay Chat* (IRC) servers
- *Videoconference* servers
- *Push broadcast* servers for Web publishing (e.g., PointCast, Marimba)
- *Publish and Subscribe* servers for dynamic relationships between applications [Hackathorn, 1997].

FIGURE 8. A Group Communication Server



Users are permitted to join and quit one or many groups. Messages consist in a variety of types (for instance: voice, video, data, objects, or text) and are multicast to the members of the group via different communication channels, selected to suit the requirements of the group.

A group may have an administrator whose tasks might include registration (and optionally moderation) management and group deletion. A group may also have a moderator whose task is to approve or reject messages sent to the group.

3.2 Entities

Here are the principal entities part of the GCS, with their attributes and an overview of their functions or capabilities, as they could be perceived by the users.

3.2.1 Groups

A group is a collection of members. In all cases, only the members of a group receive messages from the corresponding GCS. We can define different types of groups on the basis of four boolean criteria:

Administration Criterion

- *administered*: the sender must be the administrator for the destruction of a group (as in a moderated mailing list). The four criteria are allowed to be changed.
- *non-administered*: the group is destroyed when there are no member left. the criteria cannot be changed.

Subscription Criterion

- *public*: anyone can register to the group (as in a mailing list) or ask for a list of members.
- *private*: the administrator must register all new members (as in a telephone conference). Only members can get the list of group members.

Multicasting Criterion

- *opened*: anyone can send a message to the group (as in a mailing list).
- *closed*: the sender must be member of the group to send (as in IRC).

Moderation Criterion

- *moderated*: the sender must be the moderator for the multicast to take effect (as in a moderated mailing list, or as a conference animator). All other messages are forwarded to the moderator, by the group server, for approval.
- *non-moderated*: the sender does not have be the moderator for the multicast to take effect (as in an ordinary mailing list).

Note that *Abstract Data Types* will be used to handle those characteristics, via guards and predicates.

3.2.2 Manager of GCS

We distinguish a particular entity responsible for the management of groups: the *Manager of Group Communication Servers* (**MGCS**). From now on, regular groups will be referred to as *Group Communication Servers* (**GCS**) and will be responsible for most of the communication functionalities. The MGCS will however manage the creation of new groups and will store a list of existing groups.

3.2.3 Participants

In any case, all participants can request the list of groups available from the MGCS. They can also receive the list of members of a particular group, if they are registered to that group or if the group is private. We denote four types roles for users, which are not necessarily mutually exclusive for a particular group:

Administrator

By default, this is the member who created an administered group. The administrator has the privilege of deleting the group, adding and removing users in a private group, changing the moderator in a moderated group, and changing the administrator or any group criterion.

Moderator

Decides which messages are to be multicast in a moderated group. All messages sent to a moderated group (except those of the moderator) are redirected to the moderator for approval. It can also choose a new moderator for the group.

Ordinary members

Users who receive messages from their group server. They can also multicast messages to their group. Any member can quit a group where it is registered.

Non-members

Users that can potentially register to a public group and multicast messages to an opened group.

3.3 Functionalities

Ten different services are offered by a GCS and two by the MGCS. This section presents the information necessary for the implementation of these services.

The following subsection titles include, between brackets, the entity responsible for the service (MGCS or GCS) and the name of the request message.

In order not to be repetitive, note that all service requests necessitate the *member identifier* of the requestor (type *MID*). This is a unique identifier associated to every user, that could also be used as its address on the communication channel between the user and MGCS. Also, all requests sent to a GCS will be identified by its *group identifier* (*GID*). All acknowledgements from MGCS/GCS will include the member identifier of the requestor and all acknowledgements from a GCS will include its group identifier.

3.3.1 Group Creation (MGCS — CREATEGROUP)

Anyone can create a new group, if the MGCS resources and policies allow it. In this document, we will assume that there are no such constraints. The requestor must provide a *group identifier* that does not already exist (and get **GROUPCREATED**), otherwise an error message is issued (**GROUPEXISTS**). This request needs the following information:

- A new group identifier (type *GID*)

- Multicasting type, associated to the means of communication such as sockets, data, text, video, etc. (type *Chan*). We do not really consider this information in our system for the moment, as it only influences the type of multicasting that is to be used by the underlying structure.
- Requestor's channel identifier (type *CID*), as it will be registered to the group by default.
- Administered / Non-administered (type *Attribute*)
- Administrator identifier (type *MID*)
- Private / Public (type *Attribute*)
- Opened / Closed (type *Attribute*)
- Moderated / Non-moderated (type *Attribute*)
- Moderator identifier (type *MID*)

3.3.2 List Groups (MGCS — GROUPS)

The MGCS keeps a list of groups currently existing. It can provide this list (**GROUPSARE** (*list of groups*)) to any requestor.

3.3.3 Get Attributes (GCS — GETATTRIBUTES)

When the group is not administered or when the requestor is the administrator, the GCS replies with the current list of group attributes (**ATTRIBUTESARE** (*infos*)). This is the most up-to-date information about the information provided during the group creation. If the requestor is not allowed, then it receives a **NOTADMIN** error message.

3.3.4 Group Deletion (GCS — DELETEGROUP)

If the requestor is allowed, then a **GROUPDELETED** acknowledgement is sent by the GCS, and all remaining members are informed as well (by **GROUPWASDELETED**). If the requestor is not the administrator, then the requestor receives a **NOTADMIN** error message. If the group is non-administered, then **NOADMINGROUP** is sent back.

3.3.5 Member Registration (GCS — REGISTER)

If the requestor is allowed, then a **REGISTERED** acknowledgement is sent by the GCS, otherwise (when the group is private and the requestor is not the administrator) the requestor receives a **NOTADMIN** error message. This request needs the following information:

- Channel identifier (type *CID*), representing the specifics of the multicasting type (requestor's host, IP, socket, etc.) for this new member, according to the group requirements.

For administered groups, the administrator is the only user than can register another one. It must then provide the new member identifier (type *MID*)

3.3.6 List Group Members (GCS — MEMBERS)

If the requestor is allowed, then the list of the group members (**MEMBERSARE** (*list of members*)) is sent by the GCS, otherwise (when the group is private and the requestor is not a member of the group) it receives a **MEMBERNOTINGROUP** error message.

3.3.7 Member DeRegistration (GCS — DEREGISTER)

If the requestor is the group, then a **DEREGISTERED** acknowledgement is sent by the GCS, otherwise the requestor receives a **MEMBERNOTINGROUP** error message. For administered groups, the administrator is allowed to deregister another member but the member identifier (type *MID*) must be provided. When there is no member left, the group is deleted.

3.3.8 Multicast (GCS — MULTICAST)

Every such request must be accompanied by a:

- Message (type *Msg*).

If the requestor is in the (non-moderated and opened) group, then a **MESSAGESENT** acknowledgement is sent by the GCS after the multicast. For closed, non-moderated groups, the requestor receives a **MEMBERNOTINGROUP** error message when it is not member.

For moderated groups, the moderator is the only one allowed to multicast (and thus gets a **MESSAGESENT** acknowledgement). All other users' messages (if they are allowed w.r.t. the Opened/Close criteria) are forwarded by the GCS to the moderator for approval. Meanwhile, they receive a **SENTTOMODERATOR** acknowledgement. The GCS also provides the moderator with the member identifier (type *MID*) of the original sender.

Receivers (group members) will get, from the GCS, the message (*Msg*) and the sender's identifier (*MID*) on their channel (*CID*) specified at registration time.

3.3.9 Change Administrator (GCS — CHANGEADMIN)

Every such request must be accompanied by a parameter:

- New Administrator identifier (type *MID*)

If the group is not administered, then the requestor gets a **NOADMINGROUP** error message. The following alternatives assume that the group is indeed administered.

If the requestor is the administrator, and the proposed new administrator is valid (must be a group member), then the change is done and the GCS acknowledges with **ADMINCHANGED**. If the requestor is not the administrator, then it receives a **NOTADMIN** error message. If the new administrator is not member of the group, then the requestor receives a **MEMBERNOTINGROUP** error message.

An administered group could be made non-administered (but not the reverse) by specifying *Nobody* as a new administrator. **ADMINCHANGED** then results (assuming the requestor was the administrator).

3.3.10 Change Open Attribute (GCS — CHANGEOPENATTR)

Every such request must be accompanied by a parameter:

- Opened / Closed (type *Attribute*)

If the group is not administered, then the requestor gets a **NOADMINGROUP** error message. The following alternatives assume that the group is indeed administered.

If the requestor is the administrator, then the attribute is set to the value provided in the message and the GCS acknowledges with **OPENATTRCHANGED**. If the requestor is not the administrator, then it receives a **NOTADMIN** error message.

3.3.11 Change Private Attribute (GCS — CHANGEPRIVATTR)

Every such request must be accompanied by a parameter:

- Private / Public (type *Attribute*)

If the group is not administered, then the requestor gets a **NOADMINGROUP** error message. The following alternatives assume that the group is indeed administered.

If the requestor is the administrator, then the attribute is set to the value provided in the message and the GCS acknowledges with **PRIVATTRCHANGED**. If the requestor is not the administrator, then it receives a **NOTADMIN** error message.

3.3.12 Change Moderator (GCS — CHANGEMODER)

Every such request must be accompanied by a:

- New Moderator identifier (type *MID*)
- Moderated / Non-moderated (type *Attribute*)

If the group is not moderated and the requestor is not the administrator, then the requestor gets a **NOMODERGROUP** error message. The following alternatives assume that the group is indeed moderated, or that it becomes moderated.

If the requestor is the moderator (or the administrator), and the proposed new moderator is valid (must be in the group for closed groups), then the change is done and the GCS acknowledges with **MODERCHANGED**. If the requestor is not the moderator (nor the administrator), then it receives a **NOTMODER** error message. If the new moderator is not member of the group (for closed groups only), then the requestor receives a **MEMBERNOTINGROUP** error message.

The administrator of an administered group can also change the Moderated / Non-moderated flag, along with the new moderator, and then receive **MODERCHANGED** or **MEMBERNOTINGROUP** according to the validity of the new moderator.

Again, a valid request for a modification where the new moderator is Nobody results in a change of status to non-moderated, whatever the value of the attribute provided in the message.

3.4 Chapter Summary

We introduced the *Group Communication Server* and its possible use in real life. We presented the entities (MGCS, GCS, and participants) and described them summarily. Roles were defined for participants, as well as group criteria that will influence the outcome of requests sent to the server.

Twelve functionalities were operationally described in plain English, with the resulting behaviour expected according to the parameters provided and the state of the system.

Chapter 4 GCS Structure

In a design process, a structure of components[1] can be defined before, after, or at the same time as the specification of the functionality. This is often an iterative process. In our view, structures are characterized by their components and how they are combined. In this example, we develop the structure before the scenarios, but it does not have to be this way. In fact, the UCM scenarios documented in Chapter 5 could have been defined independently from the structure.

This chapter explains some of the alternatives on the underlying structure of the group communication server. We then present the selected GCS structure and its constituents.

4.1 Structural Alternatives

4.1.1 Concurrency Models

Several structural patterns for concurrency in a Group Communication Server can be defined. Among them, we considered four models based on those suggested by Schmidt for a related system (a *blob server* [Schmidt, 1996]). Figure 9 illustrates these models at a high level, using a slightly modified subset of the structural notation in [Buhr and Casselman 95]. This notation includes the following elements:

- **Processes** (parallelograms): active component, internally sequential, similar to tasks or processes in an operating system. They usually have control over passive objects.
- **Objects** (rounded corner rectangles): passive component that supports a data or procedural abstraction through an interface. Functions or databases are instances of such objects.
- **Teams** (rectangles): generic component that can include processes, objects and teams.

Components in doted lines indicate that the can be created or deleted dynamically. Stacks of components express replicability, i.e., multiple instances can exist. Lines between components represent communication paths (channels), and we use doted lines when these channels can be created or deleted dynamically. Arrows can be added to a channel to indicate the direction of the communication.

Four high-level structures of components are shown in Figure 9:

a) *Reactive*: Not really a concurrent system. It has one monolithic server process, where the information related to every groups is located (*info* object in Figure 9), that answers all requests for the managed groups.

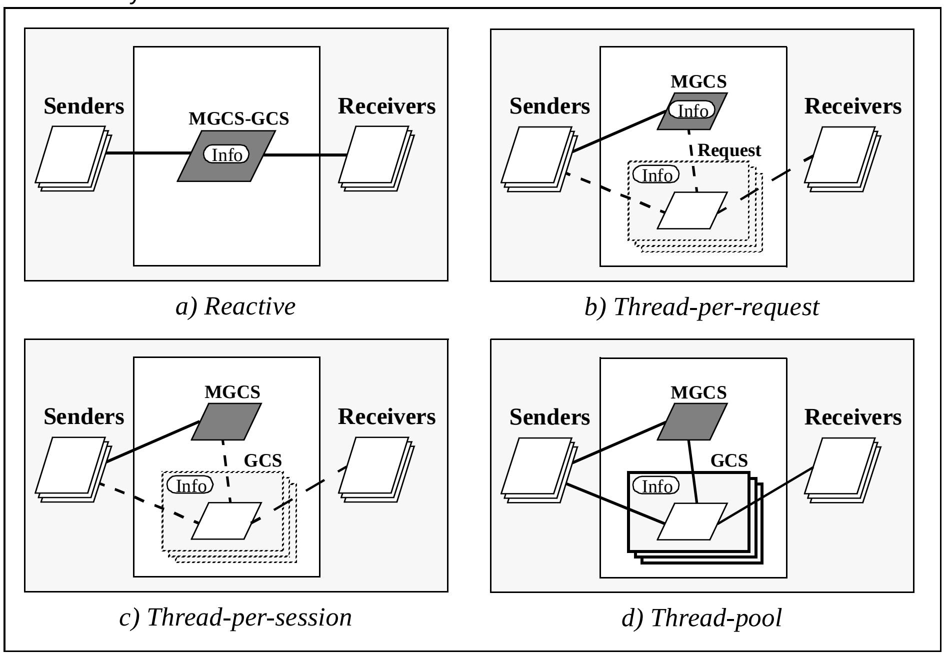
b) *Thread-per-request*: Spawns an independent thread for every incoming request (*à la HTTP*). Highly concurrent model, without any inherent limitation in terms of capacities (other than memory and storage as usual), but very costly due to dynamic creation of many threads. The group information would be passed from the MGCS to the group while spawning the thread.

1. Sometimes, a structure of components is referred to as an *architecture*. In our view however, an architecture possesses more behavioural semantics than a structure or interconnected and embedded components.

c) Thread-per-session: This model is less dynamic and costly than Thread-per-request. It has no inherent limitation, but requires resources while a session (an instance of group with its server) exists. Each group thread has its own local information. Each group could have its own process and execute concurrently with the others, *à la FTP* (although the processes internal execution would be sequential).

d) Thread-pool: Similar to Thread-per-session, but with a fixed number of threads (allocated once, usually at the beginning), hence saving the dynamic allocation performance problems. There are some limitations on the number of simultaneous groups due to the fixed number of instances.

FIGURE 9. Concurrency Models for the Structure



All these alternatives could support the functionalities (Section 3.3) of our group communication server. Option *a* was put aside quickly as it is not concurrent at all. We rejected option *b* because its level of concurrency is too fine-grained and it does not allow grout distribution over a distributed network. Our choice between options *c* and *d* was the Thread-per-session model as it eases the management of groups: if one wants a new group, the MGCS just creates a new one. The MGCS does not have to check and track an empty slot for a new group as in the Thread-pool model.

Choosing this model does not prevent us from selecting another one for internal processes, as we will see in Section 4.2, where we use a Thread-per-request model for the multicast.

4.1.2 Distribution Models

Our structure does not have to be constrained to one processor. Distributing several processes allows us to go from a concurrent system (monoprocessor) to a parallel system (multi-processor). This might result in better performances. The processors could be in one machine or distributed over a network.

In the following alternatives (Figure 10), we split the MGCS process from Figure 9c into a pair of processes (MGCS and Spawn_GCS) in order to decouple the spawning of new groups from their management.

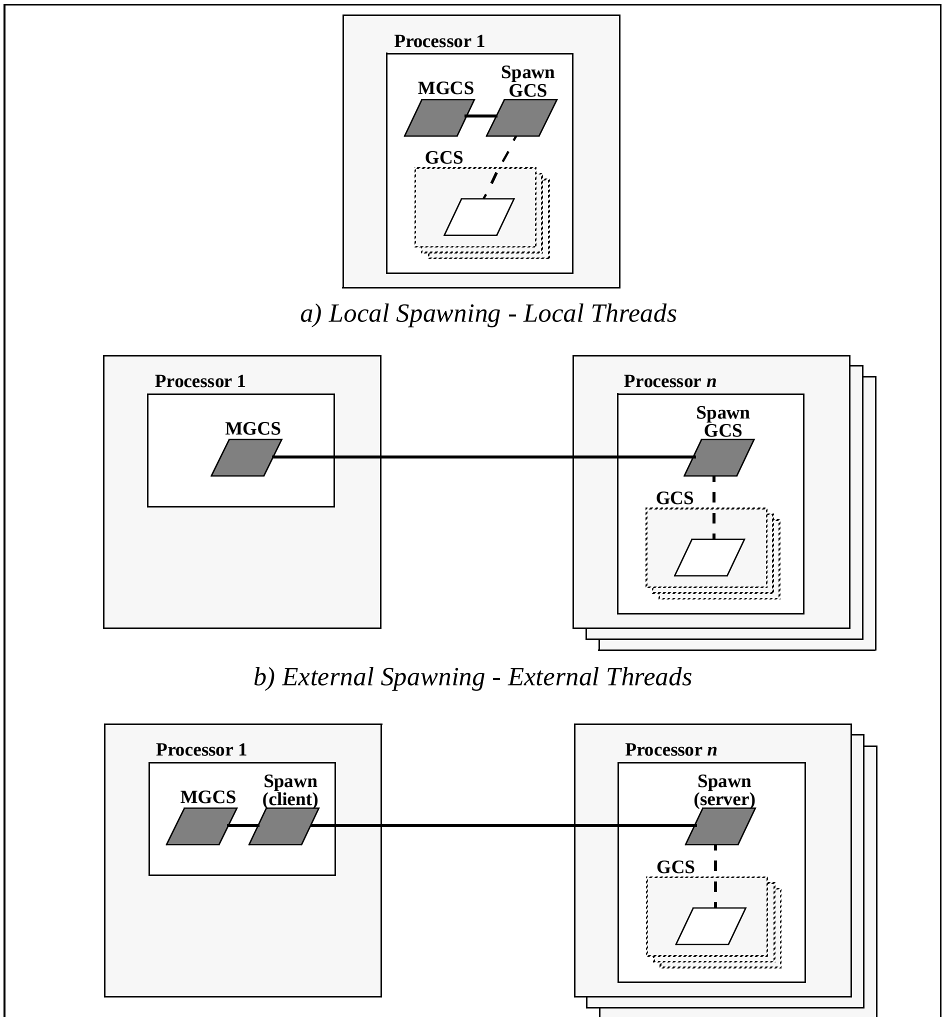
a) Local Spawning - Local Threads (LSLT): Everything is local to one processor. No distribution.

b) External Spawning - External Threads (LSET): The spawn process resides on the same processor as its groups. It is activated by a message from MGCS.

c) Shared Spawning - External Threads (SSET): The spawn process is split into a client side (on the same processor as MGCS) and a server side (on the same processor as its groups). More complex protocols could be used for the spawning, without affecting the MGCS process.

SSET is probably the most generic alternative. However, from a client's viewpoint, an additional level of complexity (the management of the communication channels between a client and a GCS on different processors/machines) occurs in *SSET* while *LSLT* avoids it (all clients communicate with the same processor/machine). We will use *LSLT* in this document because it still captures the essence of *SSET* and it lowers the complexity of the design.

FIGURE 10. Distribution Models for the Structure



a) Local Spawning - Local Threads

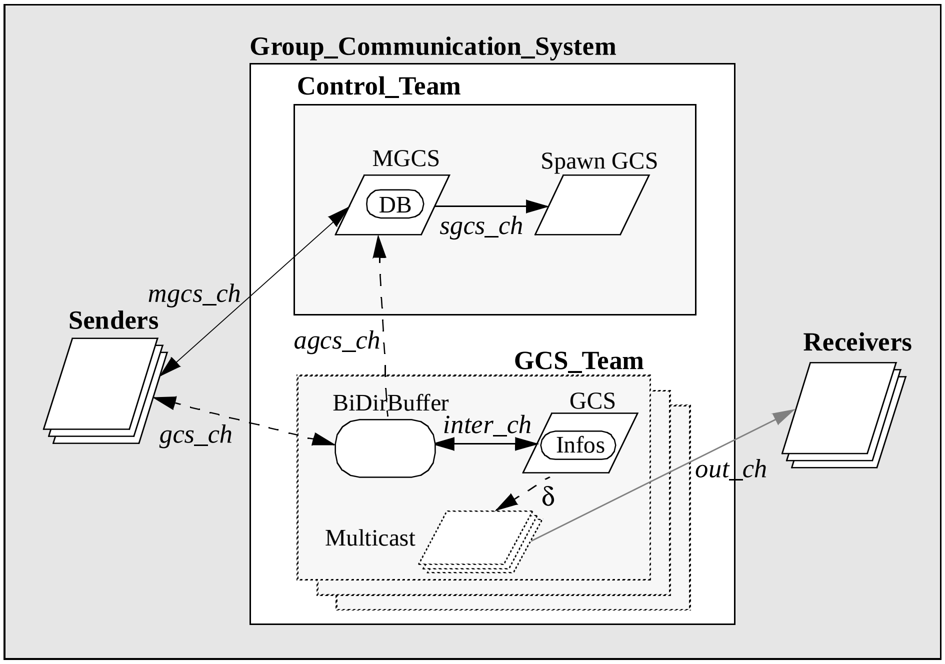
b) External Spawning - External Threads

c) Shared Spawning - External Threads

4.2 GCS Structure

This section presents our structure (Figure 11), based on the *Thread-per-session* model for the management of groups, on the *Thread-per-request* model for the multicasting of messages, and on the *Local Spawning - Local Threads* distribution model. We give further explanations on the structural components.

FIGURE 11. GCS Structure



4.2.1 Teams

The **Senders** and **Receivers** processes are presented as contextual information only (client side) and they are not really discussed in this document. We concentrate on the specification, design and testing of the server side, i.e. the **Group_Communication_System** team, which is composed of two sub-teams:

- one fixed **Control_Team**, for the group management (creation and group list);
- possibly many dynamic instances of a **GCS_Team**, which takes care of most of the functionalities of a GCS.

Note that teams here simply act as containers for objects and processes.

4.2.2 Channels

The processes and objects communicate over several channels:

- *mgcs_ch*: External MGCS channel used for requests and acknowledgements between senders and the MGCS.
- *gcs_ch*: External GCS channel used for requests and acknowledgements between a sender and a specific group.
- *out_ch*: External output channel used by a GCS to multicast messages to its members.
- *sgcs_ch*: Internal administration channel used by the MGCS to communicate with the Spawn process.
- *agcs_ch*: Internal administration channel used by the MGCS to receive group deletion announcements from GCSs (via BiDirBuffer).

- *inter_ch*: Internal channel for the communication between a GCS process and its buffer object.
- δ: Internal channel between the GCS and newly created (and soon to be destroyed) Multicast processes.

out_ch, in collaboration with the channel identifier (*CID*), implements the multicasting type (video, audio, text, etc.) introduced in Section 3.3.1. It requires the use of various means of transmission such as:

- streamed (e.g. audio/video);
- connection-oriented (e.g. IRC);
- connectionless (e.g. mailing list).

This information is used by the Multicast process, but at level of abstraction high enough so that it does not have much impact on the specification and the design.

4.2.3 Control_Team

Upon request on a special request channel (*mgcs_ch*), the MGCS announces (via *sgcs_ch*) the Spawn process that a new GCS_Team needs to be created. The MGCS may also list the existing groups, thus indicating to the senders which groups are currently available. The MGCS is notified by the GCS of its deletion; this allow the GCS list to be kept up to date. The destruction of a group is a decision left to the GCS because it possesses all the information needed to validate such a critical request.

It might be a good idea to create a super-administrator and to give the MGCS the authority of destructing a GCS, especially for administrative purposes: maintenance of the server, destruction of illicitly created GCS, overloading, groups inactive for too long, etc. Nevertheless, to simplify the design, we will neither provide this additional functionality to the MGCS nor centralize the destruction control in the hands of a super-administrator.

4.2.4 GCS_Team

Each GCS has a bi-directional, unbounded buffer (BiDirBuffer) and is independent of the other ones. It decodes a request and reacts accordingly. It announces its termination to the MGCS on *inter_ch*, and this is forwarded towards *agcs_ch* by BiDirBuffer. When a message to be multicast is received, GCS spawns as many concurrent instances of the Multicast process as there are members registered.

Multicast is responsible for the delivery of a message to its assigned group member over *out_ch*. Multicast also uses the GCS resources (*inter_ch*) to advertise the destruction of the group to its members (when there are members left).

BiDirBuffer uses *gcs_ch* to buffer requests from senders and to send back the acknowledgements. It possesses two FIFO message lists: one for requests, one for acknowledgements. Messages are forwarded to (and received from) GCS via *inter_ch*.

4.3 Chapter Summary

We presented an structural notation that permitted us to consider several topologies of components for the underlying structure of our system. We discussed different models based on concurrency and distribution criteria, and we were able to reason, at a high level of abstraction, about their foreseen usage and performances. Although this could have been done while defining the scenarios (or even after), we chose to first select the GCS structure based on the topologies presented. We gave general information about the components and the links, without committing to too many details.

Chapter 5 Use Case Maps for the GCS

The intent of our scenario-driven strategy is to lead us to the first high-level specification of our system. To do so, we first need to define a set of scenarios as complete and consistent as possible.

This section presents Use Case Maps that capture the essence of the GCS main functionalities (refer to Section 3.3 for an informal description of the requirements). Instead of individual and sequential scenarios (traces), we concentrate on aggregated UCMs that regroup closely-related scenarios, often referred to as *scenario clusters*. These clusters represent alternative outputs to the same input, usually one valid scenario and several exceptional or error scenarios. The composition of multiple scenarios into one complex scenario is simplified by the visual nature of UCMs.

We will not try to merge all aggregated scenarios together in order to synthesize a global functional specification of our GCS. Most GCS scenarios are expressed at a uniform level of abstraction and they are sufficiently independent from one another that a global merging is not necessary to the understanding of the system.

In the following Use Case Maps, we will not show the communication links in order to keep the pictures simpler. Communication will be detailed in the specification. Responsibilities are classified in the tables as follow:

- *Request*: triggering event, usually a message from the clients.
- *Ack*: resulting event, an acknowledgment message sent back to the clients.
- *Error*: resulting event, an error message sent back to the clients.
- *Internal*: internal activity hidden within the component, or internal communication.
- *Cond*: pre or post-condition (predicate).
- *External message*: message multicast or sent by the server.

We assume that each request/ack/error contains the identifier of the client. Also, we will not mention the allocation of responsibilities to the component as they are often obvious from the context.

5.1 Group Creation

The first scenario is concerned with the creation of a new group. Figure 12 shows that when a CREATEGOUP request is sent, the server checks whether the proposed identifier is already in the database or not (**a**). It answers with GROUPEXISTS when the group identifier is already in the MGCS group database (**b**). Otherwise (**c**), a new GCS_Team instance is created (**e-f**) according to the parameters provided by the sender, the MGCS database is updated, and a GROUPCREATED acknowledgement is returned. Communication between the Sender and the MGCS is done via *mgcs_ch*, and *sgcs_ch* is used between MGCS and Spawn_GCS. The responsibilities are explained in detail in Table 2.

FIGURE 12. Group Creation UCM

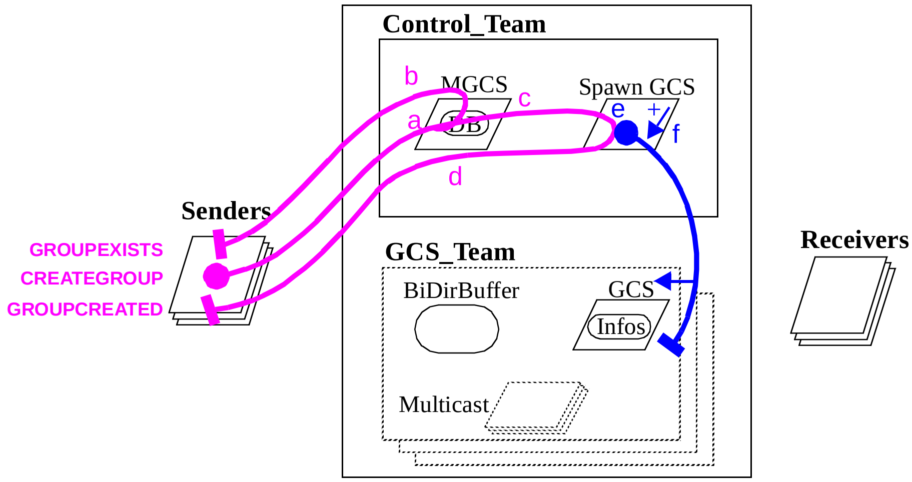


TABLE 2. Responsibilities of “Group Creation UCM”

Responsibilities	Type	Input/Output	Comment
CREATEGROUP	Request	*newgroupid, infos*	Group creation. Anyone can create.
GROUPCREATED	Ack	*newgroupid*	Group created.
GROUPEXISTS	Error	*newgroupid*	Group *newgroupid* already exists.
a	Internal		MGCS reads from the database (*DB*).
b	Cond		*newgroupid* is in *GCSlist (DB).*
c	Cond		*newgroupid* is not in *GCSlist.*
d	Internal		MGCS updates the database (inserts *newgroupid* in *GCSlist*)
e	Internal	*infos*	Relays the CREATEGROUP request.
f	Internal		Instantiates a new GCS_Team with user-provided informations (*infos*)

5.2 List Groups

Any sender can ask for the list of groups supported by a particular system. A GROUPS request is addressed on *mgcs_ch* to the MGCS, and the latter returns the list of groups, GROUPSARE(*GCSList*), contained in its database (**a**). Table 3 details these responsibilities.

FIGURE 13. List Groups UCM

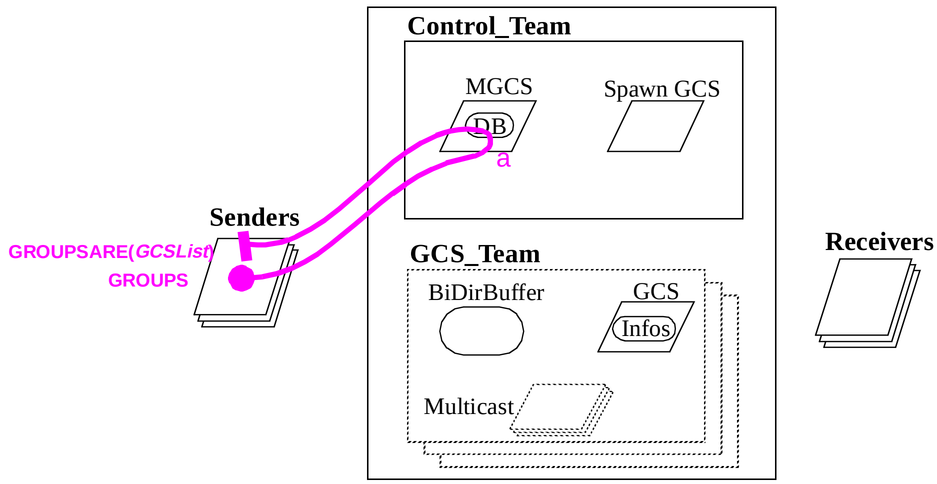


TABLE 3. Responsibilities of "List Groups UCM"

Responsibilities	Type	Input/Output	Comment
GROUPS	Request		List the existing groups within the server.
GROUPSARE	Ack	*GCSList*	List of groups in the server.
a	Internal		MGCS reads from the database (*DB*).

5.3 Get Attributes

A GETATTRIBUTES request is sent to a specific group, and the latter returns the group information, ATTRIBUTESARE(*infos*), kept in its *infos* database (**a**) when the group is non-administered or when the sender is the administrator (**b**). Otherwise, a sender who is not the administrator group (**c**) gets a NOTADMIN error message. These responsibilities are covered in Table 4.

FIGURE 14. Get Attributes UCM

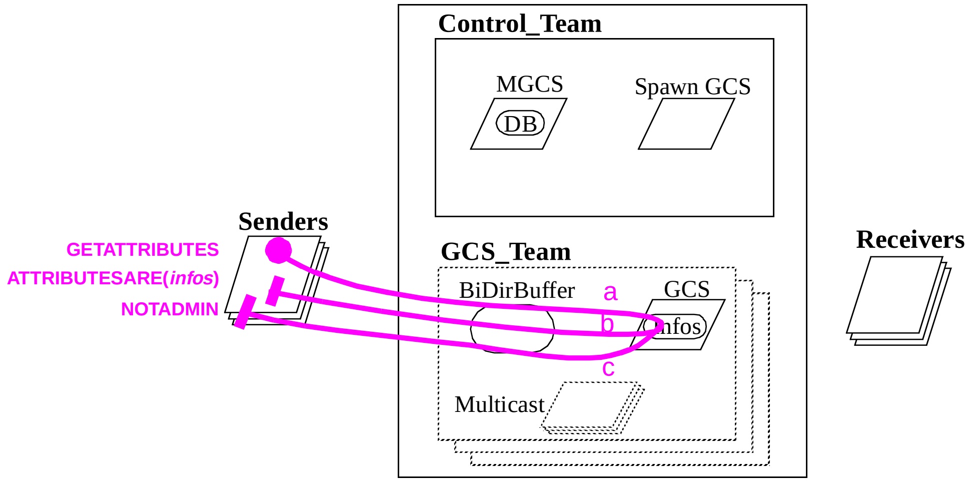


TABLE 4. Responsibilities of "Get Attributes UCM"

Responsibilities	Type	Input/Output	Comment
GETATTRIBUTES	Request		Get the group information.
ATTRIBUTESARE	Ack	*infos*	List of attributes in *infos*.
NOTADMIN	Error		Sender is not allowed to know
a	Internal		GCS checks the group informations (for *IsAdmin* and *Admin*).
b	Cond		Non-administered group, or sender is the administrator.
c	Cond		Sender is not the group administrator.

5.4 Group Deletion

Figure 15 illustrates that a DELETEGROUP request causes the group informations to be read (**a**) in order to determine the deletion policies (who is allowed to delete). The two following conditions check whether the group sender is the administrator of this group (must be administered). If it is the case (**c**), then the multicast is prepared (**d**), the MGCS database updated (**f**), the GCS_Team deleted (**g**), and the GROUPDELETED acknowledgement returned. If not (**b**), then an NOTADMIN error message is emitted.

A GROUPWASDELETED announcement (**h**) is multicast to all group members concurrently (via *gcs_ch* instead of *out_ch*). Then, the Multicast processes are destroyed (**i**) and, when all messages have been sent, synchronization with the main thread resumes its continuation (**e**). See the Table 4 for more details.

The complexity of this functionality is above average mainly because of the necessary validation (we do not want a group deleted by someone who is not allowed to) and the announcement to the remaining members of the group.

FIGURE 15. Group Deletion UCM

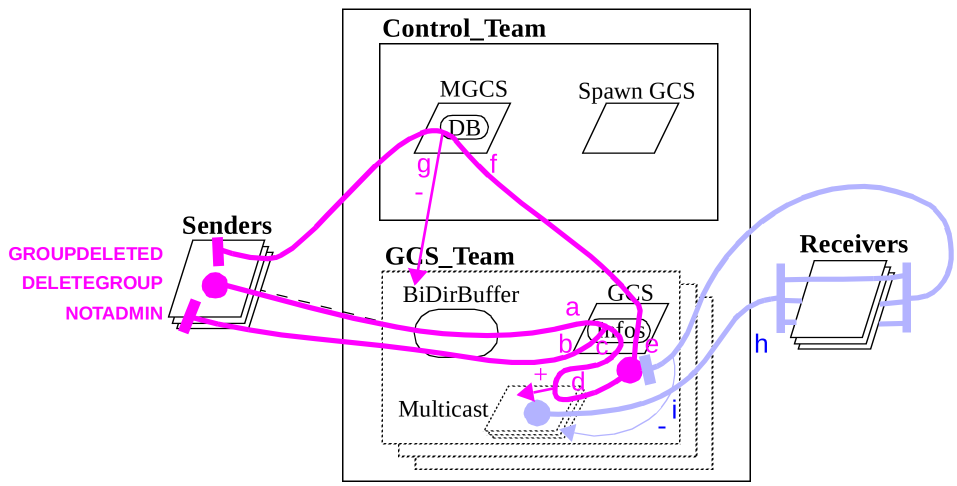


TABLE 5. Responsibilities of “Group Deletion UCM”

Responsibilities	Type	Input/Output	Comment
DELETEGROUP	Request		Group deletion.
GROUPDELETED	Ack		Group deleted.
NOTADMIN	Error		Sender does not have sufficient rights (not administrator).
a	Internal		GCS checks the policy database (*infos*, for *IsAdmin* and *admin*).
b	Cond		Non-administered group or sender is not the administrator.
c	Cond		Administered group and sender is the administrator.
d	Internal		Instantiate Multicast processes (one per group member)
e	Internal		Wait until all messages are sent
f	Internal		MGCS updates the database (removes the group from *GCSlist*)
g	Internal		MGCS destroys the GCS_Team

TABLE 5. Responsibilities of "Group Deletion UCM"

Responsibilities	Type	Input/Output	Comment
h	External message	Message: *GROUPWAS DELETED*	Announce group deletion to all members (using the control channel ***gcs_ch*** instead of ***out_ch***).
i	Internal		Destroys the Multicast processes.

5.5 Member Registration

A REGISTER request (shown in Figure 16) causes the group informations to be compared to the sender (**a**). If the group is public or if the sender is the administrator of the private group (**b**), then the information database is updated with the registered member identifier and its channel identifier (**c**), and the scenario ends with REGISTERED. Otherwise (**d**), it simply terminates with a NOTADMIN error message without any modification. When the new member is already in the member list, its channel identifier is updated with the new identifier provided in the request. Table 5 details these responsibilities.

FIGURE 16. Member Registration UCM

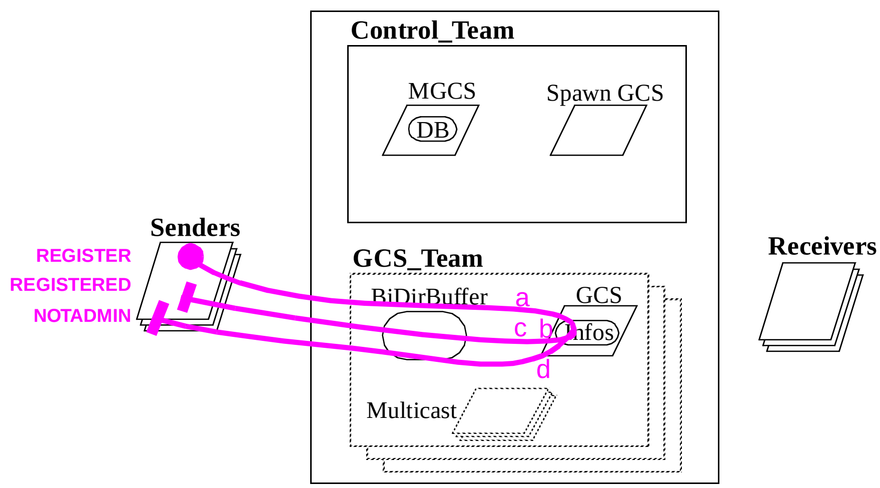


TABLE 6. Responsibilities of “Member Registration UCM”

Responsibilities	Type	Input/Output	Comment
REGISTER	Request	*ChanId*, or *MemberID. ChanID*	Register sender (or another user if admin) in the group. Also used to modify ChanId.
REGISTERED	Ack		Member registered.
NOTADMIN	Error		Sender is not allowed to register.
a	Internal		GCS checks the group informations (for *IsAdmin, Admin* and *IsPrivate*).
b	Cond		Group is public, or sender is the admin.
c	Internal		GCS updates the database (inserts the pair *MemberID.ChanId* in *mbrL*).
d	Cond		Group is private and sender is not the admin.

5.6 List Group Members

A MEMBERS request is addressed to a specific group, and the latter returns the list of its members, MEMBERSARE(*mbrL*), contained in its database (**a**) when the group is public or the sender is a member (**b**). Otherwise, a sender who is not a member of the private group (**c**) gets a MEMBERNOTINGROUP error message. These responsibilities are covered in Table 6.

FIGURE 17. List Group Members UCM

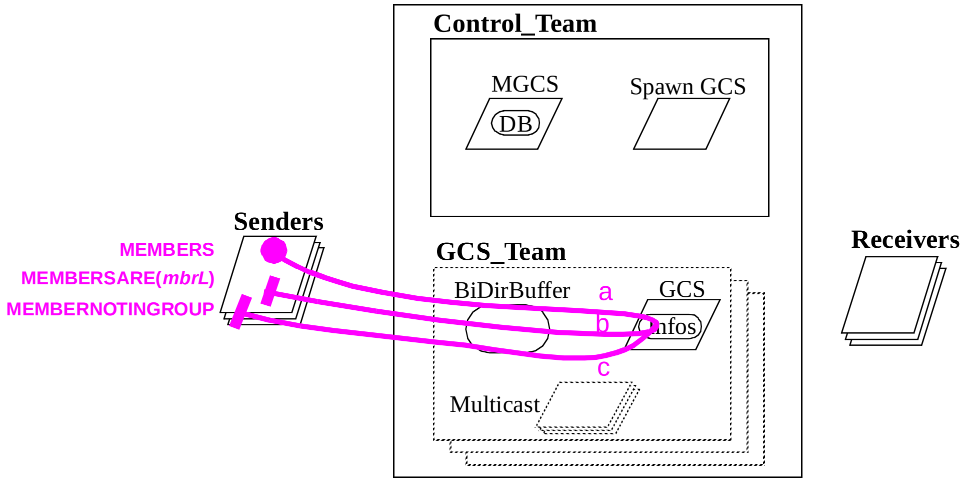


TABLE 7. Responsibilities of “List Group Members UCM”

Responsibilities	Type	Input/Output	Comment
MEMBERS	Request		List the registered group members.
MEMBERSARE	Ack	*mbrL*	List of members of the group.
MEMBERNOTIN-GROUP	Error		Sender is not allowed to know.
a	Internal		GCS checks the group informations (for *IsPrivate* and *mbrL*).
b	Cond		Public group, or sender is a member.
c	Cond		Sender not a member of private group.

5.7 Member DeRegistration

When a DEREGISTRATION request occurs, the GCS checks its information (**a**) for validation. If the sender is a member of the group, or if the administrator deregisters another group member, then it is removed from the list (**b**). The remaining list might be empty (**d**) or not (**c**), but they both cause a DEREGISTER result. In the former case, the GCS also announces the group deletion to the MGCS (**f**), which updates the group database (**g**), deletes the GCS_TEAM (**h**), and results in a GROUPDELETED announcement.

When the sender is not a member of the group, or when the administrator (assuming the group is administered) provides a member identifier that is not in the member list (**e**), then an error occurs (MEMBERNOTINGROUP). Refer to Table 7 for more details.

FIGURE 18. Member DeRegistration UCM

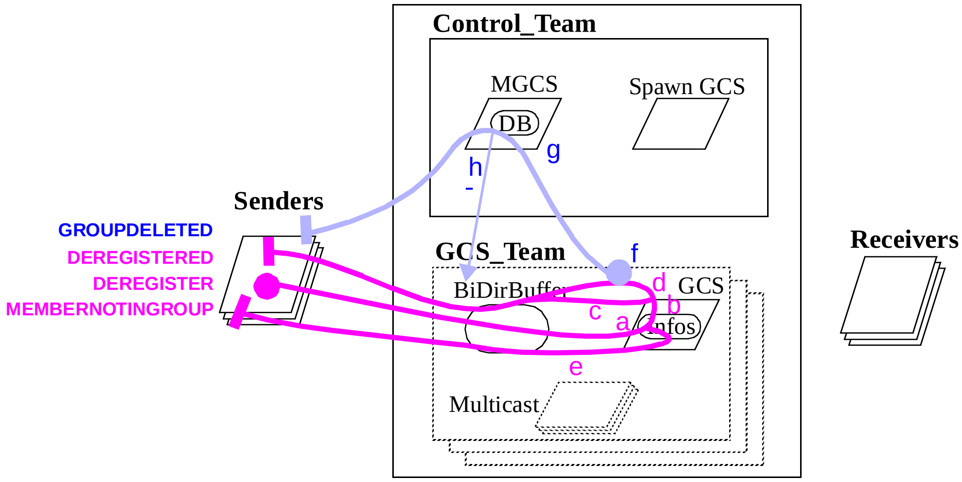


TABLE 8. Responsibilities of "Member DeRegistration UCM"

Responsibilities	Type	Input/Output	Comment
DEREGISTER	Request	Nothing, or *MemberID*	Deregister sender or member identified by administrator.
DEREGISTERED	Ack		Member deregistered.
MEMBERNOTIN-GROUP	Error		Sender is not allowed to deregister, or designed member is not in the group.
GROUPDELETED	Ack		Group deleted.
a	Internal		GCS checks the group informations (for *IsAdmin*, *Admin* and *mbrL*).
b	Cond / Internal		Sender is in the group, or member identified by the admin is in the group: removed from list (*mbrL*).
c	Cond		Member list remains not empty after deregistration.
d	Cond		Member list becomes empty after deregistration.
e	Cond		Sender not in group, and member identified by the admin is not in the group.
f	Internal		Announce the group deletion to the MGCS.
g	Internal		MGCS updates the database (removes the group from *GCSlist*)
h	Internal		MGCS destroys the GCS_Team

5.8 Multicast

As shown in Figure 19 and Table 8, the sender requests the MULTICAST of a message *Msg*. After the examination of the informations (**a**), if the group is private and the sender is not a member (**b**) (remember that the moderator is always a member when the group is private), then a MEMBERNOTINGROUP error is sent back. If the group is moderated and the sender is not the moderator (**c**), then the message is forwarded to the moderator for approval with (**d**), and a SENTTOMODERATOR acknowledgement results.

If the sender is allowed to multicast (**e**), then Multicast processes are created (**f**) to send *Msg* concurrently to all members (**g**). When everything has been sent (**h**), the processes are destroyed (**i**) and the sender receives a MESSAGESENT acknowledgement.

FIGURE 19. Multicast (Message Sending) UCM

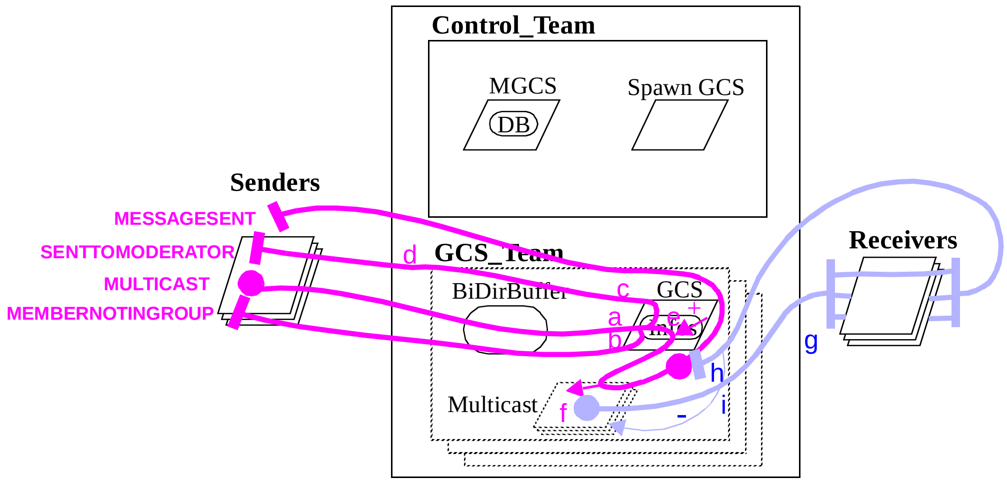


TABLE 9. Responsibilities of "Multicast (Message Sending) UCM"

Responsibilities	Type	Input/Output	Comment
MULTICAST	Request	*Msg*	Group multicast of *Msg*.
MESSAGESENT	Ack		*Msg* was sent to the group members.
SENTTOMODERATOR	Ack		*Msg* was forwarded to the group moderator for approval.
MEMBERNOTINGROUP	Error		Sender does not have sufficient rights to multicast.
a	Internal		GCS checks the group informations (for *IsOpened, IsModerated, Moderator,* and *mbrL*).
b	Cond		Group is closed and sender is not a member.

TABLE 9. Responsibilities of "Multicast (Message Sending) UCM"

Responsibilities	Type	Input/Output	Comment
c	Cond		Group is moderated and sender is not the moderator (and sender is a member if the group is closed).
d	External message	*ToApprove (Sender,Msg)*	*Msg* is forwarded to the moderator for approval.
e	Cond		If group is moderated, then sender is moderator. If group is closed, then sender is a member.
f	Internal		Create one Multicast process per member.
g	External message	*Sender, Msg*	Send *Msg* to the member on ***out_ch***.
h	Internal		Wait until all messages are sent
i	Internal		Destroy the Multicast processes.

5.9 Change Administrator

Upon reception of a CHANGEADMIN request, the GCS checks its local information (**a**). If the sender is the administrator and the proposed *NewAdmin* a member of the group (**b**), then the current *Admin* is modified (**c**) and this results into ADMINCHANGED. In the special case where *NewAdmin* is *Nobody*, the modification is done (although *Nobody* cannot be a group member) and the *IsAdmin* attribute is set to *Nonadministered*.

Such a request sent to a non-administered group (**d**) results in a NOADMINGROUP error. Otherwise, if the sender is not the administrator (**e**), then a NOTADMIN error is sent back. The new administrator must also be a member of the group, or else (**f**) MEMBERNOTINGROUP is returned.

Upon a valid modification, the new administrator could be advised of its new role. We have not included this functionality as it is not part of the informal requirements.

FIGURE 20. Change Administrator UCM

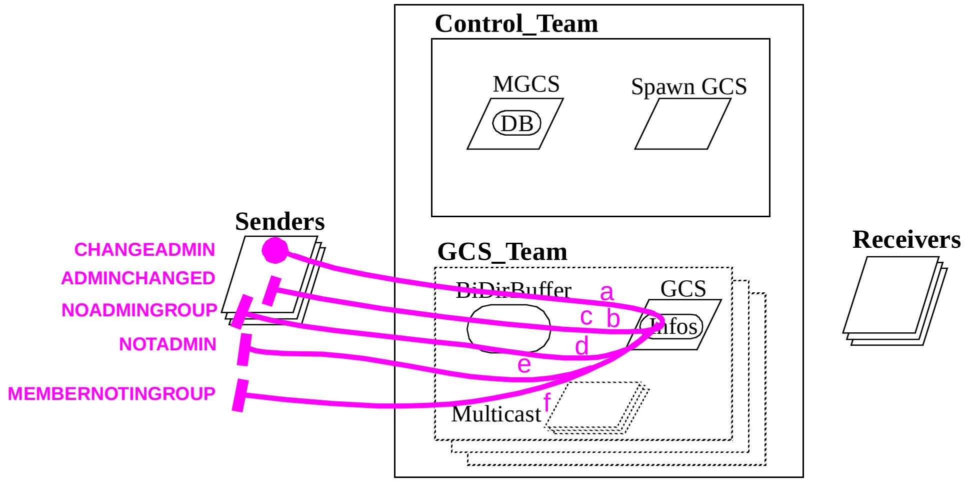


TABLE 10. Responsibilities of “Change Administrator UCM”

Responsibilities	Type	Input/Output	Comment
CHANGEADMIN	Request	*NewAdmin*	Change the group administrator.
ADMINCHANGED	Ack		The administrator and (possibly) the *IsAdmin* attribute have been changed.
NOADMINGROUP	Error		The group is non-administered.
NOTADMIN	Error		Sender is not administrator.
MEMBERNOTIN-GROUP	Error		The proposed administrator is not in the group.
a	Internal		GCS checks the group informations (for *IsAdmin*, *Admin*, and *mbrL*).
b	Cond		Sender is the administrator, and the new administrator is in the group (or is *Nobody*).
c	Internal		Set the new administrator and the *IsAdmin* attribute in *infos*.
d	Cond		The group is non-administered.
e	Cond		Administered group, but sender is not administrator.
f	Cond		Sender is administrator, but *NewAdmin* is not a group member (and is not *Nobody*).

5.10 Change Open Attribute

A CHANGEOPENATTR request, together with a *NewOpenAttr* parameter, causes the GCS checks its local information (**a**). If the sender is the group administrator (**b**), then the current *IsOpened* attribute is modified (**c**) and this results into OPENATTR-CHANGED. When the group is non-administered (**d**), the result becomes a NOAD-MINGROUP error. Otherwise, if the sender is not the administrator (**e**), then a NOTADMIN error is sent back.

FIGURE 21. Change Open Attribute UCM

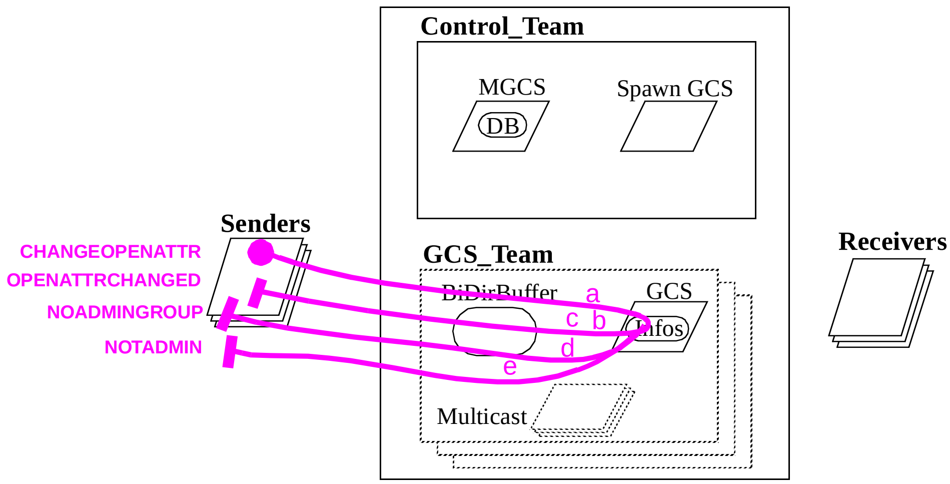


TABLE 11. Responsibilities of "Change Open Attribute UCM"

Responsibilities	Type	Input/Output	Comment
CHANGEOPENATTR	Request	*NewOpenAttr*	Change the group *IsOpened* attribute.
OPENATTR-CHANGED	Ack		The *IsOpened* attribute has been changed.
NOADMINGROUP	Error		The group is non-administered.
NOTADMIN	Error		Sender is not administrator.
a	Internal		GCS checks the group informations (for *IsAdmin* and *Admin*).
b	Cond		Sender is the administrator.
c	Internal		Set the *IsOpened* attribute in *infos*.
d	Cond		The group is non-administered.
e	Cond		Administered group, but sender is not administrator.

5.11 Change Private Attribute

A CHANGEPRIVATTR request, together with a *NewPrivAttr* parameter, causes the GCS checks its local information (**a**). If the sender is the group administrator (**b**), then the current *IsPrivate* attribute is modified (**c**) and this results into PRIVATTR-CHANGED. When the group is non-administered (**d**), the result becomes a NOAD-MINGROUP error. Otherwise, if the sender is not the administrator (**e**), then a NOTADMIN error is sent back.

FIGURE 22. Change Private Attribute UCM

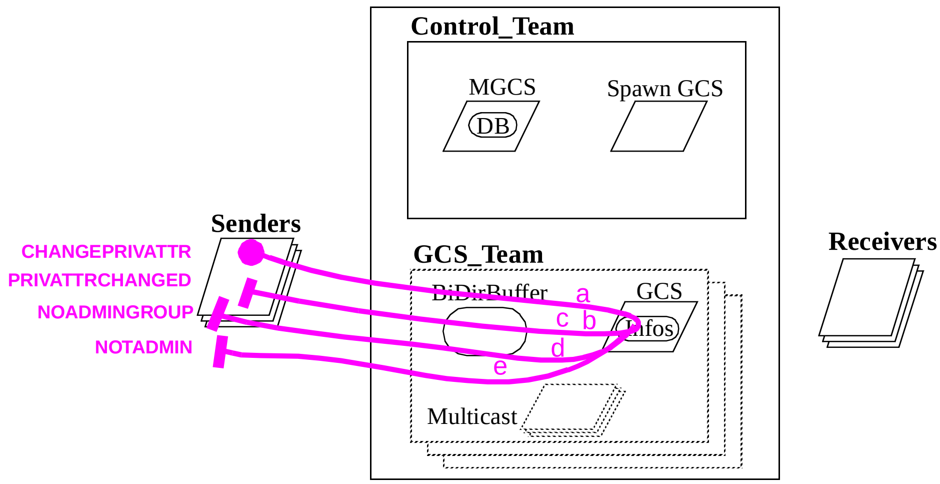


TABLE 12. Responsibilities of “Change Private Attribute UCM”

Responsibilities	Type	Input/Output	Comment
CHANGEPRIVATTR	Request	*NewPrivAttr*	Change the$group *IsPrivate* attribute.
PRIVATTRCHANGED	Ack		The *IsPrivate* attribute has been changed.
NOADMINGROUP	Error		The group is non-administered.
NOTADMIN	Error		Sender is not administrator.
a	Internal		GCS checks the group informations (for *IsAdmin* and *Admin*).
b	Cond		Sender is the administrator.
c	Internal		Set the *IsPrivate* attribute in *infos*.
d	Cond		The group is non-administered.
e	Cond		Administered group, but sender is not administrator.

5.12 Change Moderator

The occurrence of a CHANGEMODER request causes the GCS to check its local information (**a**). If the sender is the moderator (or the administrator) and the proposed *NewModer* a member of the group (**b**) (or *Nobody*), then *Moder* and *IsModerated* are updated accordingly (**c**) and this results into MODERCHANGED.

Such a request sent to a non-moderated group (**d**) results in a NOMODERGROUP error, unless the sender is the administrator. When the group is moderated, if the sender is neither the moderator nor the administrator (**e**), then a NOTMODER error is sent back. If the group is closed, the new moderator must also be a member of the group, otherwise (**f**) MEMBERNOTINGROUP is returned.

Upon a validated change, the new moderator could be advised of its new role. We have not included this functionality as it is not part of the informal requirements.

FIGURE 23. Change Moderator UCM

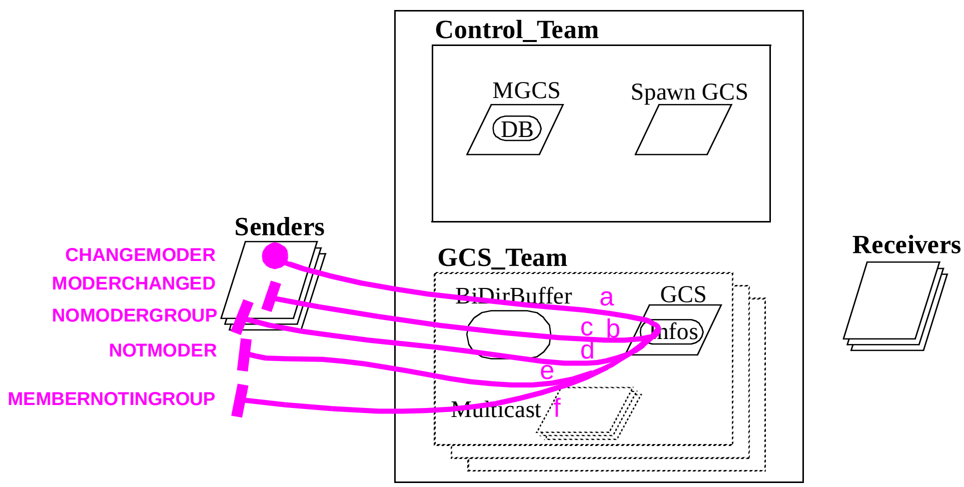


TABLE 13. Responsibilities of “Change Moderator UCM”

Responsibilities	Type	Input/Output	Comment
CHANGEMODER	Request	*NewModer*	Change the group moderator and the *IsModerated* attribute.
MODERCHANGED	Ack		The moderator and the *IsModerated* attribute have been changed.
NOMODERGROUP	Error		The group is non-moderated, and sender is not the administrator.
NOTMODER	Error		Sender is not moderator (or administrator, if any).
MEMBERNOTIN-GROUP	Error		The proposed moderator is not in the closed group (and is not *Nobody*).

TABLE 13. Responsibilities of "Change Moderator UCM"

Responsibilities	Type	Input/Output	Comment
a	Internal		GCS checks the group informations (for *IsAdmin*, *Admin*, *IsOpened*, *IsModer*, *Moder* and *mbrL*).
b	Cond		Sender is the moderator (or the administrator, if any). If the group is closed, then the new moderator is in the group.
c	Internal		Set the new moderator and the *IsModerated* attribute in *infos*.
d	Cond		The group is non-moderated, and the sender is not the administrator (if any).
e	Cond		Moderated group, but sender is not moderator (or administrator, if any).
f	Cond		Sender is moderator (or administrator), but the group is closed and *NewModer* is not a group member.

5.13 Chapter Summary

The twelve functionalities of our *Group Communication Server*, informally described in section 3.3, lead to the definition of twelve UCMs, one for each functionality. We presented the scenarios on top of the selected structure for a better understanding of the responsibilities allocation. Each UCM was further explained with a table that contains the description of the responsibilities and their type, input/output parameters, and comments. Conditions were expressed in English as we did not have a precise definition of data types, data structures, variables and databases. When such definitions are available, we can structure conditions and results more formally with, for instance, Parnas tables [Parnas *et al*., 1994] (refer to Table 14 for an example).

Chapter 6 Synthesizing a LOTOS Specification for the GCS

This chapter gives an overview of the synthesis of a LOTOS specification of the Group Communication Server (presented in Appendix A), whose functionalities were expressed as a set of Use Case Maps (scenarios).

6.1 Synthesis Strategy

6.1.1 Manual Synthesis Instead of TMDL

In [Amyot, 1994 and 1994a], the author presented a methodology for the semi-automated generation of LOTOS specifications from UCMs. The maps were manually described using the *Timethread Map Description Language* (**TMDL**)[1], and then a compiler (*tmdl2lot*) would generate the specification automatically.

Appendix B presents an example of the GCS Group Creation (Section 5.1) in TMDL. The description is given according to the UCM of Figure 12. Furthermore, we included the resulting LOTOS specification and its expanded behaviour.

Although this approach has been successfully used for a simple telephony system (Amyot *et al.*, 1995), we foresaw three major difficulties for our specific problem:

- **Data types**: TMDL does not have data types, and the GCS functionalities rely heavily on data for databases and conditions. In the example (Appendix B), we notice that databases and guards have to be explicitly and artificially inserted.
- **Composition**: In TMDL, the designer has to provide a global map where all scenarios are correctly composed. In our case, we would have to explicitly compose twelve UCMs, one for each GCS functionality. Generating such a map would result in a very large picture, difficult to understand. Stubs (a new operator used to structure large UCMs) could have help in this case, but they are not part of TMDL yet. We would rather have a composition based on the satisfaction of preconditions.
- **Components**: TMDL does not consider any structural artifact. Use case paths (timethreads) are the only type of object we can use. We call these *unbound maps* as their responsibilities are not bound to components. The resulting specification becomes consequently purely functional in nature (like a service specification, without any message passing). However, we would rather generate a design close to a future implementation where we find components, links (channels) and messages.

For these reasons, and also because scenarios represent only partial views of the system (another motivation for human intervention at the synthesis level), we opted for a direct and manual generation of the specification (see Appendix A), although it was rigorously inspired from the UCM. The specification is to be validated later on against the requirements, through test cases derived from the UCM. Note that the formalization of the UCM notation is underway. We are working on the definition of a XML-compliant (*eXtensible Markup Language*) grammar where stubs, plug-ins, and components will be supported. Data will be usable, to some extent, through the use of descriptions hooked

1. Use Case Maps were previously called *Timethread Maps*.

to the elements of a use case path. UCM textual descriptions are to be automatically generated from a UCM graphical editor (*UCM Navigator*), still under construction.

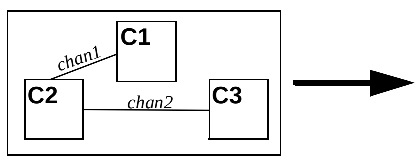
6.1.2 Some Guiding Rules

To synthesize the model from the UCMs in a **rigorous** and **traceable** way, we followed three general guiding rules:

- Responsibilities are allocated to paths and components. This is the way a complete binding between the paths and the structure is achieved.
- Components implement the causal relationships of the paths that go through them (intra-component causality, or *roles*), as shown in Figure 25. These roles can be regrouped using choice (one role or the other) or concurrency semantics (simultaneous roles).
- Causal relationships between components are refined as exchanges of messages, leading to the definition of **protocols** (inter-component causality).

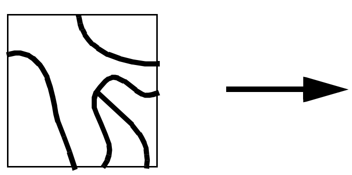
Using the examples from Figure 5 and Figure 6, we can illustrate further the synthesis approach in a LOTOS context. The structure of components is usually mapped to a structure of processes synchronized on the appropriate communication channels (Figure 24). The internal component behaviour is derived from the inside paths (roles), and this becomes the internal behaviour of the corresponding LOTOS process.

FIGURE 24. From Architecture to Process Structure




```
specification GCS[...]:noexit
... (* ADTs *)
behaviour
(* Components structure from structure *)
    C1[...]
    |[chan1]|
    ( C2[...] |[chan2]| C3[...] )
where
(* Process definitions for C1, C2, and C3*)
    ...
endspec (* GCS *)
```

FIGURE 25. From Multiple Paths to Process Behaviour



```
process C1[...]:noexit
(* Component behaviour from roles *)
    T; a; m1; C1[...]
    []
    (
        T2; a2; stop
        |||
        T3; a3; C1[...]
    )
    []
    ...
endproc (* C1 *)
```

These rules are independent from the target prototyping language. For instance, SDL, Estelle, or ROOM could be used. The application of these rules also promote bidirectional traceability. The following sections present, in more details, how we interpreted these rules with respect to LOTOS, in the context of our GCS system.

6.2 General Structure

6.2.1 Names

For a better traceability between the specification `Group_Communication_Service` (Appendix A) and the underlying structure of components (Figure 11), we mapped every component, i.e. teams, processes, and objects, onto LOTOS processes. Teams are basically simple containers that instantiate other processes. This mapping allows the preservation of naming conventions, and so a problem detected in the specification can be directly related to its design and to the informal requirements. However, databases (object *DB* and object *Infos*) were not considered as components; they are located within processes and do not really communicate. We simply interpreted them as abstract data types used as process parameters in the LOTOS specification.

All requests, acknowledgements, and errors were specified using the same names as in the UCM. They are grouped under two abstract data types: `RequestType` and `AckErrorType` (lines 440 to 528). Events interpreted as data reduce the number of gates in the specification, leading to a lower complexity and a higher maintainability.

6.2.2 Event structures

Communication between the clients (senders and receivers) and the server is done via strict message (*event*) structures within the protocols. In LOTOS, communication over gates (channels) is synchronous, untyped, and directionless[1]. These characteristics make the traceability a difficult task. We can get around this problem by using strict LOTOS event structures that allow one to syntactically determine, from execution traces, the sender and the receiver of a message. Extra values passed on the gate (e.g., the direction) might be then required.

Message types are derived similarly to LOTOS' gate merging transformation [Bolognesi *et al*., 1995]. Request, error, and acknowledgement events are transformed from events (potentially interpreted as gates) to parameters associated to a gate. For instance, REGISTER and MEMBERS events are mapped to the *gcs_ch* channel to form new events such as `gcs_ch!REGISTER` and `gcs_ch!MEMBERS`. Backward traceability would therefore be similar to gate splitting. Gate merging/splitting is a useful mechanism for describing traceability relationships.

The following event structures are the ones used in our specification. They help us finding out senders and receivers from execution traces. Those on `gcs_ch`, `out_ch` and `mgcs_ch` are visible, those on `inter_ch`, `sgcs_ch` and `agcs_ch` are hidden to the external world. The direction has to be specified on bidirectional channels `mgcs_ch`, `gcs_ch`, and `inter_ch` (see Figure 11).

Please note that terms between curly brackets (`{...}`) represent optional parameters.

Between Senders and Group_Communication_System (GCS_Team) on `gcs_ch`

- Request from a sender to BiDirBuffer (within an instance of GCS_Team)
  `gcs_ch !ToGCS !sender:MID !groupid:GID !req:Request !msg:Msg;`

1. However, gates will be typed in the upcoming E-LOTOS [Quemada, 1997], which will represent a major improvement in this case.

- Acknowledgement from BiDirBuffer (within an instance of GCS_Team) to a sender
`gcs_ch !FromGCS !sender:MID !ack:AckError !groupid:GID;`

Between Senders and Group_Communication_System (MGCS) on `mgcs_ch`

- Request from a sender to MGCS
```
mgcs_ch !ToMGCS !caller:MID !req:Request {!newgroupid:GID
                                          !infos:Msg};
```
- Acknowledgement from MGCS to a sender
`mgcs_ch !FromMGCS !caller:MID !ack:AckError {!newgroupid:GID};`

Between Receivers and Group_Communication_System (GCS_Team) on `out_ch`

- Message multicast from an instance of GCS_Team to a receiver (member)
`out_ch !receiver.channel:MBR !sender:MID !msg:Msg;`

Between Control_Team and GCS_Team (within Group_Communication_System) on `gcs_ch`

- Group deletion announcement from GCS_Team (BiDirBuffer) to Control_Team (MGCS)
`agcs_ch !GROUPDELETED !groupid:GID;`

Between MGCS and Spawn_GCS (within Control_Team) on `sgcs_ch`

- Group creation request from MGCS to Spawn_GCS
`sgcs_ch !CREATEGROUP !groupid:GID !mbrL:MemberList !infos:Msg;`

Between BiDirBuffer and GCS (within GCS_Team) on `inter_ch`

- Forwarded request from BiDirBuffer to its GCS
`inter_ch !ToGCS !sender:MID !req:Request !msg:Msg;`
- Acknowledgement to be forwarded, from a GCS to its BiDirBuffer
`inter_ch !FromGCS !sender:MID !ack:AckError;`
- Final announcement of a deletion, from BiDirBuffer to its GCS
`inter_ch !ToGCS !GROUPDELETED;`

The previous definitions can be used as patterns for the extraction of useful information from executions traces. For instance, this information could be used to better visualize the traces as Message Sequence Charts (see Appendix D).

6.2.3 Patterns and Styles

Several known LOTOS patterns have been reused in the specification. Among these, the *Installer* pattern for dynamic creation of objects [Tuok, 1996] was adapted for the spawning of new groups (lines 907 to 916) and the creation of concurrent multicast threads (lines 1528 to 1562). An *Installer* (shown below) uses recursion and concurrency to simulate the dynamic instantiation of objects.

```
process Installer[g](usedObjIds:SetObjIds):noexit:=
    g ?newObjId:ObjectId [NotId(newObjId,usedObjIds)];
    (
        Object[g](newObjId)
        |||
        Installer[g](Insert(newObjId, usedObjIds))
    )
where
    process Object[g](myId:ObjectId):noexit:=
        ... (* process behaviour *)
    endproc (* Object *)
endproc (* Installer *)
```

Three common LOTOS specification styles [Vissers *et al.*, 1991] were used in the specification. We represent the structure using a resource-oriented style, where component behaviour is mostly in state-oriented or monolithic style (guarded alternatives). The test processes are almost always in monolithic style.

6.3 Data Types

The data part of LOTOS is based on the algebraic language ACT ONE. *Abstract data types* (ADT) are defined in terms or sorts, operations (signatures) and equations. A type can be defined in terms of other types by means of renaming, extension, or actualization.

We use the booleans, natural numbers, and hexadecimal numbers, as defined in the International Standard (IS) library (ISO, 1988), as the basis for our abstract data types. However, we modified three types (`FBoolean`, `Element`, and `Set`) in order for their equations to be used as rewriting rules for our simulation tools (LOLA and XELUDO).

With renaming, most identifiers were simply represented as enumerations. Hence, we created a formal type `EnumType`, a subset of the natural numbers. We actualized (subclassed) it for `MIDType`, `CIDType`, `GIDType`, `ChanType` and `InfoMsgType`. This is a simple and efficient way of reusing the comparison operations already defined in the IS.

The IS `Set` type was extended to implement several list types such as `MIDListType`, `MemberListType`, and `GroupListType`.

The characteristics of a group, or *group information*, was represented as a tuple (`GCSinfoRecordType`, line 405) whose fields include administrative details concerning the identification and group type, administration, moderation, openness, and privacy.

The encoding of message packets sent to the server is done with `MsgType` (line 549). Its main operation, named `Encode`, takes a list of parameters which may not be the same for each request. Therefore, it makes these different parameter lists uniform under a same sort. We also provide many equations to extract the values from these parameters.

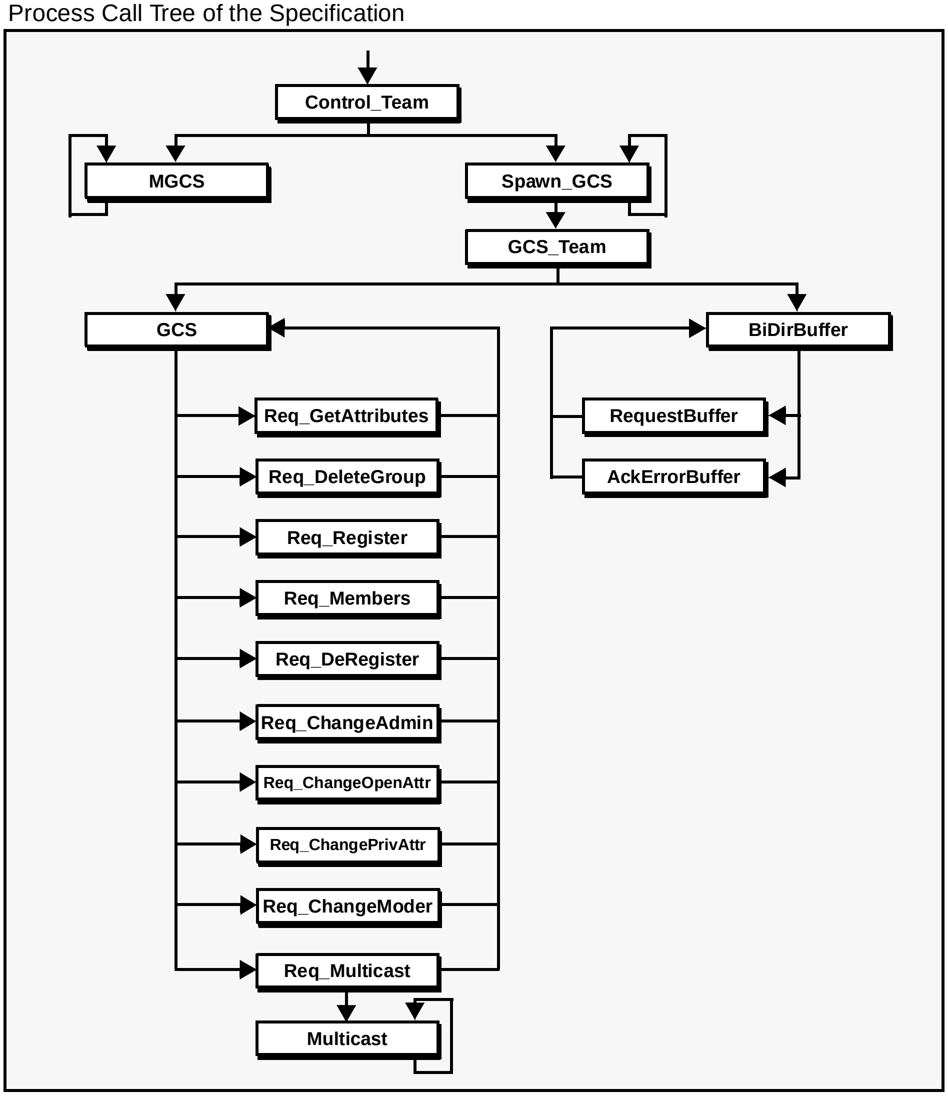
The last types declared in the specification are based on `FIFOType` (line 724), a generic FIFO list type that is actualized as FIFO buffers for requests (`FIFOreqsType`, line 792) and acknowledgements/errors (`FIFOackerrsType`, line 807). BiDirBuffer uses them to store and forward requests and acknowledgements.

6.4 Processes

6.4.1 Process Call Tree

Nineteen processes were used to describe the system. Figure 26 illustrates the process call tree of the specification. We notice two processes that act as containers (Control_Team and GCS_Team) and five recursive processes (`MGCS`, `Spawn_GCS`, `GCS`, `BiDirBuffer` and `Multicast`) that correspond exactly to the components defined in the structure (Figure 11). Ten processes (starting with `Req_`) are sub-processes within GCS, one for each functionality. BiDirBuffer also uses two subprocesses. They were defined only to structure the specification more clearly.

FIGURE 26. Process Call Tree of the Specification



6.4.2 Parnas Tables

We can use a special type of decision tables (*Parnas tables*) [Parnas *et al.*, 1994] in order to explain the behaviour of processes having a high degree of complexity due to multiple conditions and cases. These multiple cases, emerging from guards that include complex ADT equations, will help us ensure that all cases are covered (*completeness*) and that all alternatives are mutually exclusive (*deterministic*).

Table Notation

We use the following notation (see Figure 27). Apostrophes are used to indicate variable states: the current state of a variable V is shown by ‘V, and the next state by V’. Since we not only deal with new values assigned to variables but also with exchanges of messages (events on gates), we will associate a special symbol (**!**) to a gate name through which a message (whose value is specified in the table) is sent. We also use the dot notation (**.**) to access fields within complex data structures such as records. **NC**(*variables*) means that these *variables* remain unchanged at the end of the process.

FIGURE 27. Table Notation

Conditions →

	Condition	*Not(Condition)*
Variable1’	‘Variable1 + 1	‘Variable1
GateName2 !	Message1	Message2
Record.Field1’	0	1

Names ↑ Values ↑

An Example: Process Req_ChangeAdmin

This process (lines 1334 to 1371) manages the Change Administrator operation shown in Figure 20. It is very complex in nature as it contains many conditions and possible outcomes.

TABLE 14. Req_ChangeAdmin Table Description

<table>
<tr><th><code>Req_ChangeAdmin</code></th><th colspan="5">External variables: <code>sender:MID, msg:Msg, id:GID, mbrL:MemberList, infos:Msg</code></th></tr>
<tr><td colspan="6">R =</td></tr>
<tr><td rowspan="4"></td><td colspan="4">‘infos.IsAdmin = Administered</td><td rowspan="4">‘infos.IsAdmin ≠ Administered</td></tr>
<tr><td colspan="3">‘infos.Admin = ‘sender</td><td rowspan="3">‘infos.Admin ≠ ‘sender</td></tr>
<tr><td rowspan="2">‘msg.Admin ∈ ‘mbrL</td><td colspan="2">‘msg.Admin ∉ ‘mbrL</td></tr>
<tr><td>‘msg.Admin = NoBody</td><td>‘msg.Admin ≠ NoBody</td></tr>
<tr><td>infos.IsAdmin’</td><td>‘infos.IsAdmin</td><td>False</td><td>‘infos.IsAdmin</td><td>‘infos.IsAdmin</td><td>‘infos.IsAdmin</td></tr>
<tr><td>infos.Admin’</td><td>‘msg.Admin</td><td>NoBody</td><td>‘infos.Admin</td><td>‘infos.Admin</td><td>‘infos.Admin</td></tr>
<tr><td>infos.IsPrivate’</td><td>‘msg.IsPrivate</td><td>False</td><td>‘infos.IsPrivate</td><td>‘infos.IsPrivate</td><td>‘infos.IsPrivate</td></tr>
<tr><td>inter_ch !sender !</td><td><code>ADMINCHANGED</code></td><td><code>ADMINCHANGED</code></td><td><code>MEMBERNOTINGROUP</code></td><td><code>NOTADMIN</code></td><td><code>NOADMINGROUP</code></td></tr>
<tr><td colspan="6">∧ NC (<code>sender</code>, <code>msg</code>, <code>id</code>, <code>mbrl</code>, other fields of <code>infos</code>)</td></tr>
</table>

This table corresponds to the behaviour of the LOTOS process combined to the equations within the abstract data types. The expressions in the table can be written using some logic notation (for instance, *‘msg.Admin ∉ ‘mbrL*), or using the ADT operations from the specification (*‘msg.Admin NotIn ‘mbrL*). The latter would be less generic, but would simplify the test case generation.

In theory, the whole specification could be described using a hierarchical collection of such tables. However, we believe the task would be tedious and counterproductive. Nevertheless, their usefulness in the description of critical sections of the specification proved to be effective. They help structuring the conditions and ensuring the coverage of all possibilities in a deterministic way.

Moreover, test cases can be derived more systematically, thus ensuring the coverage of all cases with a comprehensive test suite. For instance, the number of possible results in Table 14 indicates to the test designer that (at least) five test cases are necessary to cover all possibilities. We might have been tempted, from the UCM in Figure 20, to derive only four test cases as there no visual clue that indicated that there were two ways to get `ADMINCHANGED` (although this was more explicit in the comments on the conditions). In our test suite, we effectively derived five acceptance test cases for this scenario (see Test_17, line 2518).

6.5 Multicast

This specification concentrates on the management of groups and members, but it also exhibits a multicasting functionality. The `Multicast` process (lines 1518 to 1582) specifies a simple protocol where a message is sent to all group members, concurrently. No other messages can be sent by a specific GCS until the first one is sent to all members. There is no acknowledgement whatsoever (unless the medium ensures it).

This specification is structured in such a way that another multicast protocol could be *plugged-in*, or we could even extend it in order to have a selection of protocols. Appendix C describes three such protocols:

- **Sequential Multicast** (Appendix C.1): instead of sending the messages to receivers concurrently, the sending is done sequentially in a LIFO order.
- **Best Effort Sequential Multicast** (Appendix C.2): as for Sequential Multicast, the sending is done sequentially in a LIFO order. However, problems may occur on the sending of messages, or on their reception if the sending is synchronous. This protocol includes a time-out mechanism to ensure that such failures do not block the protocol. The number of successful messages sent is also counted.
- **Broadcast** (Appendix C.3): instead of using point-to-point communication, we assume some underlying broadcast mechanism (such as IP broadcast) to be used to send a message to all group members at once. Receivers are responsible for the filtering of relevant messages based on their belonging to specific groups.

These processes are included to show the modularity of the specification and the ease with which we can make the multicast protocol more complex without affecting the rest of the specification. Some test cases would however have to be changed according to the new constraints or new flexibility. For instance, a system using Sequential Multicast would output messages in LIFO order, and not in any order.

6.6 Chapter Summary

We provided general guiding rules for the synthesis and gave a general overview of the specification and how it was generated. We explained why TMDL did not seem to be suitable for this system, especially wit respect to data types, scenario composition, and component description. We therefore had to synthesize the specification manually, with the UCM as the main inspiration for the definition of our components’ behaviour.

While defining the structure of the specification, we considered several issues for future needs. We kept the names of components and messages as defined in the requirements, therefore improving the traceability between the different design phases. Well-defined event structures were used to represent components interactions, leading to an efficient interpretation of the execution traces. These traces could be converted to MSCs, as shown in Appendix D. Well-known LOTOS specification styles and patterns helps in the organization and the readability of the specification. Most of our data types reuse or extend ADTs from the IS library, hence making their understanding and implementation easier.

The specification processes and the calling structure were introduced. We showed how we could describe some critical processes using Parnas tables. We believe they are useful for documentation and for ensuring that a set of complex conditions is complete and deterministic. They can also help in getting a better structuring of the conditions in a process, and in defining equivalence classes for test cases.

Finally, we highlighted the modularity of the specification by showing how our multicast protocol could be substituted by other ones, with little impact on the rest of the specification. Three different multicast protocols are presented in Appendix C.

Chapter 7 Validating the GCS Specification Throught Testing

We present the techniques used to validate our LOTOS specification against the requirements. We first review the validation and testing theory in our specific context, then we derive the validation test suite from the Use Case Maps (scenarios), apply the tests according to the LOTOS testing theory, present the results, give a measure of the coverage, and complete the test suite when necessary.

7.1 Three Approaches to LOTOS-Based Validation

Three of the most common approaches to the validation of a LOTOS specification against (informal) requirements are equivalence checking, model checking, and functionality-based testing.

- **Equivalence checking** usually requires a formal representation of (part of) the requirements, seldom available in the early stages of the design process. However, this approach is most useful when checking the conformity of one specification against another, after some refinement or modifications.
- **Model checking** aims to validate a specification against safety, liveness, or responsiveness properties derived from the requirements. These properties can be expressed, for instance, in terms of temporal logic or μ-calculus formulas. In the LOTOS world, this technique usually requires that the specification be expanded into a corresponding model, which is some graph representation (labelled transition system, finite state machine, or Kripke structure) of the specification's semantics. On-the-fly model checking techniques, where the whole model does not have to be generated a priori, exist as well. Since the validation is at a semantics level, unreachable code will hardly be detectable, simply because it will not be expanded. Also, the languages used to define properties are very flexible and powerful, yet they can be quite complex; it is a difficult problem to determine whether a property really reflects the intents of informal requirements.
- **Functionality-based testing** is concerned with the existence (or the absence) of traces, use cases, or more generally scenarios in the specification. These scenarios reflect system functionalities, usually in terms of operational or user-centered instances of intended system behaviour. They can easily be transformed into black-box test cases that can be composed with the specification for validating the latter against requirements. Test cases are often more manageable and understandable than properties, and they relate more closely to informal requirements. However, they are usually less powerful and expressive than liveness or safety properties expressed in temporal logic.

Among these three approaches, we favored functionality-based testing for the validation of the GCS. Equivalence checking was not possible because we aimed to produce a first high-level specification from informal requirements. Since these requirements were expressed mostly operationally, scenarios (UCMs) were easier to extract than properties, so model checking was not used at first. Note however that these approaches are in general not mutually exclusive, but complementary.

7.2 Testing Concepts

Although *testing* is discussed most commonly in the sense of implementation testing, executable specifications can also be tested in order to see whether they satisfy requirements. Some authors call this activity *validation*, but many of the methods and concepts of implementation testing apply. For this reason, in this report we use these terms interchangeably.

7.2.1 Goal

The ultimate goal of testing is to detect errors as soon as possible, especially in the specification. A good test is a test that highlights a fault in the specification. A good test suite is a test suite that covers, under some hypothesis and assumptions, critical aspects, if not all aspects, of a specification.

In our specific case, we plan to validate the specification against the functional requirements by using a test suite derived from the scenarios (UCMs). Users and designers can both *inspect* the UCMs derived from the requirements, thus establishing their validity. UCMs are defined at a level of abstraction that is efficient for early inspection of the system design. Inspection is known to be a very cost-effective quality improvement technique, especially for requirements documents [Johnson, 1998]. Oppositely to inspection, testing require an executable or formally defined artifact, which is, in our case, the LOTOS prototype resulting from the synthesis phase. Testing acts as an essential supplement to inspection for detecting behavioural problems.

We believe that tests derived from UCMs are closer to the requirements than a whole specification. In other words, the gap between the definition of a test and the requirements is smaller than the gap between the definition of the entire specification and those same requirements. The human mind can handle the level of complexity associated to a scenario or a test case, especially during inspection, but not the one associated to a complex specification. Moreover, tests become formal representations of partial requirements and can be used to formally detect faults or errors in a specification. Hence, testing helps us validate the specification with respect to the requirements before going on to the next stage of the development cycle.

Once errors that have been detected are corrected, we would like to assess the coverage of our abstract test suite in order to check that it is sufficient according to some criteria. In our case, we want to achieve a functional coverage based on the functionalities expressed in the requirements and based on the syntactic structure of the specification.

When the coverage is achieved, the abstract test suite can be reused for regression testing (when we modify the requirements) or for testing further refinements leading to the implementation, and ultimately the implementation itself.

7.2.2 General Notions

In the context of formal conformance testing, methods usually assume that both the specification and the implementation can be modeled in the same way. In our case, we intend to check the validity of a specification (or high-level design) with respect to informal requirements. The latter are obviously not modeled formally. We will assume that a collection of test cases, often called *(abstract) test suite*, represent formal partial models of our requirements. The testing of the validity relation will therefore occur between the specification and the test suite.

According to *Formal Methods in Conformance Testing* (FMCT) [ISO, 1996], a test suite can be:

- **Exhaustive**: all passing implementations are compliant to the specification.
- **Sound**: all implementations that do not pass are not compliant.
- **Complete**: it is both sound and exhaustive,

In our context, which is different from traditional conformance testing, the term "implementation" becomes "*specification under test* (SUT)", and "specification" becomes "*requirements*". If a test suite is neither sound nor exhaustive, then nothing concerning conformance or validity can be concluded by means of testing.

Pragmatically, it is not possible to construct a finite exhaustive test suite for most real-life systems. Consequently, we aim to produce a sound test suite using the scenarios (UCM) already available. Any error detected by a sound test suite proves that the SUT is incorrect, but not finding an error does not mean that the SUT is without errors. Optimization of such test suites targets the minimization of the number of test cases and their complexity/length/cost, and the maximization of the discriminatory power of the tests. A test suite TS_1 is said to discriminate more than another one (TS_2) if TS_1 finds faults in more specifications than TS_2.

The *Conformance Testing Methodology and Framework* (CTMF) [ISO, 1991] details the definition of a an **abstract test suite** as being composed of **test groups**. Each group consists of several **test cases** according to a logical ordering of execution. A test case contains **test steps**, each of which consists of several **test events**, the atomic interactions between the tester and the implementation or SUT.

A test case is often composed of several components:

- **Test purpose**: describes the objective of the test case (expected behaviour, verification goal, etc.).
- **Test preamble**: contains the necessary steps to bring the SUT into the desired starting state.
- **Test body**: defines the test steps needed to achieve the test purpose.
- **Test postamble**: used to put the SUT into a stable state after a test body is executed.

Test cases for *finite state machines* (FSM) usually have a test body that contains one transition followed by a **Test Verification** (checking sequence, unique input/output, distinguishing sequence, etc.), which identifies the target state. A preamble may also contain a verification sequence that checks the initial state. However, in most test suites, the initial state resulting from the preamble has already been checked as a target state in a previous test case.

7.2.3 Combination of Techniques

There exists an enormous number of testing techniques for formal methods, but most fall into one of the three following categories:

- **Black box**: testing based on the externally visible behaviour.
- **White-box**: testing based on the internal structure of a specification or program.
- **Grey-box**: testing based on the design.

Our approach uses ideas from all these categories. We focus mainly on causes and effects with UCMs (grey-box), on LOTOS testing (black-box), on coverage measurement techniques (white-box), and on the use of relevant data values (boundary analysis and equivalence classes, i.e., black-box testing).

In this report, we used several guidelines and assumptions related to testing. The genericity of our UCMs already provides us with implicit equivalence classes of data and behaviour. We will try to take advantage of this characteristics of UCMs. The focus will also be on deterministic test cases (as sequences of events) whenever possible, i.e., when the specification under test is deterministic. They usually lead to faster executions and simpler interpretations of the results. Finally, recursion will be dealt with to a short extent only, unless requirements express critical warnings related to this issue.

7.2.4 LOTOS Testing

LOTOS exhibits interesting static semantics features, implemented in most of its compilers and interpreters. The successful compilation of a LOTOS specification ensures that several dataflow anomalies, such as the use of an undefined or unassigned value identifier (variable), cannot occur. Since most of these problems are automatically avoided, we shall not consider them further in our approach.

Dynamic behaviour, however, is a totally different story. This is where testing can help. The LOTOS testing theory has a test assumption stating that the implementation (the SUT in our case), modeled as a *Labeled Transition System* (LTS), communicates in a symmetric and synchronous way with external observers, the test processes. There is no notion of initiative of actions, and no direction can be associated to a communication.

Definitions

This section gives an overview of this theory in our context. To keep the list of definitions short, we reuse the concepts of *canonical tester* (*CT(S)*), *testing equivalence* (te), *reduction* (red), and *conformance* (conf) relations, as defined in [Brinksma, 1988]. We will however define them informally after the following list of definitions for specification, testing, and relation domains:

Specification Domain

- *Req* : Informal requirements.
- **SPECS** : Universe of specifications.
- *S, S1, S2...*: Specifications. $S \in$ **SPECS**, $S1 \in$ **SPECS**, $S2 \in$ **SPECS**, ...
- *SUT* : Specification Under Test. $SUT \in$ **SPECS**.

- *UCMs* : Set of LOTOS interpretations (behaviour) of Use Case Maps used for the generation of specification *SUT*. $UCMs \subseteq$ **SPECS**.
- UCM_n : Use case map *n* used for specification *SUT*. $UCM_n \in UCMs$.

Testing Domain

- **TESTS** : Universe of test cases. In LOTOS, tests are also specifications: **TESTS** $\subseteq$ **SPECS**.
- *CT(S)* : Canonical tester of specification *S*. $CT(S) \in$ **TESTS**.
- *TS* : Test suite (set of test cases) for specification *SUT*. $TS \subseteq$ **TESTS**.
- TG_n : Test group *n*. $TS = \bigcup_{g=1}^{n} TG_g$
- T_x : Test case *n*. $\forall T_x, \exists TG_g \mid T_x \in TG_g \wedge TG_g \subseteq TS$.
- $VP(T_x)$: Set of visited probes for T_x.
- $TP(T_x)$: Test purpose of T_x.

Relation Domain

- conf : Conformance relation. conf $\subseteq$ **SPECS** x **SPECS**.
- red : Reduction relation. red $\subseteq$ **SPECS** x **SPECS**.
- te : Testing equivalence relation. te $\subseteq$ **SPECS** x **SPECS**.
- val : Validation relation. val $\subseteq$ **SPECS** x *Req*.

In terms of traces (obtained from their respective LTS), *SUT* conf *S* expresses that testing the traces of *S* against the behaviour of *SUT* will not lead to deadlocks that could not occur with the same test performed with *S* itself (no unexpected deadlock can occur). This relation is mainly used for conformance testing. In LOTOS as both the specification and the tests are represented as processes, *S* could simply be a test case.

The reduction relation states that *S1* red *S2* if *S1* can only execute actions that *S2* can execute, and *S1* can only refuse actions that can be refused by *S2*. We say that *S2* is *irreducible* if *S1* red *S2* $\Rightarrow$ *S1* te *S2*.

Two specifications are testing equivalent (*S1* te *S2*) if they cannot be distinguished by any test case. Equation 1 shows an interesting property of these relations:

$$S1 \ \underline{te} \ S2 \Leftrightarrow S1 \ \underline{red} \ S2 \wedge S2 \ \underline{red} \ S1 \Leftrightarrow S1 \ \underline{conf} \ S2 \wedge S2 \ \underline{conf} \ S1 \quad \textbf{(EQ 1)}$$

Two test cases also have the same detectability power if they are testing equivalent, i.e., when T_x te T_x. Therefore, this property applies to irreducible test cases.

Every specification *S* has a canonical tester *CT(S)*, a complex process that tests *S* completely according to te. Many such testers exist for any given specification, and they are all testing equivalent with each other. *CT(S)* represents the only test case necessary to check that a specification *S1* conforms to *S* (*S1* conf *S*). An interest property is that *CT(CT(S))* te *S*.

For most realistic specifications, a canonical tester cannot be directly generated as it may be infinite, especially when data values or recursive processes are involved. Also, since these testers are usually non-deterministic, the use of canonical testers on implementations does not guarantee that an error will be highlighted (by an unexpected deadlock). Consequently, *CT(S)* should not be used for testing conformance directly, but only used to guide the generation of an adequate test suite (with deterministic test cases) from it. There exist more simplified canonical testers [Leduc, 1991] for implementation relations slightly different from our conf relation, but they will not be considered in this report.

Test Suites and Verdicts

A correct test case is a reduction of the specifications's canonical tester (T_x red *CT(S)*). To verify the successful execution of a test case, such a test process T_x and the specification under test *SUT* are composed in parallel, synchronizing on all gates but one (a *Success* event). If a deadlock occurs prematurely, i.e, if *Success* is not always reached at the end of each branch of the LTS resulting from this composition, then the *SUT* failed this test. If this is not the case, then it must have passed the test. On the basis of ideas found in [Brinksma *et al.*, 1991], we present more formal definitions of these notions:

- *ACCEPT* : Set of acceptance test cases (**Must tests**). $ACCEPT \subseteq TS$.
- *REJECT* : Set of rejection test cases (**Reject tests**). $REJECT \subseteq TS$.
- passes : Pass relation for one test case: passes ⊆ **SPECS** x **TESTS**.
  Pass relation for a test suite: passes ⊆ **SPECS** x *PowerSet(***TESTS***)*.
- fails : Failure relation for one test case: fails ⊆ **SPECS** x **TESTS**.
  Failure relation for a test suite: fails ⊆ **SPECS** x *PowerSet(***TESTS***)*.
- failsall : Failure relation for one test case: failsall ⊆ **SPECS** x **TESTS**.
  Failure relation for a test suite: failsall ⊆ **SPECS** x *PowerSet(***TESTS***)*.

The difference between fails and failsall is that the *Success* event is never reached in failsall, while it may be so for some test runs in fails as long as at least one test run leads to a deadlock (or to an infinite loop).

- *SUT* passes $T_x \Leftrightarrow \forall$trace *t* in *SUT* |[all gates but *Success*]| T_x, *t* reaches *Success*.[1]
- *SUT* fails $T_x \Leftrightarrow \neg$(*SUT* passes T_x).
- *SUT* failsall $T_x \Leftrightarrow \forall$trace *t* in *SUT* |[all gates but *Success*]| T_x, *t* does not reach *Success*.
- *SUT* passes *TS* $\Leftrightarrow \forall T_x \in TS$, *SUT* passes T_x.
- *SUT* fails *TS* $\Leftrightarrow \neg$(*SUT* passes *TS*) $\Leftrightarrow \exists T_x \in TS$, *SUT* fails T_x.
- *SUT* failsall *TS* $\Leftrightarrow \forall T_x \in TS$, *SUT* failsall T_x.

This testing theory, inspired from [Hennessy, 1988] is implemented in the tool LOLA [Quemada *et al.*, 1993], which expands this composition to analyze whether the execu-

1. Note that an infinite loop is not considered to be a successful execution.

tions reach the success event or not. Three *verdicts* can occur after the execution of one test case:

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

In the real world, test cases must be executed more than once when there is non-determinism in either the test or the implementation (under some fairness assumption). However, LOLA avoids this problem because it determines the response of a specification to a test by a complete state exploration of the composition [Pavón *et al.*, 1995]. For tests that do not contain **exit**, we have the composition on the left, whereas the composition on the right is for tests that do contain **exit**:

```
SUT[{EventSUT}]                      ( SUT[{EventSUT}]
|[{EventSUT} ∪ {EventTx}]|             |[{EventSUT} ∪ {EventTx}]|
Tx[{EventTx} ∪ {Success}]              Tx[{EventTx} ∪ {Success}]
                                     ) >> Success; stop
```

LOLA analyzes all the test terminations for all possible evolutions (called *test runs*). The successful termination of a test run consists in reaching a state where the termination event (*Success*) is offered. A test run does not terminate if a deadlock or internal livelock[1] is reached. We differentiate three types of tests:

- **May test**: T_x is a may test of *SUT* if it terminates for at least one test run when applied to *SUT* (∃trace in *SUT* |[all gates but *Success*]| T_x that leads to a *Success*). Corresponds to an optional scenario.
- **Must test**: T_x is a must test of *SUT* if it terminates for every test run when applied to *SUT* (*SUT* passes T_x). Corresponds to a mandatory scenario.
- **Reject test**: T_x is a reject test of *SUT* if it does not terminate successfully for any test run when applied to *SUT* (*SUT* failsall T_x). Corresponds to a forbidden scenario.

These types relate to what we call Acceptance/Rejection testing. An acceptance test (a must test in *ACCEPT*) checks that a functionality is present or that an expected result is indeed output. A failure in that case is seen as catastrophic. A rejection test (a reject test in *REJECT*) checks that the *SUT* rejects one or many events after a given set of interactions (trace). A success in that case is catastrophic.

For a given *SUT*, the sets *REJECT* and *ACCEPT* are mutually exclusive (*REJECT* ∩ *ACCEPT* = ∅) and together they constitute the test suite (*REJECT* ∪ *ACCEPT* = *TS*). Reject tests are

1. There is no notion of fairness in this theory. Whenever there is a loop of internal events (τ-loop) which is not under the control of the test process, then the test run has to be truncated. We try to avoid these loops as much as possible in our specifications. Although some theories and simplifications (through weak bisimulation) exist, there are not implemented in LOLA.

useful for implementation (run-time) testing: a *success* in that case always indicates a problem while the *success* of an acceptance test does not mean anything.

May tests will not be used in our approach as the interpretation of the verdict (May pass, composed of successful and unsuccessful traces) usually requires human intervention. Although canonical testers can be reduced to sets of deterministic test cases [Brinksma, 1988], if a SUT happens to be non-deterministic, then an acceptance test could also result in a May pass verdict. In this case, the test case has to be augmented with alternatives so that it results in a Must pass verdict.

Validity Relation

Suppose that *TS* is a test suite generated from informal requirements (*Req*) through a collection of UCMs. *TS* is composed of acceptance test cases (*ACCEPT*) and rejection test cases (*REJECT*), as shown in Figure 28.

FIGURE 28. Partitioning of Acceptance and Rejection Test Groups and Test Cases.

Test Suite *TS*

	ACCEPT			*REJECT*		
Test Groups	TG_{A1}	TG_{A2}	...	TG_{R1}	TG_{R2}	...
Test Cases	$T_{A1.1}$, $T_{A1.2}$, ...	$T_{A2.1}$, $T_{A2.2}$, ...	...	$T_{R1.1}$, $T_{R1.2}$, ...	$T_{R2.1}$, $T_{R2.2}$, ...	...

We can characterize *TS* with respect to the notions defined in section 7.2.2. This allows us to define our *validity* relation val in terms of the successful execution of a test suite.

- *TS* is **sound** $\Leftrightarrow$ (necessary condition)
  ($\forall SUT \in$ **SPECS**, *SUT* val *Req* $\Rightarrow$ *SUT* passes *ACCEPT* $\wedge$ *SUT* failsall *REJECT*).
- *TS* is **exhaustive** $\Leftrightarrow$ (sufficient condition)
  ($\forall SUT \in$ **SPECS**, *SUT* passes *ACCEPT* $\wedge$ *SUT* failsall *REJECT* $\Rightarrow$ *SUT* val *Req*).
- *TS* is **complete** $\Leftrightarrow$ *TS* is **sound** $\wedge$ *TS* is **exhaustive**.

Since our test suite *TS* will be sound but not exhaustive, then we know that (*SUT* passes *ACCEPT* $\wedge$ *SUT* failsall *REJECT*) is a necessary condition for SUT to be valid with respect to the requirements (*SUT* val *Req*). The soundness of *TS* will come from its derivation from individual UCMs interpreted in LOTOS. This also means that there could be invalid implementations that are declared valid by our test suite ($\exists SUT \in$ **SPECS**, *SUT* passes *ACCEPT* $\wedge$ *SUT* failsall *REJECT* $\wedge \neg$(*SUT* val *Req*)). Intuitively, val is a relation weaker than conf (i.e., conf $\Rightarrow$ val).

7.2.5 Testing Cycle

The testing cycle in Figure 7 can be detailed in the following way. Our start point is composed of a specification and a validation test suite. After the successful compilation, indicating that static semantics rules have been satisfied, test cases are applied to the specification (batch testing under LOLA). If unexpected results are found, then the specification and/or the test cases have to be fixed, and the cycle re-executed. When all test cases have resulted in the expected verdict (we say that the functional coverage is achieved), probes are inserted and a new specification is generated according to the strategy to be discussed in Section 7.5. The structural coverage can then be measured by executing the same test suite and by collecting statistical results. If the coverage is not complete, then new test cases can be added (often derived from simulations), or unreachable code can be removed from the specification. This cycle can be executed iteratively each time a specification is modified.

At the end of this process, we get a specification and a validation test suite that are highly consistent and complete. This abstract test suite can then serve for regression testing and as a basis for implementation testing.

Most systems can be seen as *servers*, and the entities that use it as *clients*. This is the paradigm we used to test the GCS. As a first step, we suggest the validation of the SUT with several clients (if they are available) to check realistic scenarios. We believe that interactions between the system and its environment has to be validated first ((GCS || Clients) || TS). Then, we can focus on system testing (GCS || TS), where the SUT is the only entity to be checked. This is a good opportunity to check the robustness of the SUT with test cases involving actions that a client would not normally do. Finally, the verification of the SUT's internal components can be performed (GCS_Component || TS), to increase their robustness and their reusability as self-contained entities. Clients themselves can also be tested individually for robustness (Clients || TS).

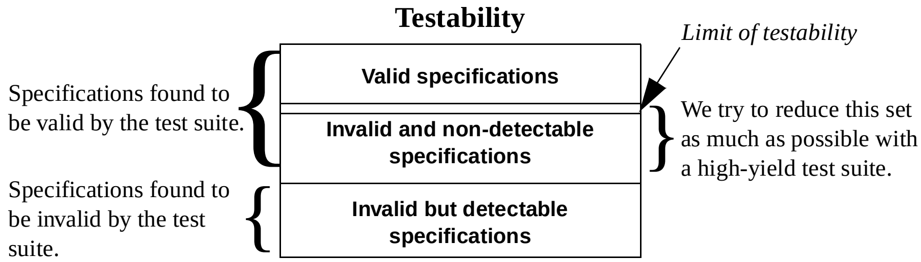
In our example, we focus on system testing without clients, because none was specified for our rather generic GCS.

7.3 Derivation of Validation Test Cases from UCMs

7.3.1 Testability

We can derive validation test cases from a UCM according to many strategies. What we really want however is a validation test suite that will detect invalid specifications under test with the most success and the least cost. Figure 29, adapted from [Drira and Azéma, 1995], illustrates what we intend to achieve. In this diagram, the notion of *detectability* means that a test suite detects the invalidity of a specification with respect to the requirements. *Testability* exposes some limits caused by constraints on the accessibility, observability, and controllability of the SUT. Other limits might also relate to the fact that the behaviour may be infinite. Because LOTOS specifications are highly testable (in opposition to traditional software), those constraints are much weaker with an abstract specification, but they are nonetheless present.

FIGURE 29. Limit of Testability



There is a limit of testability beyond which invalid specifications are not detected by a sound test suite. This set of invalid SUTs has to be reduced as much as possible. The test case derivation and selection strategy relates directly to the size of this set. Of course, a good strategy leads to a good detectability but also to higher costs of derivation and/or execution.

System Testing vs. Unit Testing

Testing processes in most software lifecycles start with unit (or component) testing and end with system and acceptance testing. Our approach suggests quite the opposite at the specification level. We first focus on system testing (the end-to-end functionalities expressed in the requirements) and postpone unit testing. We make the assumption that testing these functionalities and the collaboration between the components (system testing) is more important at the requirements level than testing the internal behaviour of the components (unit testing). This observation has been done for large OO projects in the telecommunication industry [Corriveau, 1996]. We also believe that this global strategy increases the detectability and lowers the limit of testability in the early stages of requirements validation.

7.3.2 Structure of Validation Test Suites from UCMs

We use a structure very similar to the standard one from CTMF (Section 7.2.2). A test suite is a collection of test groups, where each group is linked to one UCM. A group contains test cases that are composed of the following sections:

- **Test purpose**: acceptance or rejection. May tests are not really considered as they require much time investment for the interpretation of the results. The test purpose also contains the specific UCM route that is covered. This ensures traceability from test cases to the requirements.
- **Test preamble**: test events needed to bring the SUT in a state that satisfies the UCM's preconditions. They can come from another UCM.
- **Test body**: the selected UCM route, with data values.
- **Test verification** (optional): functionality-based events (possibly a route from another UCM) used to check that the postcondition is reached. The verification is not based on FSM techniques, such as unique input/output, because we do not have a FSM for the requirements.

- **Test postamble** (optional): in an ordered list of test cases, brings the SUT back to an acceptable initial state.

Postambles are not used in our test cases as the execution of each test case starts with the initial state of the LOTOS specification. Moreover, the selection of data values is eased by the fact that we only have constraints and conditions associated to one path to satisfy, starting from a well-known initial state. However, if the test suite is meant to be refined as an *implementation* test suite, then postambles become most relevant because the cost of resetting a real machine might be too high. In this case, we need to give much attention to three points:

- **Ordering**: the order in which test groups and test cases within test groups are executed becomes relevant. An ordering strategy is needed for reducing the cost of executing the test suite.
- **Postambles**: they become necessary for bringing the SUT back to an acceptable initial state, where the preamble of the next test case can satisfy its precondition. We do not suggest the merging of postambles and preambles as we would rather not having test cases too coupled together, in case we want to reorder them differently.
- **Data values**: their selection becomes critical as they have more constraints to satisfy among many test cases. They need to be carefully chosen and be consistent within a test group.

Test cases contain test events only. We do not see the need to have more general test steps, as suggested in CTMF. In our validation test suite, verification sections will be included where they seem necessary and feasible. For instance, a test body that passes does not prove that modifications to databases have been correctly done. These databases have to be interrogated to check this fact.

Tests Groups

For each individual UCM, we suggest the creation of at least two test groups: one for acceptance test cases and another one for rejection test cases (see Figure 28). Groups can be described as a collection of individual test processes, one for each test case. This is the safest way to represent groups. However, to increase performances or to reduce the number of processes, test cases may be regrouped under one process, as illustrated by `Group_of_Tests`:

```
process Group_of_Tests [gatelist, Success] : noexit :=
    i; TestSequence1... (* First test case *)
    []
    i; TestSequence2... (* Second test case *)
    []
    ...
    []
    i; TestSequenceN... (* Nth test case *)
endproc (* Group_of_Tests *)
```

Internal events need to be inserted to ensure the execution of all test cases.[1] If we are sure that the first action of each test case will be executed, then these internal actions can be removed. This is the way we represented test groups in our test suite. Since the

CREATEGROUP operation that starts each test case is never refused, addition internal events are not necessary.

For complex test cases and complex specifications, we recommend the creation of one process for each test case. The state space resulting from the testing composition will be more manageable by the tools.

Another option that would merge acceptance and rejection test cases is also possible. Most of the time, a rejection test case differs from its corresponding acceptance test case only by the last action:

- Acceptance test case: `a?x:int; b; c!x [x gt 3]; success;` **`stop`**
- Rejection test case: `a?x:int; b; c!x [x le 3]; reject;` **`stop`**
- New acceptance test case: `a?x:int; b; (c!x [x gt 3]; success;` **`stop`**
  `[] c!x [x le 3]; reject;` **`stop`**`)`

7.3.3 Testing Strategies

In most LOTOS techniques, test selection is done either informally, or formally through a LTS (sometimes using a canonical tester). Be believe we have access to a better, more suitable, and more organized representation of the requirements than LTSs, namely the Use Case Maps. We suggest a selection approach based on the *causal paths* of a UCM, which represent the most relevant, interesting, and critical functionalities of the system.

Our approach is very similar to the selection of test cases for white-box testing. However, instead of using the structure of a program, we are using the paths of several UCMs. The UCMs being at a level of abstraction between the requirements and the specification, the assumption is that we minimize the number of test cases that we generate, while at the same time maximizing the coverage of the informal requirements. Moreover, such test cases are more likely to be correct w.r.t. the requirements than test cases derived manually from those same informal requirements.

As explained in the previous section, we use a UCM as the basis for the generation of an acceptance test group and a rejection test group. Each UCM route, where parameters (if any) are instantiated, is a candidate for becoming a black-box test case. By following these routes, we aim to produce test cases that are as sequential and deterministic as possible.

A UCM may enable many possible routes, all of which might not be necessary for testing purposes. In our context, the traditional question "*how much testing is enough testing?*" boils down to "*what are the routes to be tested?*". There is no unique answer to this question. Depending on how critical, important, or relevant are the routes, a UCM may be tested more or less thoroughly. The important thing here is to mention and document the *strategy* used to derive test cases from a specific UCM. Different route selection strategies will lead to different test groups that would achieve a higher functional coverage, usually at a higher cost.

---

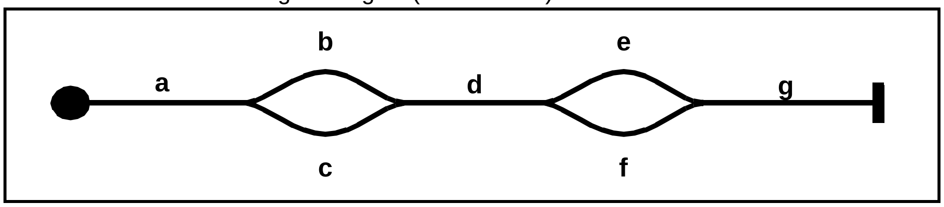
1. A test case that is not even able to perform its first action will not be detected as a failure if all the other test cases are successful. We must use the choice ([]) operator with caution in test cases.

A strategy represents a *test hypothesis* that aims to reduce the test effort while achieving an equivalent coverage according to some fault domain. In a sense, the UCMs that were used as a design artifacts now help us define our *fault models* [Petrenko, 1998].

Strategies for Alternatives

A path might express several routes between a start point and an end point. Figure 30 shows a UCM used to illustrate strategies for alternatives. Seven responsibilities (**a** to **g**) identify the different path segments in this UCM.

FIGURE 30. Reference UCM for Testing Strategies (Alternatives)



These four strategies are inspired from control flow testing, a white-box testing technique. They are ordered according to their coverage (and cost), from the least effective to the most effective. The routes (delimited by angle brackets) enumerated after each strategy do not represent the only solution; other possibilities may exist.

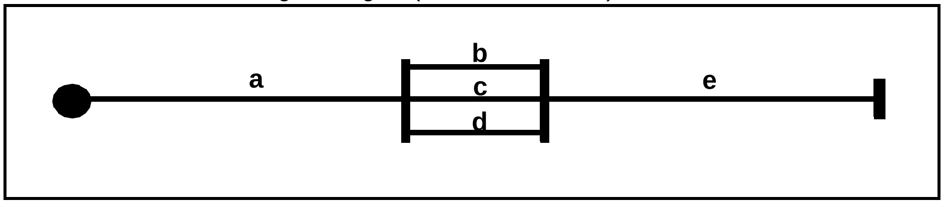
- Alternative — All results : {**<a, b, d, e, g>**}
  Each end point (result) is covered. There could be many results in one scenario cluster (UCM).
- Alternative — All paths : {**<a, b, d, f, g>, <a, c, d, e, g>**}
  All decisions (e.g., true or false) of conditions are exercised. Also referred to as “all branches”.
- Alternative — All path combinations :
  {**<a, b, d, e, g>, <a, b, d, f, g>, <a, c, d, e, g>, <a, c, d, f, g>**}
  All combinations of conditions (e.g., TT, TF, FT, FF) are explored. Also referred to as “all branch combinations” or “all decision combinations”.
- Alternative — All combinations of sub-conditions within a complex condition.
  A complex condition includes more than one operator. The following LOTOS guard is an example : `[(c1 AND c2) OR (c3 AND c4)]`. Since `c1`, `c2`, `c3`, and `c4` can be either True or False, there is a total of $2^4 = 16$ combinations for this alternative only[1]. This strategy can further be applied to multiple conditions when necessary.

Strategies for Concurrent Paths

We suggest three strategies for path segments that run concurrently (see Figure 31). Again, they are ordered according to an increasing level of complexity.

1. If sub-conditions are not independent, some combinations might be impossible to satisfy. For instance, in [x < 3 OR x > 5], we cannot find a solution so that x<3 is true and x>5 is true.

FIGURE 31. Reference UCM for Testing Strategies (Concurrent Paths)



- Concurrent — One combination : {**<a, b, c, d, e>**}
  The simplest one, when concurrency is not critical.
- Concurrent — Some combinations : {**<a, b, c, d, e>, <a, d, b, c, e>**}
  When concurrency is important, but when the total number of possible combinations is too high. The more routes there are, the higher becomes the level of confidence.
- Concurrent — All combinations : {**<a, b, c, d, e>, <a, b, d, c, e>, <a, c, b, d, e>, <a, c, d, b, e>, <a, d, b, c, e>, <a, d, c, b, e>**}
  Only when concurrency is critical and when the number of combinations is practical.

Although we covered combinations of multiple OR-Forks in the strategies for alternatives, we do not see any need for considering combinations of multiple AND-Forks. Each AND-Fork can be treated independently from the other ones in the UCM.

Note that all combinations can be tested by a LOTOS process that makes use of the parallel operator (|||). For instance, the following test case would be sufficient for testing all six combinations enumerated in the third option:

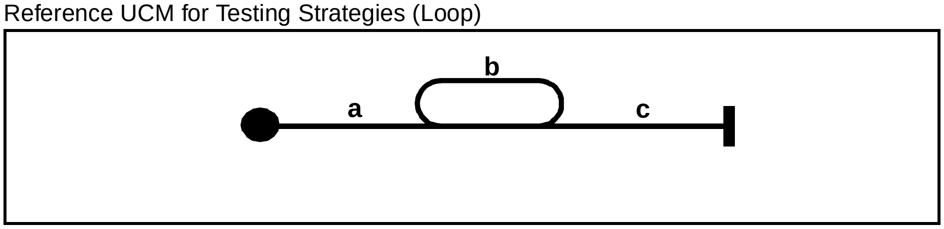
```
a; (b; exit ||| c; exit ||| d; exit) >> e; ...
```

For pragmatic considerations related to the performance of tools such as LOLA, we discourage the use of the parallel operator in test processes.

Strategies for Loops

Loops in a UCM (interpreted as recursion in LOTOS) also require special attention. With the help of Figure 32, we present three general strategies, which could be adapted in the case where a minimum number of iterations is required.

FIGURE 32. Reference UCM for Testing Strategies (Loop)



- Loop — 1 iteration : {**<a, b, c>**}
  The minimal set of test cases required to cover all alternatives.

- Loop — At most 2 iterations : {**<a, c>, <a, b, c>, <a, b, b, c>**}
  This is useful when the number of maximum iterations is unknown or very high.
- Loop — 0, 1, n, and $n+1$ iterations :
  {**<a, c>, <a, b, c>, <a, b, b, b, b, c>, <a, b, b, b, b, b, c>**}
  When the maximum number of iterations n is known and practical ($n=4$ in this example), then this tests the boundaries of the loop. Note that the case with $n+1$ iterations becomes a rejection test case.

Strategies for Value Selection

When parameters need to be instantiated, the values must first comply with the selected route (i.e., they need to satisfy the right guards in alternatives). When this is required by the relative importance of the UCM, other strategies related to traditional black-box testing might be considered. Two of the most well-known strategies are equivalence classes and boundary interior analysis.

Completeness and Determinism Issues

Use Case Maps may contain some non-deterministic behaviour due to conditions that overlap. Suppose a two-branches OR-Fork where two conditions *C1* and *C2* (applied on a subset of natural numbers: {0,1,2,3,4,5}) are located. We identify four cases for these conditions. The generation of test cases can be influenced by the lack of completeness and/or determinism.

- Complete and disjoint: *C1* is $X>3$ and *C2* is $X\leq 3$
  The simplest case. Any value will lead to the selection of one specific alternative.
- Complete with conjunction: *C1* is $X>3$ and *C2* is $X<5$
  $X=4$ is a test that will result in a non-deterministic execution.
- Incomplete and disjoint: *C1* is $X>3$ and *C2* is $X<3$
  $X=3$ is a test that will result in a deadlock.
- Incomplete with conjunction: *C1* is $0<X<3$ and *C2* is $1<X<5$
  $X=2$ is a test that will result in a non-deterministic execution.
  $X=5$ is a test that will result in a deadlock.

The second case indicates a UCM where refinement of conditions may be needed in order to get the final implementation. The last two cases are symptomatic of a potentially problematic UCM. Parnas tables can help to assess that a collection of conditions is deterministic and complete.

Strategy for Rejection Test Cases

As a generic strategy for rejection test cases, we suggest the use of the aforementioned strategies for acceptance test cases with a minor change: the resulting event should accept (almost) anything but the expected result. In UCM terms, we are doing path mutation on the resulting event. In the testing world, such fault model is also referred as *off-by-1*. This fault model seems rather simple, but it increases our confidence that the expected result, usually found in a corresponding acceptance test, is the only one the system can offer.

As an illustrative example, we can use a simple vending machine that gives tea when a coin is inserted. We show a LOTOS interpretation of its `UCM`, a possible implementation `SUT` (which might be incorrect), an acceptance test case `Acc`, and a rejection test case `Rej` (where the success event has been renamed `Reject`):

- `UCM := coin; out !tea; UCM`
- `SUT := coin; (out !tea; SUT [] out !coffee; SUT)`
- `Acc := coin; out !tea; Success;` **`stop`**
- `Rej := coin; out ?drink [drink ne tea]; Reject;` **`stop`**

The sound acceptance test suite, solely composed of `Acc`, suggests that `SUT` val `UCM` because the success event is always reached in `SUT |[coin, out]| Acc`. This points out one weakness of LOTOS testing: if the implementation has more (undesirable) behaviour than the specification, then this might be hard to detect. Nevertheless, the fact that our machine could, for the same coin, give coffee instead of tea can be detected by `Rej`. This rejection test case, whose last event accepts anything but tea, unveils the problem in `SUT` because the `Reject` event can be reached in `SUT |[coin, out]| Rej`.

This strategy is useful for checking critical *values* that are expected in a resulting event, not the LOTOS gates themselves. If no parameter is associated to the gate `out`, then we might not be able to generate a rejection test case in this way.

LOTOS-Based Strategy

Our validity relation val, when satisfied, ensures that Equations 2 and 3 are satisfied:

$$\forall T_x,\ T_x \in \mathit{ACCEPT} \Rightarrow \mathit{SUT}\ \underline{\text{passes}}\ T_x \quad \textbf{(EQ 2)}$$

$$\forall T_x,\ T_x \in \mathit{REJECT} \Rightarrow \mathit{SUT}\ \underline{\text{failsall}}\ T_x \quad \textbf{(EQ 3)}$$

Under the assumption of the existence of a LOTOS interpretation for each use case map UCM_y, we can link the concept of a route to the reduction relation (red) on canonical testers (*CT*). We already stated that a UCM route leads to a test purpose (*TP*), and eventually to a test body when values are fixed. The test purpose of an acceptance test case can thus be seen as a reduction of the canonical tester of its corresponding UCM interpreted in LOTOS (Equation 4). For rejection test case, the test purpose must not be a reduction of the canonical tester of any UCM_y (Equation 5).

$$\forall T_x,\ T_x \in \mathit{ACCEPT} \Rightarrow \exists UCM_y,\ TP(T_x)\ \underline{\text{red}}\ CT(UCM_y) \quad \textbf{(EQ 4)}$$

$$\forall T_x,\ T_x \in \mathit{REJECT} \Rightarrow \neg(\exists UCM_y,\ TP(T_x)\ \underline{\text{red}}\ CT(UCM_y)) \quad \textbf{(EQ 5)}$$

Although we have used a test selection and derivation strategy based on UCM inspection and routes in this report, we believe it would be possible to view the problem from another angle and use LOTOS-based derivation techniques on UCMs themselves. The advantages and drawbacks of such an approach are yet to be determined.

7.4 GCS Testing Results

7.4.1 GCS Test Groups

The application of the strategies, enumerated in Section 7.3.3, to test case generation from our twelve UCMs (Section 5) resulted in 24 test groups numbered from 1 to 24 in Table 15. Odd numbered test groups gather 59 acceptance test cases (*ACCEPT*) and even numbered test groups gather 51 rejection test cases (*REJECT*). Each test group has a corresponding LOTOS process in the specification presented in Appendix A.

Test groups 25 and 26 are supplementary processes that illustrate other testing possibilities. The first one is an acceptance test case from the client viewpoint, which checks the refusal of requests to unknown groups by the server. The client needs a timer to detect such problems and then it reacts accordingly. The second process is a complex acceptance test case represented in a more generic format. Using a preamble, it first brings the system from the initial state to a specific state that satisfies a pre-condition. Then, it executes the scenario (test body), and it finally checks the scenario post-condition. With such verifications within a process, there is no real need for a rejection test case. However, a process structured in this way leads to more costly executions with tools.

7.4.2 An Example: Change Administrator

We illustrate the application of strategies to the generation of 5 acceptance and 6 rejection test cases for the Change Administrator functionality (Figure 20). We use the *Alternative — All paths* strategy based on the UCM and on the conditions as structured in Table 14. We derive the following routes, which correspond to the bodies of the test cases in process `Test_17`. As specified in Table 10, activities **a** to **f** are internal to the system, and therefore they are not part of the LOTOS process (we put them between parenthesis in the routes). The following are our test purposes for the acceptance test cases:

- **<CHANGEADMIN, (a), (d), NOADMINGOUP>** : Non-administered group.
- **<CHANGEADMIN, (a), (e), NOTADMIN>** : Administered group, sender is not the admin.
- **<CHANGEADMIN, (a), (f), MEMBERNOTINGROUP>** : Administered group, sender is the admin, but the new admin is not in the group.
- **<CHANGEADMIN, (a), (b), (c), ADMINCHANGED>** : Administered group, sender is the admin, and the new admin is in the group.
- **<CHANGEADMIN, (a), (b), (c), ADMINCHANGED>** : Change from administered group to non-administered, sender is the admin.

The main difference between the last two routes resides in their NewAdmin parameter (some group member in the first case, and *Nobody* in the second case). When necessary, preambles (usually Group Creation) and verification sequences (usually Get Attributes) are added to test bodies in order to bring the system to a correct initial state and to check the final result.

Six rejection test cases are also included in `Test_18`. We have used the same routes as the ones for `Test_17`, but the fourth route was split into two test cases. The first one checks that we cannot get a result other than **ADMINCHANGED**, and the second one also checks that the database has been updated correctly.

The other test groups may have be derived according to different strategies, depending on the structure and the importance of their respective UCM.

7.4.3 Execution Results

Table 15 presents the results of the execution of the test cases on the specification with the *TestExpand* functionality of LOLA. TestExpand analyzes the response of a specification to a given test according to the compositions presented in Section 7.2.4. It has parameters for limiting the depth of the expansion, for maintaining internal events or for removing them according to equivalence rules, for specifying the expected verdict, for generating traces for diagnostics, and for doing partial expansions according to state space and memory usage heuristics.

Because our tests and specification were quite deterministic (without too much interleaving), we had a total of 128 execution traces for our 112 test cases. All of them were successful, i.e., acceptance tests passed and rejection tests failed according to plan. Of course, several defects and discrepancies between the specification and the tests have been found along the way, but they were easily fixed. LOLA allows for the tester to look at execution traces ending with an unexpected result, which eases the diagnostic. Also, LOLA allows for batch testing. All test groups can be executed in sequence, and their individual expected result can be checked. With the help of simple shell scripts and the Unix/DOS `grep` command, any unexpected result of a test can be discovered very quickly. This approach becomes very useful for regression testing. A change to the specification or to the test suite can be checked in a few seconds.

The last two columns of this table will be discussed in Section 7.5.

TABLE 15. Testing Results

Test Group #	Test Scenario (in GCS Specification)	Acceptance / Rejection	# of Test Cases	# Exec. (no probes, no *i*)	# Exec. (with probes, no *i*)	# Exec. (with *i, b, &* probes)
1	*Group Creation (A)*	Accept	3	3	3	3
2	*Group Creation (R)*	Reject	2	2	2	2
3	*Group List (A)*	Accept	3	3	3	3
4	*Group List (R)*	Reject	3	3	3	3
5	*Get Attributes (A)*	Accept	3	3	3	12
6	*Get Attributes (R)*	Reject	3	3	3	9
7	*Registration (A)*	Accept	6	6	6	23
8	*Registration (R)*	Reject	4	4	4	12
9	*Group Members (A)*	Accept	4	4	4	16
10	*Group Members (R)*	Reject	3	3	3	9
11	*Deregistration (A)*	Accept	7	9	9	17
12	*Deregistration (R)*	Reject	5	6	6	13
13	*Multicast (A)*	Accept	6	7	7	19
14	*Multicast (R)*	Reject	6	6	6	18

TABLE 15. Testing Results

15	*Group Deletion (A)*	Accept	6	19	95	29
16	*Group Deletion (R)*	Reject	5	4	4	10
17	*Change Admin (A)*	Accept	5	5	5	20
18	*Change Admin (R)*	Reject	6	6	6	15
19	*Change Moder (A)*	Accept	9	9	9	31
20	*Change Moder (R)*	Reject	7	7	7	21
21	*Change Opened (A)*	Accept	4	4	4	16
22	*Change Opened (R)*	Reject	3	3	3	9
23	*Change Private (A)*	Accept	4	4	4	16
24	*Change Private (R)*	Reject	3	3	3	9
25	*Client Timeout*	Accept	1	1	1	1
26	*Complex Test*	Accept	1	1	1	4
		TOTAL :	**112**	**128**	**204**	**340**

7.5 Coverage

The generation of test cases from scenarios (or by other means) is an *a priori* approach to validation. Such test cases can be derived in parallel with the specification, or even before the specification is written. We assume that the *functional coverage* is achieved, according to selected strategies, when this test suite is executed successfully (*SUT* passes *ACCEPT* $\wedge$ *SUT* failsall *REJECT*).

However, the quality of the test suite can be further enhanced by observing the structure of the specification (branches, events, etc.). The *structural coverage* of a test suite relates to the parts of the specification that have been visited by test cases. When this coverage is unsatisfactory, new test cases an be added *a posteriori*. New types of faults or defects can be uncovered along the way. Under the assumption of a complete functional coverage, we use this structural coverage as a basis for test suite completeness.

This section is concerned with the coverage of a formal specification by a validation test suite. In particular, we focus on the structural coverage of LOTOS specifications using *probe insertion* [Amyot and Logrippo, 1998]. We can instrument a specification and then assess that the structural coverage is achieved when all probes are visited. The goal is to provide hints and assistance in the detection of unreachable portions of the specification and to measure the completeness of the test suite with respect to the *syntactic* structure of the specification, and not necessarily its underlying semantics. We also aim to cast these ideas in an environment where the necessary steps for coverage measurement are automated as much as possible.

7.5.1 Issues in the Use of Probes

Probe insertion is a well-known white-box technique for monitoring software in order to identify portions of code that has not been yet exercised, or to collect information for performance analysis. A program is instrumented with probes (generally counters initially set to 0) without any modification of its functionality. When executed, test cases trigger these probes, and counters are incremented accordingly. Probes that have not been “visited” indicate that part of the code is not reachable with the tests in consideration. One obvious reason may be that the test suite is incomplete.

There are difficult issues related to probe insertion approaches:

1. The first one is concerned with the preservation of the original behaviour. We need to ensure that new instructions do not interfere with the intended functionalities of the original program or specification, otherwise tests that ran successfully on the original behaviour may not do so any longer.
2. Another issue relates to the category of coverage that is possible to achieve by instrumenting a specification with probes. Because probes are implemented as counters of some sort, it is easier to measure the coverage in terms of control flow than in terms of data flow or in terms of faults. Other techniques are more suitable for the two last categories of coverage criteria [Charles, 1997].
3. The optimization of the number of probes represents a third important issue. In order to minimize the performance and behavioural impact of the instrumentation, the number of probes has to be kept to a minimum, and the probes need to be inserted at the most appropriate locations in the specification or in the program.
4. Finally, what we can assess from the data collected during the coverage measurement represents another issue that needs to be addressed. Questions like “Are there test cases that are redundant?”, “Does a high number of visits of a particular probe imply a possible bottleneck?”, and “Why hasn’t this probe been visited by the test suite?” are especially relevant.

These issues will be discussed for sequential program in the next section, and then explored in the context of the GCS case study.

7.5.2 Probes in Sequential Programs

For well-delimited programs, [Probert, 1982] suggests a technique for inserting the minimal number of *statement probes* necessary to cover all branches. Table 16 illustrates this concept with a short Pascal program (a) and an array of counters named `Probe[]`. The counters count the number of times the probe has been reached. Intuitively, (b) shows three statement probes being inserted on the three branches of the program. In (c), we can achieve the same result with two probes only. Using control flow information, we can deduce the number of times that `statement3` is executed by computing `Probe[1]-Probe[2]`. After the execution of the test suite, if `Probe[2]` is equal to

`Probe[1]`, then we know that the ‘else’ branch that includes `statement3` has not been covered.

TABLE 16. Example of Probe Insertion in Pascal

a) Original Pascal code	b) 3 probes inserted in the code	c) Optimal number of probes (2)
statement1; **if** (*condition*) **then** **begin** statement2 **end** **else** **begin** statement3 **end** *{end if}*;	statement1; inc(Probe[1]); **if** (*condition*) **then** **begin** inc(Probe[2]); statement2 **end** **else** **begin** inc(Probe[3]); statement3 **end** *{end if}*;	statement1; inc(Probe[1]); **if** (*condition*) **then** **begin** inc(Probe[2]); statement2 **end** **else** **begin** statement3 **end** *{end if}*;

It has been shown in [Probert, 1982] that the optimal number of statement probes necessary to cover all branches in a well-delimited program is $|E| - |V| + 2$, where $|E|$ and $|V|$ are respectively the number of edges and the number of vertices of the underlying extended delimited Böhm-Jacopini flowgraph of the program.

Regarding the issues enumerated in Section 7.5.1, we can observe the following:

1. If the probe counters are variables that do not already exist in the program, the original functionalities are preserved.
2. The coverage is related to the control flow of the program.
3. There exists a way to reduce the number of statement probes.
4. This technique covers all branches in a well-delimited program.

7.5.3 Probe Insertion in LOTOS

Similarly to probe insertion in sequential Pascal programs, we would like to use LOTOS constructs to instrument a specification at specific locations while preserving its general structure and its externally observational behaviour. Although we allow the execution of test cases to be slowed down by this instrumentation, we do not want it to affect the functionality of the specification or the results of the validation process.

Among all the LOTOS constructs, the most likely candidate for being a probe is an internal event with a unique identifier. Such event would be composed of a hidden gate name that is not part of any original process in the specification (we name it *Probe*), followed by a unique value of some new enumerated abstract data type (*P_0*, *P_1*, *P_2*, *P_3*, ...).

A Simple Insertion Strategy

We define a *basic behaviour expression* (BBE) as being either the inaction **stop**, the successful termination **exit**, or a process instantiation (`P[...]`). In LOTOS, a *behaviour expression* (BE) can be one of the following[1]:

- A BBE (such a BE is also called a *simple BBE*).
- A BE prefixed by a unary operator, such as the action prefix (`;`), a **hide**, a **let**, or a guard (`[predicate]->`).
- Two BEs composed through a binary operator, such as a choice (`[]`), an enable (>>), a disable (`[>`), or one of the parallel composition operators (`|[...]|`, `||`, or `|||`).
- A BE in parentheses.

We also define a *sequence* as a BBE preceded by one or more events (separated by the action prefix operator).

Probes allow us to easily check every event in a behaviour expression, and thus in a whole specification. The simplest strategy consists in adding a probe after each event at the syntactic level. For each event *e* and each behaviour expression *B*, the expression *e; B* is transformed into *e; Probe!P_id; B* where *Probe* is a hidden gate and *P_id* a unique identifier. A probe that is visited guarantees, by the action prefix inference rule, that the prefixed event has been performed. In this case, if all the probes are visited by at least one test case in the validation test suite, then we have achieved a total *event coverage*, i.e., the coverage of all the events in the specification (modulo the value parameters associated to these events).

Table 17 illustrates this strategy on a very simple specification *S1* (a). Essentially, since there are three occurrences of events in the behaviour, three probes, implemented as hidden gates with unique value identifiers, are added to *S1* to form *S2* (b). The validation test suite is composed of two test cases that remained unchanged during the transformation. We will discuss the third specification (c) later.

1. We consider a very common subset of LOTOS where there are no generalized `Par` of `Choice` operators.

TABLE 17. Simple Probe Insertion in LOTOS

a) Original LOTOS specification (*S1*)	b) 3 probes inserted in the specification (*S2*)	c) 2 probes inserted, using the improved strategy (*S3*)
specification S1[a,b,c] : **exit** ... *(* ADTs *)* **behaviour** a; **exit** [] b; c; **stop** **where** **process** Test1 [a]:**exit** := a; **exit** **endproc** *(* Test1 *)* **process** Test2 [...]:**noexit** := b; c; Success; **stop** **endproc** *(* Test2 *)* **endspec** *(* S1 *)*	**specification** S2[a,b,c] : **exit** ... *(* ADTs *)* **behaviour** **hide** Probe **in** (a; Probe!P_1; **exit** [] b; Probe!P_2; c; Probe!P_3; **stop**) **where** ... *(* Test1 and Test2 *)* **endspec** *(* S2 *)*	**specification** S3[a,b,c] : **exit** ... *(* ADTs *)* **behaviour** **hide** Probe **in** (a; Probe!P_1; **exit** [] b; c; Probe!P_2; **stop**) **where** ... *(* Test1 and Test2 *)* **endspec** *(* S3 *)*

Probe insertion is a syntactic transformation that also has an impact on the underlying model. Table 18 presents the LTSs resulting from the expansion of *S1* and *S2*. Although (a) and (b) are not equal, they are observationally equivalent. Therefore, the tests that are accepted and refused by *S1* will be the same as those of *S2*.

TABLE 18. Underlying LTSs

a) Original LOTOS specification(*S1*)	b) 3 probes inserted in the specification (*S2*)	c) Composition of *S2* with two test cases: Test1 & Test2

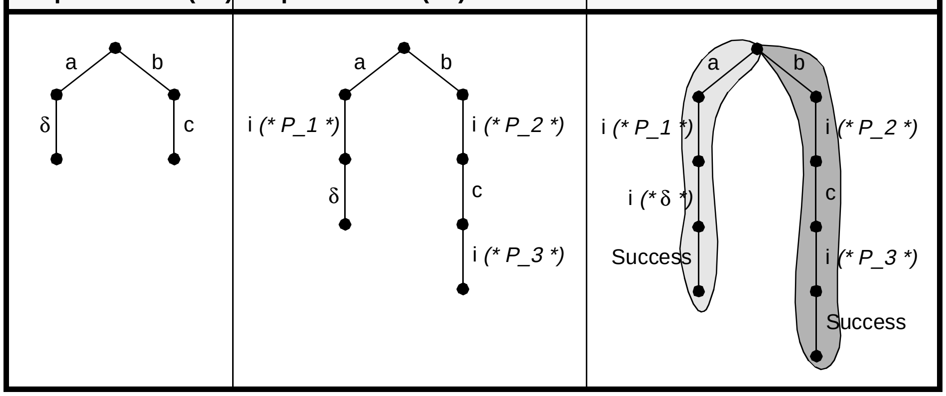


Table 18(c) presents two traces, resulting from the composition of each test process found in Table 17(a) with *S2*, that cover the events and probes of *S2*. `Test1` covers `P_1` in the left branch of (c) while `Test2` covers `P_2` and `P_3` in the right branch. Neither of these tests covers all probes, but together they cover all three probes, and therefore the event coverage is achieved, as expected from the validation test suite.

Going back to the four issues enumerated in Section 7.5.1, we make the following observations:

1. Probes are unique internal events inserted *after* each event (internal or observable) of a sequence. They do not affect the observable behaviour of the specification; this insertion can be summarized by the LOTOS congruence rule:

$$e;\ B\ \approx_c\ \textbf{hide}\ Probe\ \textbf{in}\ (e;\ Probe!P\_id;\ B)\ =\ e;\ \textbf{i};\ B$$

2. The coverage is concerned with the structure of the specification, not with its data flow nor with fault models. We have an *event coverage* where we make abstraction of the values in the events (e.g., we do not distinguish `gate!0` from `gate!succ(0)`).
3. The total number of probes equals the number of occurrences of events in the specification. Reducing the number of probes is the focus of the next section.
4. This strategy covers all events syntactically present in a specification, modulo their value parameters.

Improving the Probe Insertion Strategy

The simple insertion strategy leads to interesting results, but two problems remain. First, the number of probes required is much too high. The composition of a test case and a specification where multiple probes were inserted (and transformed into internal events) can easily result in a state explosion problem. Second, this approach does not cover simple BBEs as such, because they are not prefixed by events. Simple BBEs may represent a sensible portion of the structure of a specification that needs to be covered as well.

In a sequence of actions, the number of probes can be reduced to one probe, which is inserted just before the ending BBE. If such a probe is visited, then by the action prefix inference rule we know that all the events that precede the probe in the sequence were performed. The longer a sequence, the better this optimization becomes. Table 17(c) shows specification *S3* where two probes are necessary instead of three as in *S2*. This *sequence coverage* is equivalent to event coverage, with fewer probes (or the same number in the worst case). However, an event coverage that uses the simple strategy might lead to better diagnostics when a sequence is only partially covered, because we would be able to pinpoint the problematic event in the sequence.

The use of parenthesis in *e; (B)*, where *B* is not a simple BBE, does not require a probe either. The behaviour expression *B* will most certainly contain probes itself, and a visit to any of these probes ensures that event *e* is covered (again, by the prefix inference rule).

For the structural coverage of simple BBEs (without any action prefix), there are some subtle issues that need to be explored. Suppose that `*` is one of the LOTOS binary operators enumerated at the beginning of this section. If we are to prefix the BBE with a probe in the generic patterns `BBE * BE` and `BE * BBE`, we must be careful not to introduce any new non-determinism:

- `BBE` is `stop`: This is the inaction. No probe is required on that side of the binary operator (`*`) simply because there is nothing to cover. This syntactical pattern is useless and should be avoided at the specification level.

- `BBE` is a process instantiation `P[...]`: A probe before the BBE can be safely used except when `*` is the choice operator (`[]`), or when `*` is the disable operator (`[>`) with the BBE on its right. In these cases, a probe would introduce undesirable non-determinism that might cause some test cases to fail partially (may pass verdict). A solution would be to prefix the process instantiation. One way of doing so is to partially expand process `P` with the expansion theorem.
- `BBE` is `exit`: The constraints and solution are the same as for the process instantiation.

Assuming that the definition of process `P` is not a simple BBE, we can further reduce the necessary number of probes for a `BBE` that is `P[...]` when `P` is not instantiated in any other place in the specification, except for recursion in `P` itself (a process call tree such as the one in Figure 26 can help here). In this case, a probe before `P` is not necessary because probes inserted within `P` will ensure that the instantiation of `P` is covered. For example, suppose a process `Q` that instantiates `P`, where `P` is not a BBE nor instantiated in any other process than `P` itself:

```
Q[...] := e1; e2; e3; stop [] P[...]
```

A probe inserted before `P` would make the choice non-deterministic. However, if `P` is not a simple BBE and if it is not instantiated anywhere else, then no probe is required before `P` in this expression. This situation happens often in processes that act as containers for aggregating other processes.

To complete the answers to the four issues given for the simple strategy, the improved probe insertion strategy reduces the number of probes required for event/sequence coverage. It also expands the structural coverage to include event coverage and BBE coverage, except in the cases where a probe would introduce non-determinism. In these cases, some relief strategies (such as prefixing or partial expansion) can be applied.

Tool Support

Though we believe that full automation of probe insertion is possible, we opted for a semi-automated approach in our three examples because we were still experimenting with the technique and some special cases (with problematic BBEs) were not trivial to manage.

A filter was written in LEX, to translate special comments inserted in the original specification (`(*_PROBE_*)`) into internal probes with unique identifiers (e.g., `Probe!P_0;`). Also, a new abstract data type (`ProbeLib`) was added to the specification, to enumerate all the unique identifiers for the probes. Care was taken not to add any new line to the original specification, in order to preserve two-way traceability between the transformed specification and the original one. This tool is called LOT2PROBE.

Since we did not have any full synchronization operator in our specifications, the `Probe` gate was hidden at the topmost level of the specification (the `behaviour` section), and was added to the list of gate parameters of all process definitions and instantiations. In the case where a full synchronization operator is used, probes have to be hidden on each side of this operator, otherwise unexpected deadlocks might occur:

```
B1 || B2  becomes  (hide Probe in B1) || (hide Probe in B2)
```

We used batch testing under LOLA (with the *Command* operation) for the execution of the validation test suite against the transformed specification. Several batch files, written in PERL and LEX, compute probe counts for each test and give a summary of the probes visited by the test suite, with a highlight on probes that were not covered.

7.5.4 Coverage Results

Using the improved probe insertion strategy, we needed only 54 probes in the original GCS specification, even if there were 59 instances of events in the processes, as well as many simple BBEs.

On the specification with probes, the tests resulted in the same verdict as on the original specification, so no new non-determinism had been added. However, by using *TestExpand* without removing internal actions (e.g., the probes) in the expanded LTSs, the statistics showed that 5 of the 54 probes inserted had not been covered by the test suite (see Table 19):

- Two (#23 and #25) were related to a feature that was not part of the requirements or the UCMs, but that was specified in LOTOS anyway (a group is deleted when there is no member left). As such, relevant test cases could not have been derived from the UCMs. We added two test cases (tests 11.1 and 11.2, obtained from a step-by-step simulation of the specification) to cover these probes.
- One probe (#45) was not covered because we had split a UCM path into a choice between two guarded behaviour expressions with different values. It seemed easier to implement in such a way this particular UCM path in LOTOS. However, the test case derived from the UCM covered one alternative only. We simply added another test case (test 19.9) with the right value for the other alternative to be covered.
- The remaining two probes (#0 and #2) were reachable when doing a step-by-step execution of the composition of the relevant tests (1.3 and 3.1) and the specification. However, TestExpand had not output the probe internal events in the resulting LTSs. This is in fact due to an internal problem with TestExpand. No new test case was required as such because we knew we obtained full structural coverage with our validation test suite.

TABLE 19. Probes Coverage

Probe #	Line #	Tests 1 to 26	Reach-able?	Probe #	Line #	Tests 1 to 26	Reach-able?
0	867	0	Y (1.3)	**27**	1290	4	
1	874	146		**28**	1309	11	
2	884	0	Y (3.1)	**29**	1319	5	
3	894	7		**30**	1327	5	
4	968	4		**31**	1342	5	
5	983	267		**32**	1349	4	
6	995	271		**33**	1355	5	
7	1010	305		**34**	1362	5	
8	1029	4		**35**	1368	5	
9	1039	97		**36**	1373	4	
10	1119	39		**37**	1387	2	
11	1127	5		**38**	1396	5	
12	1145	7		**39**	1404	5	
13	1152	4		**40**	1420	2	
14	1158	7		**41**	1425	4	
15	1173	39		**42**	1435	5	
16	1180	6		**43**	1443	5	
17	1191	7		**44**	1463	15	
18	1200	15		**45**	1469	0	Y (19.9)
19	1207	2		**46**	1480	9	
20	1224	24		**47**	1489	2	
21	1229	5		**48**	1502	5	
22	1244	5		**49**	1510	5	
23	1257	0	Y (11.1)	**50**	1532	29	
24	1261	5		**51**	1545	24	
25	1279	0	Y (11.2)	**52**	1549	39	
26	1283	5		**53**	1557	3	

7.5.5 Interpreting Coverage Results

Missing Probes

We have shown instances of problems associated to probes that are not visited by a validation or conformance test suite. They usually fall into one of the following categories:

- Incorrect specification. In particular, there could be unreachable code caused by processes that cannot synchronize or by guards that cannot be satisfied.

- Incorrect test case. This is usually detected before probes are inserted, during the verification of the functional coverage.
- Incomplete test suite. Caused by an untested part (an event or a BBE) of the specification (e.g., a feature of the specification that is not part of the original requirements.)
- For our scenario-based approach, there could be some discrepancy between a UCM and the specification caused by ADTs, guards, and the choice ([]) operator.

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

LOLA's *FreeExpand* could be used to expand the whole specification in order to check that all probes are in the underlying LTS. This would ensure that no part of the code is unreachable. However, for most real-size specifications, this approach is not likely to work because of the state explosion problem. Using on-the-fly model checking, the verification of an appropriate property, which would state that a particular probe can be eventually reached, seems a more practical solution.

Goal-oriented execution [Haj-Hussein *et al.*, 1993], a technique based on LOTOS' static semantics, could be a promising approach to the determination of the reachability of a unique probe. However, this technique would first have to be extended in order to allow specific internal events (the probes) to be used as goals.

Compositional Coverage of the Structure

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 would reduce the size of the resulting LTSs to a minimum, and thus help avoiding the state explosion problem. This approach was applied to another similar project, where there were too many probes to handle them all at the same time [Amyot *et al.*, 1998]

Specification Styles

Two equivalent specifications written using different styles might lead to different coverages for the same test suite. The way a LOTOS specification is structured usually reflects more than its underlying LTS model. For instance, in a resource-oriented style, the structure can be interpreted as the architecture of the system to be implemented. In a constraint-oriented style, processes impose local or end-to-end constraints on the system behaviour. The impact of the specification style on the structural coverage approach is a research direction that is yet to be explored.

7.5.6 Reducing the Test Suite Using Coverage Statistics

Redundant tests add cost but not rigor. If a probe is covered by one test only, then the presence of this test case is obviously required in the test suite. However, two tests that cover exactly the same probes might indicate some redundancy. Nevertheless, this redundancy is mainly structural, and perhaps not functional (according to the strategy used in the test plan). Therefore, both tests might still be required in the test suite to achieve the functional coverage.

Consider the following three equations:

$$(VP(T_x) - \bigcup_{y,\, x \neq y} VP(T_y) \neq \emptyset) \Rightarrow T_x \text{ is necessary} \quad \textbf{(EQ 6)}$$

$$(VP(T_x) \subseteq \bigcup_{y,\, x \neq y} VP(T_y)) \Rightarrow T_x \text{ is useless} \quad \textbf{(EQ 7)}$$

$$(\exists y, x \neq y \wedge VP(T_x) \subset VP(T_y)) \Rightarrow T_x \text{ is useless} \quad \textbf{(EQ 8)}$$

Equation 6 is obviously true because T_x is a test case that covers one or more probes that the other test cases do not, so it becomes necessary to keep T_x in our test suite.

Equation 7 says that if the probes visited by T_x are all already covered by the other test cases, then T_x becomes useless and we do not have to keep it. In other words, if no new information is provided by this test case, we do not get a good return on our investment. Although this sounds reasonable at first sight, we believe this is *not always true*. For instance, if we derived four test cases from the UCM in Figure 30 following the “all path combinations” strategy, we can see that the probes visited by each of these test cases would be visited by the union of the other three. Nonetheless, removing any of these test cases would violate the testing strategy, especially for a critical scenario that addresses critical parts of the system.

If a test case T_y covers more probes than another one (T_x), does this mean that T_x is useless, as suggested by Equation 8? This is probably a good candidate for removal, but we again have to be cautious. Probes are inserted in the behaviour part of the specification only. Two test cases could cover the same probes with different data values in order to check, for instance, recursion, conditions combinations within guards, or boundary analysis on conditions. T_x covering fewer probes than T_y does not mean T_x must be removed. We have to consider the whole testing strategy and test purposes.

Although such metrics can be used to provide hints about test cases that are good candidates for being removed, one has to be cautious not to act on this sole piece of information. We believe that minimization of test suites based on probe coverage cannot be automated, although it can provides useful hints as to which test cases are good candidates for removal, and which should remain in the test suite.

7.5.7 A Note on LOLA’s Heuristic Expansion

Several lengthy test cases led to state explosion problems when we required to keep internal actions in the LTSs. For these tests, we had to use the heuristic expansion option of *TestExpand* instead of the default exhaustive expansion. In all the instances where we used this option, the probe coverage was the same as for the exhaustive expansion, but there was an important reduction (about 99%) of the size of resulting LTS and of the time required for the expansion (see Table 20 and Table 21 for an example with test #8).

TABLE 20. Comparison Between Heuristic and Exhaustive Expansions for Test #8

Category	Heuristic Expansion	Exhaustive Expansion
Number of executions	17	28768
Number of states	443	135097
Number of transitions	459	164064
Size of expansion file	9.4 KB	22.5 MB
Memory needed for expansion	< 10 MB	39.5 MB
Time needed for expansion (Pentium, 150MHz)	3 seconds	3 minutes
Number of probes visited (see Table 21)	92	75644

TABLE 21. Probes Visited by Test #8 using Heuristic and Exhaustive Expansions.

Probe #	Heuristic Expansion	Exhaustive Expansion
1	5	3713
3	1	22
4	19	8925
5	19	9678
6	20	21108
8	11	5483
12	2	114

Probe #	Heuristic Expansion	Exhaustive Expansion
15	2	57
17	6	19242
19	2	15
21	2	1332
23	2	5940
24	1	15
TOTAL:	**92**	**75644**

This option, used to derive some results in the last column of Table 15, allowed for the generation of coverage statistics for the whole test suite in less than a minute. Such a time period seems short enough for this technique to be used in a heavily iterative design process.

7.6 Synthesis, Testing, and Model Checking

In [Probert and Saleh, 1991], the authors presents two categories of design approaches for communication protocols that can be generalized to most reactive and distributed systems:

- **Analytic approach**: the designer iteratively produces versions of the system by defining messages and their effect on the entities. This often results in incomplete and erroneous designs that require analysis, verification (testing), and correction of errors.

- **Synthetic approach**: a partially specified design is constructed or completed such that the interactions between its entities proceed without manifesting any error and (ideally) provide the set of specified services. No verification is needed as the correctness is insured by construction.

Three types of properties must be guaranteed, independently of the approach chosen:

- **Safety properties**: something bad never happens (no deadlocks, no livelocks, absence of unspecified reception errors, etc.).
- **Liveness properties**: something good will eventually happen, i.e. the system performs its intended functions.
- **Responsiveness properties**: the system respects the response time requirements (timeliness) and it has the possibility of recovering in the case of transient failures (fault-tolerance).

These properties can be usually guaranteed by verifying the absence of syntactic and semantic design errors:

- **Syntactical or Logical design errors**: relate to the logical structure of the exchange of messages among the entities. These errors are usually independent of the service or functionality: deadlocks, unspecified receptions, instabilities, livelocks, overspecifications, and channel overflow. The absence of such design errors guarantees the safety properties.
- **Semantic design errors**: relate to the functionalities to be provided by the system. Such an error is manifested by the abnormal functioning of the system and its inability to meet its intended purpose.

The synthesis approaches for protocols covered in [Probert and Saleh, 1991] do not appear satisfactory for our purpose as they focus on messages too soon. Right now, we see our approach as being somewhere between the analytic and synthetic approaches. We first use several guiding rules for the synthesis of the specification (Section 6.1), and use an analytical approach, namely validation testing (Section 7.3). Our main focus is on semantic errors as they relate to the system purpose while syntactical errors are more or less independent of the functionalities to be provided.

As explained in Section 7.1, the three types of properties, and especially safety and liveness properties, are difficult to check using functionality-based testing. Model checking represents the usual alternative, but our experience suggests that such a technique is hardly applicable, even for on-the-fly model checking, without simplifying the specification or considering only part of it. Several attempts at solving this complexity issue can be found in the literature. [Chehaibar *et al.*, 1996] is worth mentioning as the authors express these types of properties as graphs (FSM) that are checked, through branch equivalence and bisimulation equivalence, against the specification. However, even this approach needs a graph (FSM) representation of the specification, which cannot be generated from ours without drastic simplification in the ADTs and in the number of processes that can be instantiated.

We therefore suggest *testing by counter-example* as being a pragmatic compromise between model checking, equivalence checking, and functionality-based testing. This approach does not result in a proof of the presence/absence of a property, but clever test cases may again improve our level of confidence in the specification. For instance, one could define a new global safety property for the GCS, which is not part of the initial requirements: *no group can be non-administered and private* (Equation 9).

$$\forall group \in GroupList,\ \neg(\ \neg IsAdministered(group) \wedge IsPrivate(group)\ ) \qquad \textbf{(EQ 9)}$$

One could generate tests (from UCMs or by “intuition”) that would aim at defeating this property. In our current specification, such a counter-example could be found because this property was not used at the requirements analysis and design phases. One obvious test case consists in trying to create a non-administered and private group and then check its attributes:

```
mgcs_ch !tomgcs !user1 !creategroup !group1 !encode(mail, chan1,
              nonadministered,nobody,opened,private,nonmoderated,nobody);
mgcs_ch !frommgcs !user1 !groupcreated !group1;
gcs_ch !togcs !user1 !group1 !getattributes !nomsg;
gcs_ch !fromgcs !user1 !attributesare(encode(mail,chan1,nonadministered,
              nobody,opened,private,nonmoderated,nobody)) ! group1;
```

This trace indeed possible in our specification, hence showing that this new property does *not* hold. Several processes and UCMs (CREATEGROUP, CHANGEADMIN, CHANGEPRIVATTR) would need to be slightly modified for this property to hold.

7.7 Chapter Summary

Many concepts related to the validation of specifications in terms of UCMs and LOTOS were defined and illustrated in this chapter.

In Section 7.1, we identified functionality-based testing as a pragmatic approach to the validation of LOTOS specifications derived from operational scenarios. Test cases can be generated from the same scenarios used to synthesize the specification, because they are simple to understand and sufficiently close to the informal requirements.

We reviewed the conformance testing methodology and the LOTOS testing theory in Section 7.2, and adapted them to our scenario context. We formalized a validity relation in terms of sound acceptance/rejection test cases derived from UCMs. A test cycle that includes test case generation from UCMs and functional/structural coverage measurement was also presented.

In Section 7.3, we decided to focus on system testing, instead of unit/component testing, in order to lower the limit of testability in the early stages of requirements validation. We adapted the generic CTMF hierarchical structure to test suites generated from UCMs. In particular, we suggested the creation of one acceptance test group and one rejection test group for each UCM. As for the generation of test cases in those test groups, we presented several strategies based on the coverage of UCM critical routes (including alternatives, concurrent paths, loops, and value selection). The need for rejection test cases was illustrated, and an off-by-1 strategy, where we mutate the expected result event of acceptance test cases, was defined accordingly. We also defined a relation between test purposes and UCMs in terms of reductions of corresponding canonical testers.

In Section 7.4, We illustrated the test cases generation for the Change Administrator service. The results of our complete test suite on the GCS specification were then presented. We assume that the functional coverage is achieved, according to the chosen strategy, when the test suite is executed successfully (*SUT* passes *ACCEPT* $\wedge$ *SUT* failsall *REJECT*), although this does not guarantee that the specification under test is valid w.r.t. the requirements (*SUT* val *Req*). We discussed the use of testing tools (e.g., LOLA) on many occasions.

Section 7.5 motivated the need for measuring the completeness of a validation test suite in terms of the coverage of the specification structure, and for detecting unreachable code in a specification. We discussed issues related to probe insertion approaches in general and illustrated these concepts with an existing technique for sequential programs. We then proposed an approach based on the insertion of probes for measuring the structural coverage of the behaviour section of LOTOS specifications. This pragmatic and semi-automated technique can help detect incomplete test suites, inconsistencies between a specification and its test suite, and unreachable parts of the specification, with respect to the requirements in consideration. We suggested a strategy for the insertion of probes in a specification to measure the coverage of all the instances of events. We improved this strategy by reducing the number of probes required for a structural coverage that includes sequence/event coverage and basic behaviour expressions (BBE) coverage. Concrete tool support for this approach was also addressed. Using our validation process, which includes structural coverage, we presented the results for the GCS specification, which motivated the need for additional test cases in our test suite. A discussion followed on the interpretation of coverage results in general, and on the use of LOLA's heuristic expansion.

We presented, in Section 7.6, where our global approach for the generation and validation of specifications with scenarios fits in the traditional synthesis and analytic approaches. We also proposed testing by counter-example as being a pragmatic way, in our context, for validating new requirements expressed as global properties.

Chapter 8 Discussion and Future Work

In this chapter, we discuss several issues encountered during the generation of UCMs, the synthesis of the LOTOS model, and the validation of the GCS. Some lessons and advice are presented, as well as several items for future work.

8.1 Comments on the Iterative Approach

8.1.1 Generating a Collection of Scenarios

Generating useful and descriptive scenarios (whether they are causal or not) from informal requirements requires experience.

- In general, the problem is understood at the same time as people work on it. This motivates the need for iterations in the approach.
- There is no need to specify what we cannot validate in a given iteration. For instance, we haven't specified the whole GPRS system, nor asynchronous communication between components. This could be done in a next macro-iteration if necessary.
- Creating UCMs when there is only one person involved in their description in one thing. Having many people involved in their generation, especially when they are distributed over several locations, is a challenge. There is a crucial need for some sort of data dictionary, glossary, or ontology management. Such a tool would allow designers to share and synchronize their definitions of responsibilities and other named items, and therefore to improve consistency. It could also be used to check early completeness of scenarios.

8.1.2 Synthesizing the Model

- In traditional object-oriented approaches (e.g., OOA+OOD), systems are often presented from three viewpoints: structure, scenarios, and components internal. Our approach aims to guide the synthesis of the internal using the two first models.
- While synthesizing a model (in LOTOS or any other language), one should try to sort scenarios. Priority should be given to the most important ones, i.e. the ones with the most impact on the specification, and to the ones that are the least likely to change. This aims to reduce integration risks during the synthesis.
- UCMs can be superimposed on other alternative structures. In our GCS example, we introduced many such structures according to criteria related to distribution and concurrency. Any of these could become another candidate for synthesizing a new specification. Decisions taken during a first synthesis should be reused as much as possible for subsequent synthesis where the paths are mapped to other structures. These decisions needs to be documented along the way.
- The structure of the model (a LOTOS specification in our case) and the selected specification style will also influence the ease with which new scenarios can be introduced. For instance, in the design of the GCS, seven UCMs were defined in the first version (new functionalities were added later). However, the impact of adding five new UCMs was minimal; the structure was defined in such a way that those new functionalities resulted into five new sub-processes in `GCS`, one for each new scenario. The other processes remained basically untouched. New test cases and probes were added to achieve the required functional and structural coverage.

- With some precautions, addition of new scenarios that are alternatives to previous ones (they are branching at the UCM level) could become simple to incorporate in the synthesized model.
- When adding a new scenario, the nature and the importance of the impact always depend on this new scenario. If the latter is a minor variation of a previously defined UCM, then the overall impact will be insignificant. If it is a radically new scenario, completely different from the other ones, then the model might need to be completely rewritten. Many existing test cases, or at least test purposes, would however still be useful and valid.

8.1.3 Validating the Model

- The introduction of a new scenario triggers some traceability issues. For instance, one could measure the impact on the model, but also on the validation test suite for regression testing.
- In order to lower the impact of such a modification on the validation, test cases could be related together more closely for regression testing. In an iterative and incremental design process, one wants to minimize the number of tests to be re-executed. With an executable specification, this seems to be less of a problem, because testing at this level is far less expensive. Nonetheless, testing is never free and therefore we should aim to minimize its use.

8.1.4 Improving the Rigorous Scenario-Based Approach

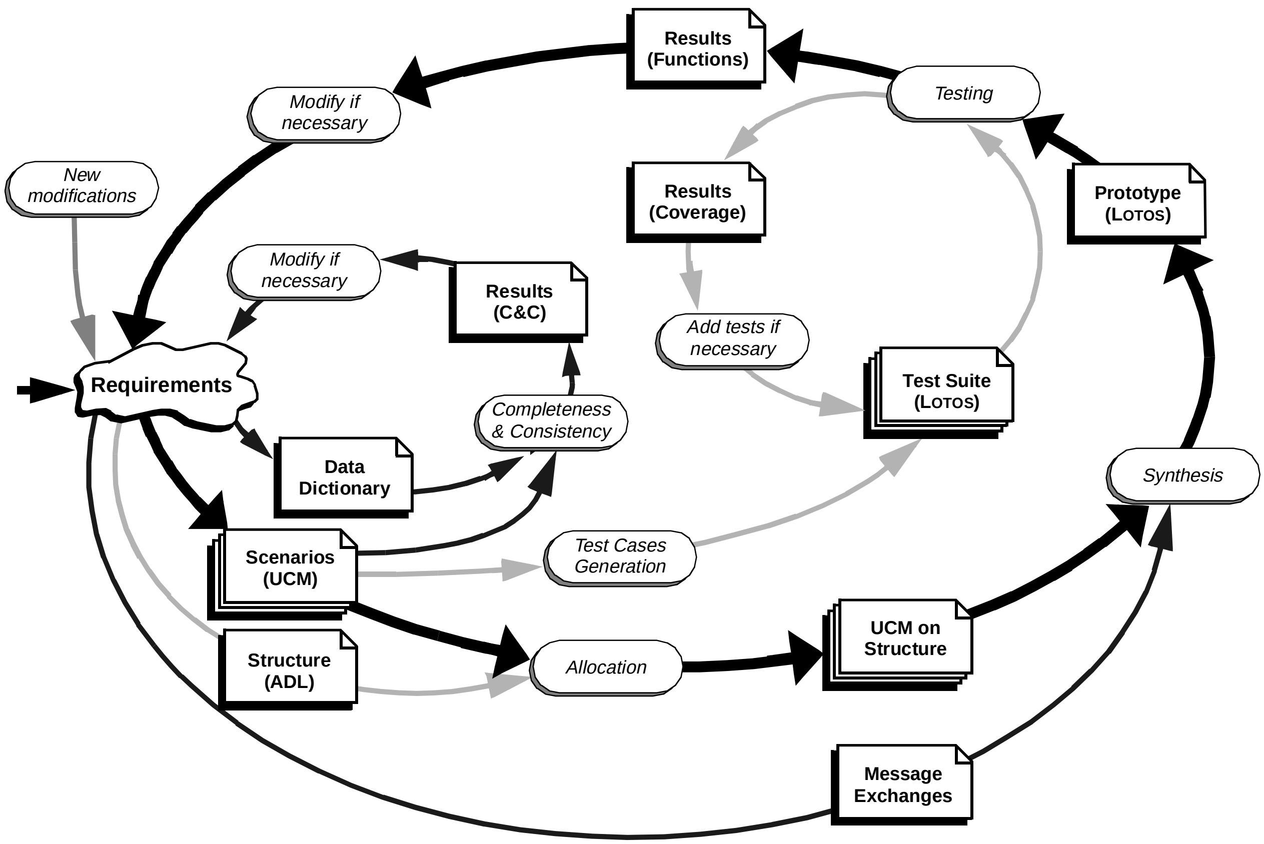
As suggested in Section 8.1.1, the approach would gain from the support of some data dictionary. Figure 33 presents an additional cycle in our approach where such a data dictionary, built at the same time as the scenarios, is used for early verification of consistency and completeness properties in the scenarios.

Some simple properties could be checked at first:

- All the terms in the data dictionary need to be mentioned in at least one scenario (completeness property).
- All terms used in the scenario need to be part of the data dictionary (completeness property).
- Two terms in the data dictionary that express the same concept in the same context should be reduced to one term (consistency property).
- Terms should not be used outside their context (consistency property). For instance, the name of a responsibility should not be used as a triggering event of a waiting place.
- Etc.

Figure 33 also introduces another improvement on the approach. The synthesis of the model could take into consideration message exchange patterns for causality relationships between two responsibilities located in different components. These patterns would become a traceable and documented mean of refining such relationships.

FIGURE 33. Suggested Scenario-Based Approach



The structure definition could gain from the knowledge found in many Architecture Description Languages (ADL, e.g., ACME [Garlan *et al.*, 1997]) and other structure notations (ROOM, UML, SDL, etc.). They could provide for different mechanisms for describing components and their relationships, possibly more in line with the culture of a specific design group. In essence, the structure notation could be adapted to the needs and knowledge of the design team.

The synthesis and test cases generation also need to cover other important elements of the UCM notation such as stubs and plug-ins.

8.2 Thoughts on the Testing Cycle

8.2.1 Automation Issues

The automation of the testing cycle in our approach is yet another important issue that needs to be addressed. Part of it, as illustrated in the GCS example, is already automated (batch testing, results collecting, and structural coverage measurement). Steps currently done manually include the test cases generation and the insertion of probes. The second step is likely to be automatable (although perhaps not optimally). The first step is unlikely to be fully automated because of issues such as:

- Test case selection according to different strategies related to the functional coverage.
- The derivation of test purposes from UCMs.
- Transformation of a test purpose into a LOTOS test process, while at the same time considering the LOTOS structure, the message exchange patterns, and the necessary preambles and verification steps.

Strategies for test selection and derivation are affected by the required functional coverage. In this document, we assume that this functional coverage is achieved when the tests derived according to the selection strategy (for the most interesting paths) have passed as intended. Although useful, this definition does not give factual numbers about the coverage in terms of events or traces allowed by the UCMs. The following section attempts to present candidate solutions to that problem.

8.2.2 Metrics for Functional Coverage Based on UCM Paths

To evaluate the functional coverage of a collection of test cases, we can base our definition on the number of possible traces that the system may exhibit. Intuitively, if all those traces are covered by the validation test suite, then a complete functional coverage would be achieved.

Suppose that test cases are sequential, unique, and linear in nature (i.e., traces). We can evaluate the functional coverage (FC) by computing:

FC = number of test cases / number of system traces. **(EQ 10)**

The problem with Equation 10 is that the number of traces that the system may exhibit can be infinite due to data, unbounded loops, and recursion. Since the number of test cases is usually finite, FC would almost always reduce to 0, and hence this metric would not be very useful.

Under the same assumptions, it is possible to get a more interesting definition by considering *symbolic* traces instead of simple traces (Equation 11). In a nutshell, symbolic traces abstract from the data they carry, although conditions cumulated along a trace have to be satisfiable. Explicit values are substituted by symbolic values, therefore whole (and possibly infinite) sets of values can be reduced to several equivalence classes. Although the number of symbolic system traces may also be infinite, it is much less likely to be infinite than the number of system traces.

FC = number of test cases / number of *symbolic* system traces. **(EQ 11)**

The problem then reduces to evaluating the number of symbolic traces, and this can be complex, especially at the system level. We suggest the use of individual UCMs to evaluate this number. In order to further simplify the problem, we propose a local definition of FC. Each test group, originally derived from one UCM, would have its own FC related to the symbolic traces derivable from this same UCM (Equation 12). Note that the concept of symbolic trace is equivalent to the concept of *route* as defined in Section 2.2.

$$\forall UCM_n \in UCMs,\ FC_n = |TG_n| / |Routes(UCM_n)| \qquad \textbf{(EQ 12)}$$

$Routes(UCM_n)$ is the set of routes allowed by UCM_n. Note that $|Routes(UCM_n)|$ is an **upper bound** on the number of symbolic traces. Some routes might not be feasible due to unsatisfiable path conditions, which are abstracted from in symbolic traces.

With this definition in mind, the focus is now the computation of $|Routes(UCM_n)|$. We would rather approximate directly the cardinality of $Routes(UCM_n)$ than explicitly develop this set of routes. Metrics should be obtained as quickly and effortlessly as possible.

By looking at the way path segments are interconnected in a UCM, we can get an approximation of the number of routes it contains (or at least an upper bound). Intuitively, a sequential segment allows only one route, whatever the number of events contained in that segment. An OR-Fork will introduce as many routes as there are alternative output segments. With an AND-Fork, however, the number of routes depends on the number of events.

The following guidelines aims to compute the number of routes in a UCM that has a single triggering event (this is the case for most of the GCS scenarios):

- Start from a resulting event and go backward towards the initial triggering event. Keep track of the number of events/actions along the path, but do not consider conditions as actions.
- If an OR-Fork is encountered, then compute the number of routes and events of all its other outgoing paths. The total number of routes is the sum of the number of each path routes. Store the number of events for each path as they could be used later.
- If an OR-Join is encountered, then add the current number of routes and events to those of each incoming path.
- If an AND-Fork is encountered, then compute the number of routes and events (for each trace) of all its other outgoing paths. The total number of routes is the result of *InterComb* formula (Equation 13) applied to these events and paths.
- If an AND-Fork followed by AND-Join are encountered, then use *InterComb* formula for the concurrent segments, and then multiply by the number of traces after the AND-Join. This could become much more complex when followed by another AND-Fork or OR-Fork.

InterComb (interleaved combination) is a function that computes the number of routes resulting from the number of events (n_i) on each k **concurrent** paths.

$$InterComb(n_1, n_2, ..., n_k) = \frac{(n_1 + n_2 + ... + n_k)!}{n_1! \times n_2! \times ... \times n_k!} = \frac{\left(\sum_{i=1}^{k} n_i\right)!}{\prod_{i=1}^{k} n_i!} \qquad \textbf{(EQ 13)}$$

This function is used as is when each path has only one route. When concurrent paths have more than one route, resulting from an AND-Fork or an OR-Fork, then *InterComb* has to be applied to each combination of routes (with their respective number of events). The resulting number of routes is the sum of all these partial results.

Although incomplete, this definition of functional coverage provides an opportunity for the definition of testing metrics based on test cases and on probes to measure the improvement of our testing process (especially the test case generation). The generalization of these guidelines into a algorithm for system symbolic traces would involve considering the full UCM notation (including stubs, plug-ins, and aborts), and relationships between multiple UCMs.

8.2.3 Metrics for Structural Coverage Based on Probes: Number of Visits

The problem of using the number of visits per probe for performance measures is outside of the scope of our work. LOTOS specifications focus on functionalities, not on performance. The specification style will influence the number of visits, and so will the options used in *TestExpand* and *FreeExpand* under LOLA. A very low number of visits for a probe might indicate the need for more thorough testing, while a high number of visits might indicate a potential contention of bottleneck. Again, this is a research direction that needs to be explored.

8.3 Chapter Summary

We presented different issues related to the generation of scenarios and to the synthesis of models from structures and scenarios. We discussed the impact of a scenario modification or addition on the synthesis and validation (including regression testing).

We also discussed the inclusion of a data dictionary and message exchange patterns in an improved approach. Several plausible consistency and completeness properties, applied to a collection of UCMs, illustrated the usefulness of the data dictionary.

The automation of the testing cycle remains an important issue in our approach. We presented what could be automated in it, and we discussed a metric for computing the functional coverage of a validation test group w.r.t. a UCM. We suggested the use of symbolic traces and UCM routes as a potential mechanism for defining the functional coverage, and showed what is involved in their computation. This represents the first step in what could become a better way of measuring the effectiveness of a test suite and of improving the generation of validation test cases.

Chapter 9 Conclusions

9.1 General Conclusions

In this report, we presented many ideas related to a development process that goes from scenario-based requirements to high-level specifications, and eventually to a component-oriented form suitable for implementation (although we did not explicitly discuss any concrete implementations).

The approach, illustrated with the Group Communication Server example, focuses on the preliminary steps of scenario-based requirements engineering and on the generation and validation of a first prototype design. It aims to rapidly produce prototypes from causal scenarios and structures. The generation of a validated test suite, used to check the model with respect to the informal requirements, is another goal of this iterative and incremental design process. We chose Use Case Maps as the notation for representing high-level causal scenarios and LOTOS as the modeling language for specifying and validating the prototypes.

Several particularities of this approach, introduced in Section 2.4, are as follow:

- **Separation of the functionalities from the underlying structure**: this helps focusing on the causal sequences of responsibilities to be performed by the system, independently from message exchange sequences between components. It also furthers the reuse of scenarios and their mapping on different alternative structures (see Chapter 4).
- **Fast prototyping through synthesis**: the scenarios guide the synthesis of a component-oriented specification. Chapter 6 provided general guiding rules for this synthesis and gave a general overview of how we generated the GCS. In Chapter 8, we discussed the impact of the modification and addition of scenarios on the synthesis and validation cycles.
- **Test cases generation and validation**: scenarios provides the mean for deriving test purposes that lead to high-yield test cases. In Chapter 7, we identified functionality-based testing as a pragmatic approach to the validation of LOTOS specifications derived from operational scenarios. We formalized a validity relation in terms of sound acceptance/rejection test cases derived from UCMs according to several strategies based on the coverage of critical routes. We also adapted the generic CTMF hierarchical structure to test suites generated from UCMs. Section 7.5 motivated the need for measuring the completeness of a validation test suite in terms of the coverage of the specification structure, and for detecting unreachable code in a specification. Hence, we proposed an approach based on the insertion of probes for measuring the structural coverage of the behaviour section of LOTOS specifications. Finally, we shared many practical experiences with tools used for the validation of the GCS specification.
- **Design documentation**: the scenarios and the specification provide useful and traceable documents for further steps in the design, implementation, and maintenance processes.

This experiment allowed us to gain knowledge and experience in the precise definition, through scenarios, of a non-trivial application. As a result, we obtained a documented and validated design for the GCS, and we enumerated along the way many issues related to component-oriented specifications generated from a system-centric view.

9.2 Future Work Items

Many items for future work are distributed among the previous chapters. The following list recalls the most important ones:

- Definition of a new rigorous approach that would include a data dictionary for early consistency and completeness verification of scenarios, and message exchange patterns for refining causal relationships.
- Definition of a set of usable message exchange patterns for distributed systems.
- XML-based description language for UCM (with structures, stubs, and plug-ins).
- Improvement of the synthesis of component behaviour from scenarios, with stubs and plug-ins.
- Improvement of the theory for test cases generation from scenarios, with stubs and plug-ins.
- Definition of a metric for evaluating the functional coverage of a test suite in terms of the routes allowed by UCMs.
- Automatic probe insertion algorithm and tool for structural coverage.
- Completion of other examples such as phases I and II of the forthcoming Wireless Intelligent Network standard, and feature interaction detection and resolution in agent systems.

The general goal is therefore to provide solid grounds in order to automate this approach as much as possible and pave the way to rapid prototyping and validation of distributed systems.

Chapter 10 References

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

Amyot, D. (1994b) *LOTOS Generation from Timethread Maps: A Language and a Tool*. CSI 5900 project report, Dept. of Computer Science, University of Ottawa, Canada. http://www.csi.uottawa.ca/~damyot/ucm/tmdl/tmdl.pdf

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, 1996. http://www.csi.uottawa.ca/~damyot/phd/forte95/forte95.pdf

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

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

Amyot, D., and Logrippo, L. (1998) "Structural Coverage of LOTOS Specifications Through Probe Insertion". http://www.csi.uottawa.ca/~damyot/forte98/forte98.pdf

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

Bordeleau, F. and Amyot, D. (1993) "LOTOS Interpretation of Timethreads: A Method and a Case Study". TR-SCE-93-34, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada

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.

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.

Brinksma, E., Tretmans, J., and Verhaard, L. (1991) "A Framework for Test Selection". In: B. Jonsson, J. Parrow, and B. Pehrson (Eds.), *Protocol Specification, Testing and Verification XI*, Elsevier Science Publishers B.V.

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

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

Buhr, R.J.A. (1997b) "Scenario-Path Signatures as Architectural Entities for Complex Systems". In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Ottawa, Canada, January 1998.
http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.pdf

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: *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998.
http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam98.pdf.

Cavalli, A., Kim, S., and Maigron, P. (1993) "Improving Conformance Testing for LOTOS". In: R.L. Tenney, P.D. Amer and M.Ü. Uyar (Eds), *FORTE VI, 6th International Conference on Formal Description Techniques*, North-Holland, 367-381, October 1993.

Charles, Olivier. (1997) *Application des hypothèses de test à une définition de la couverture*. Ph.D. Thesis, Université Henri Poincaré — Nancy 1, Nancy, France, October 1997.

Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., and Zulian, F. (1996) "Specifictaion and Verification of the PowerScale™ Bus Arbitration Protocol: An industrial Experiment with LOTOS". In: R. Gotzhein and J. Bredereke (Eds), *Proceedings of FORTE/PSTV'96*, Kaiserslautern, Germany, 435-450, October 1996.

Cheung, T. Y. and Ren, S. (1992) *Operational Coverage and Selective Test Sequence Generation for LOTOS Specification*. TR-92-07, Dept. of Computer Science, University of Ottawa, Canada, January 1992.

Corriveau, J.-P. (1996) "Retraçage et processus de développement pour des projets industriels orientés objet". In: APIIQ, *L'Expertise informatique*, Vol. 3, No. 1, 18-22, Summer 1996. http://www.crim.ca/APIIQ/expertise/

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.

Drira, K. and Azéma, P. (1995) "Les graphes de refus pour la vérification de conformité et l'analyse de testabilité des protocoles de communication". In: *Electronic Journal on Networks and Distributed Processing*, No. 1, April 1995, 27-47.

ETSI (1996), Digital Cellular Telecommunications system (Phase 2+); *General Packet Radio Service (GPRS); Service Description Stage 1 (GEM 02.60), Version 2.0.0* (November 1996).

Fraser, M.D. and Vaishnavi, V.K. (1997) "A Formal Specifications Maturity Model". In: *Communications of the ACM*, Vol. 40, No. 12, December 1997, 95-103.

Garlan, D., Monroe, R. T., and Wile, D. (1997) "ACME: an Architectural Interchange Language". In: *Proceedings of ICSE'97, 19th IEEE International Conference on Software Engineering*.

Ghribi, B. and Logrippo, L. (1993) "A Validation Environment for LOTOS". In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII*, North-Holland.

Grégoire, J-C. and Ferguson, M.J. (1997) "Neglected Topics of Feature Interactions: Mechanisms, Architectures, Requirements". In: P. Dini *et al.*, *Feature Interactions in Telecommunications and Distributed Systems IV*, IOS Press, 3-12.

Hackathorn, R. (1997) "Data Delivery When You Want It". In: *BYTE*, vol. 22, no. 6, June 1997.

Haj-Hussein, M., Logrippo, L. and Sincennes, J. (1993) "Goal Oriented Execution for LOTOS". In: M. Diaz and R. Groz (Eds), *Formal Description Techniques, V*, North-Holland, 311-327.

Hennessy, M. (1988) *Algebraic Theory of Processes*. Foundations of Computing, MIT Press, Cambridge, USA.

Herbsleb, J., Zubrow, D., Goldenson, D., Hayes, W., and Paulk, M. (1997) "Software Quality and the Capability Maturity Model". In: *Communications of the ACM*, Vol. 40, No. 6, June 1997, 31-40.

ISO (1988), Information Processing Systems, Open Systems Interconnection, "LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", IS 8807.

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

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

ITU (1996), "Recommendation Z. 120: Message Sequence Chart (MSC)". ITU (formerly CCITT), Geneva.

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

Johnson, P. M., (1998) “Reengineering inspection”. In: *Communications of the ACM*, Vol. 41, No. 2, February 1998, 49-52.

Ladkin, P.B., and Leue, S. (1995) “Four issues concerning the semantics of Message Flow Graphs”. In: D. Hogrefe and S. Leue (Eds), *Formal Description Techniques, VII*, Chapman & Hall.

Leduc, G. (1991) “Conformance relation, associated equivalence, and minimum canonical tester in LOTOS”. In: B. Jonsson, J. Parrow, and B. Pehrson (Eds.), *Protocol Specification, Testing and Verification XI*, Elsevier Science Publishers B.V., 249-264.

Moreira, A. M. D., and Clark, R. G. (1996) “Adding rigour to object-oriented analysis”. In: *Software Engineering Journal*, IEE, September 1996, 270-280.

Myers, G. J. (1979) *The Art of Software Testing*. Wiley-Interscience, New-York.

Nursimulu, K. and Probert, R. (1995) “Cause-Effect Graphing Analysis and Validation Requirements”. Department of Computer Science, University of Ottawa, Canada, TR-95-14 (June).

Parnas, D. L., Madey, J., and Iglewski, M. (1994) “Precise Documentation of Well-Structured Programs”. In: *IEEE Transactions on Software Engineering*, Volume 20 Number 12 (December), 948-976.

Paulk, M., Curtis, B., Chrissis, M.B., and Weber, C. (1993) *Software Capacity Maturity Model, Version 1.1*. Software Engineering Institute, CMU/SEI-93-TR-25 (February).

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

Petrenko, A. (1998) “Modeling Faults in Object State Machines”. In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Ottawa, Canada, January 1998.

Pressman, R. S. (1987) *Software Engineering — A Practitioner’s Approach*. McGraw-Hill, USA.

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

Probert, R.L., and Saleh, J. (1991) “Synthesis of communications protocols: survey and assessment”, *IEEE Transactions on Computers*, Vol. 40, No. 4, April 1991, 468-476

Richardson, D.J., O’Malley, O, and Tottle, C. (1989) “Approaches to Specification-Based Testing”. In: R.A. kemmerer (Ed), *Software Engineering Notes*, Vol. 14, No. 8, 86-96, December 1989.

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

Quemada, J., (1997). *Working Draft on Enhancements to LOTOS*. ISO/IEC JTC1/SC21/WG1, "Enhancement to LOTOS" (1.21.20.2.3), January 1997.

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

Sarashina, K., Ando, T., Ohta, M., Tokita, Y., and Takahashi, K. (1993) "An Integrated Specification Support System for Communication Software Design Based on Stepwise Refinement and Graphical Representation". In: R. L. Tenney, P. D. Amer, and M. Ü. Uyar (Eds), *Formal Description Techniques, VI*, IFIP/North-Holland.

Schmidt, D.C. (1994) "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software". In: *Sun User Group conference*, 1994.

Schmidt, D.C. (1996) "Object-Oriented Design Patterns for Concurrent, Parallel, and Distributed Systems". Invited talk, University of Carleton, march 1996.

Selic, B., Gullekson, G., and Ward, P.T. (1994) *Real-Time Object-Oriented Modeling*, Wiley & Sons.

Somé, S., Dssouli, R., and Vaucher, J. (1996) "Un cadre pour l'ingénierie des exigences avec des scénarios". In: Bennani, A., Dssouli, R., Benkiran, A., and Rafiq, O. (Eds), *CFIP 96, Ingénierie des protocoles*, ENSIAS, Rabat, Maroc.

Somé, S. (1997) *Dérivation de Spécifications à partir de Scénarios d'interaction*. Ph.D. Thesis, Département d'IRO, Université de Montréal, Canada.

Tretmans, J. (1989) "Test Case Derivation from LOTOS Specifications". In S. T. Vuong (Ed), *Formal Description Techniques II*. North-Holland, 345-360, December 1989.

Tuok, R. (1996) *Modeling and Derivation of Scenarios for a Mobile Telephony System in LOTOS*. M.Sc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

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

Appendix A LOTOS Specification of the GCS

This LOTOS specification of the GCS contains the Abstract Data Types we needed (lines 51 to 812), the behaviour (processes) corresponding to the functionalities and selected component structure (lines 814 to 1562), and the test suite (lines 1565 to 3160).

Probes insterted in the specification are comments in ***bold italic***.

```
1      (****************************************************************************)
2      (*                                                                          *)
3      (*      Group Communication Service, VERSION 2.02                           *)
4      (*      =========================================                           *)
5      (*                                                                          *)
6      (*      Daniel Amyot and Jacques Sincennes,                                 *)
7      (*      University of Ottawa,                                               *)
8      (*      Ver 2.02: June 9 - June 13, 1997                                    *)
9      (*      Ver 2.01: May 16 - May 18, 1997                                     *)
10     (*      Ver 2.00: April 18, 1997                                            *)
11     (*      Ver 1.15: January 24, 1997                                          *)
12     (*      Ver 1.14: January 14, 1997                                          *)
13     (*      Ver 1.13: August 15, 1996                                           *)
14     (*      Ver 1.12: July 8, 1996                                              *)
15     (*      Ver 1.0 : Spring 1996                                               *)
16     (*                                                                          *)
17     (*      Purpose:  Multicast messages to the members of a group.             *)
18     (*      =======                                                             *)
19     (*                                                                          *)
20     (*      Channels:                                                           *)
21     (*      ========                                                            *)
22     (*      mgcs_ch:  "Manager of Group Communication Servers" Channel (1)      *)
23     (*      gcs_ch:   Group Communication Server Channels (1 per group)         *)
24     (*      out_ch:   Output channels to distribute messages to group members   *)
25     (*                (1 per group)                                             *)
26     (*                                                                          *)
27     (*      Groups:                                                             *)
28     (*      ======                                                              *)
29     (*      - administered: Administrator alone creates and destroys group.     *)
30     (*                      Administrator can also change admin or moderator.   *)
31     (*                                                                          *)
32     (*      - moderated:    Moderator is the only one allowed to multicast.     *)
33     (*                      All other messages are forwarded to the moderator,  *)
34     (*                      by the group server, for approval.                  *)
35     (*                                                                          *)
36     (*      - public :      Anyone can register to a group (e.g. mailing lists) *)
37     (*      OR                                                                  *)
38     (*      - private:      Admin must register all new members. A user must be *)
39     (*                      member to see list of group members (e.g. telephone *)
40     (*                      conferences)                                        *)
41     (*                                                                          *)
42     (*      - opened:       Anyone can multicast to the group                   *)
43     (*                      (e.g. mailing lists)                                *)
44     (*      OR                                                                  *)
45     (*      - closed:       A user must be member of the group to multicast     *)
46     (*                      (e.g. Internet Relay Chat)                          *)
47     (*                                                                          *)
48     (****************************************************************************)
49
```

```
specification Group_Communication_Service[mgcs_ch, gcs_ch, out_ch]:noexit

library
    Boolean, NaturalNumber, HexDigit
endlib

(*=============================================*)
(*             IS8807 ADT definitions          *)
(*=============================================*)

(* Types FBoolean, Element, and Set contain corrections *)
(* to the library from the International Standard 8870  *)

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

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

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

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

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

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

type    Set is Element, Boolean, NaturalNumber
sorts   Set
opns    {}                                  :               -> Set
        Insert, Remove                      : Element, Set  -> Set
        _IsIn_, _NotIn_                     : Element, Set  -> Bool
        _Union_, _Ints_, _Minus_            : Set, Set      -> Set
        _eq_, _ne_, _Includes_, _IsSubsetOf_    : Set, Set      -> Bool
        Card                                : Set           -> Nat

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

            x IsIn Insert(y,s) =>
```

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

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

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

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

        ofsort  Bool

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

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

        s IsSubsetOf t          = t Includes s          ;

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

        s ne t                  = not(s eq t)           ;

        ofsort  Nat

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


(*=============================================*)
(*              GCS ADT definitions            *)
(*=============================================*)

(* A group can be Administered or not, Moderated or not *)
(* Private or Public, Opened or Closed. We need four attributes. *)
type    GroupType is NaturalNumber
sorts   Attribute
opns
        Administered, NonAdministered,
        Moderated, NonModerated,
        Private, Public,
        Opened, Closed  :                       -> Attribute
        N               : Attribute             -> Nat
        _eq_, _ne_      : Attribute, Attribute  -> Bool
```

```
eqns
        forall  at1, at2: Attribute

        ofsort Nat
        N(Administered)     = 0;
        N(NonAdministered)  = Succ(N(Administered));
        N(Moderated)        = Succ(N(NonAdministered));
        N(NonModerated)     = Succ(N(Moderated));
        N(Private)          = Succ(N(NonModerated));
        N(Public)           = Succ(N(Private));
        N(Opened)           = Succ(N(Public));
        N(Closed)           = Succ(N(Opened));

        ofsort Bool
        at1 eq at2 = N(at1) eq N(at2);
        at1 ne at2 = N(at1) ne N(at2);

endtype     (* GroupType *)

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

(* Generic ADT for identifiers and enumerations. *)
type    EnumType is NaturalNumber
sorts   Enum
opns
        (* Keep Elem0 for special purposes when necessary *)
        Elem0, Elem1, Elem2, Elem3, Elem4, Elem5 : -> Enum
        (* Mapping on Naturals, for comparison with other elements *)
        N            : Enum       -> Nat
        _eq_, _ne_   : Enum, Enum -> Bool
eqns
        forall  enum1, enum2: Enum

        ofsort Nat
        N(Elem0)  = 0;
        N(Elem1)  = Succ(N(Elem0));
        N(Elem2)  = Succ(N(Elem1));
        N(Elem3)  = Succ(N(Elem2));
        N(Elem4)  = Succ(N(Elem3));

        ofsort Bool
        enum1 eq enum2 = N(enum1) eq N(enum2);
        enum1 ne enum2 = N(enum1) ne N(enum2);
endtype (* EnumType *)

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

(* A group member (or any user) is identified by an member identifier. *)
type        MIDType is EnumType renamedby
sortnames   MID for Enum
opnnames
            Nobody for Elem0 (* Special MID reserved for admin/moder modif. *)
            User1  for Elem1
            User2  for Elem2
            User3  for Elem3
            User4  for Elem4
endtype (* MIDType *)

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

```

(* List of MIDs.  Used to answer "Members" requests *)
(* Implemented as a set. *)
(* We avoid the problem with ISLA's renaming in actualization *)
type            MIDListType0 is Set
actualizedby    MIDType using
sortnames
            MID for Element
            Bool  for FBool
endtype     (* MIDListType0 *)


type        MIDListType is MIDListType0 renamedby
sortnames
            MIDList for Set
opnames
            Empty   for {}      (* Empty list of members *)
endtype     (* MIDListType *)

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

(* We define several Channel Identifiers (within a channel type) *)
(* Used to describe the specifics of the multicasting type (requestor's *)
(* host, IP, socket) according to group requirements *)
type        CIDType is EnumType renamedby
sortnames   CID for Enum
opnames
            Chan5  for Elem0 (* No special CID reserved. *)
            Chan1  for Elem1
            Chan2  for Elem2
            Chan3  for Elem3
            Chan4  for Elem4
endtype (* CIDType *)

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

(* Set of pairs (member, channelID), to be registered in a group *)
type    MemberType is MIDType, CIDType
sorts   MBR
opns
        _._          : MID, CID      -> MBR
        MID          : MBR           -> MID
        CID          : MBR           -> CID
        _eq_, _ne_   : MBR, MBR      -> Bool
eqns
        forall  mid1, mid2: MID,
                cid1, cid2: CID,
                mbr1, mbr2: MBR

        ofsort MID
        MID(mid1.cid1)              = mid1;

        ofsort CID
        CID(mid1.cid1)              = cid1;

        ofsort Bool
        (mid1.cid1) eq (mid2.cid2)  = (mid1 eq mid2) and (cid1 eq cid2);

        mbr1 ne mbr2                = not(mbr1 eq mbr2);
```

```
endtype (* MemberType *)

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

(* Type for a list of member-channel pairs (implemented as a set) *)
(* We avoid the problem with ISLA's renaming in actualization *)
type            MemberListType0 is Set
actualizedby    MemberType using
sortnames
            MBR for Element
            Bool  for FBool
endtype     (* MemberType0 *)


type        MemberListType1 is MemberListType0 renamedby
sortnames
            MemberList for Set
opnnames
            NoMBR   for {}      (* Empty list of members *)
endtype     (* MemberListType1 *)


(* Additional operations needed to act like a list *)
type    MemberListType is MemberListType1, MIDListType
opns
        ErrorNoTop      :                   -> MBR
        ErrorNoTail     :                   -> MemberList
        Top             : MemberList        -> MBR
        Tail            : MemberList        -> MemberList

        (* The following functions act on the MID only. *)
        _IsIn_          : MID, MemberList   -> Bool
        _NotIn_         : MID, MemberList   -> Bool
        RemoveMBR       : MID, MemberList   -> MemberList
        MembersOnly     : MemberList        -> MIDList
eqns
        forall  member  : MBR,
                m       : MemberList,
                id1, id2: MID,
                chnl    : CID

        ofsort MBR
        Top (NoMBR)                 = ErrorNoTop;  (* Should not happen *)
        Top (Insert (member, m))    = member;

        ofsort MemberList
        Tail (NoMBR)                = ErrorNoTail; (* Should not happen *)
        Tail (Insert (member, m))   = m;

        RemoveMBR (id1, NoMBR)                  = NoMBR;
        RemoveMBR (id1, Insert(id1.chnl, m))    = m;
            id1 ne id2 = (true of Bool) =>
        RemoveMBR (id1, Insert(id2.chnl, m))    = Insert(id2.chnl,
                                                         RemoveMBR(id1, m));

        ofsort Bool
        id1 IsIn m                          = id1 IsIn MembersOnly(m);
        id1 NotIn m                         = not(id1 IsIn m);
```

```
        ofsort MIDList
        MembersOnly(NoMBR)                   = Empty;
        MembersOnly(Insert(id1.chnl, m))    = Insert(id1, MembersOnly(m));

endtype (* MemberListType *)

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

(* Several GCS IDentifiers *)
type        GIDType is EnumType renamedby
sortnames   GID for Enum
opnnames
            Group5  for Elem0 (* No special gID reserved. *)
            Group1  for Elem1
            Group2  for Elem2
            Group3  for Elem3
            Group4  for Elem4
endtype (* GIDType *)

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

(* List of channel multicasting types available *)
(* Others could be added. Those are only examples. *)
(* They do not really influence any behaviour here. *)
type        ChanType is EnumType renamedby
sortnames   Chan for Enum
opnnames
            Mail   for Elem0
            Socket for Elem1
            Text   for Elem2
            Audio  for Elem3
            Video  for Elem4
endtype (* ChanType *)

(* Indicate direction over bi-directional channels *)
type    DirectionType is
sorts   Direction
opns
        FromMGCS, ToMGCS,
        FromGCS, ToGCS      :   -> Direction
endtype (* DirectionType *)


(***************************************************************************)
(* A group contains characteristics and a list of members.                 *)
(* Characteristics are tuples (records):                                   *)
(*                  (ChannelType,       (of Chan)                          *)
(*                   Channel Identifier (of CID)                           *)
(*                   AdminAttribute,    (of Attribute)                     *)
(*                   Administrator,     (of MID)                           *)
(*                   OpenedAttribute,   (of Attribute)                     *)
(*                   PrivateAttribute,  (of Attribute)                     *)
(*                   ModerAttribute,    (of Attribute)                     *)
(*                   Moderator          (of MID) )                         *)
(*                                                                         *)
(* A "rationale" or "purpose" field could also be added (not shown here).  *)
(*                                                                         *)
(* Used in messages and in GCS databases.                                  *)
(***************************************************************************)
```

```

type    GCSinfoRecordType is GIDType, MemberListType, ChanType, GroupType
sorts   Msg
opns
        Encode      : Chan, CID, Attribute, MID, Attribute,
                      Attribute, Attribute, MID     -> Msg
        _eq_, _ne_  : Msg, Msg                      -> Bool
eqns
        forall  gcs1, gcs2  : Msg

        ofsort Bool
        gcs1 eq gcs2        = true;   (* Artificial eq needed by ISLA. *)
        gcs1 ne gcs2        = not(gcs1 eq gcs2); (* DO NOT USE!!! *)
endtype (* GCSinfoRecordType *)

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

(* List of Groups for Master GCS, implemented as a set. *)
(* We avoid the problem with ISLA's renaming in actualization *)
type            GroupListType0 is Set
actualizedby    GIDType using
sortnames
            GID for Element
            Bool  for FBool
endtype     (* GroupListType0 *)


type        GroupListType is GroupListType0 renamedby
sortnames
            GroupList for Set
opnames
            NoGCS     for {}    (* Empty list of GCS *)
endtype     (* GroupListType *)

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

(* Request messages to be sent to the server *)
type        RequestType is HexDigit renamedby
sortnames   Request for HexDigit
opnames
            CREATEGROUP     for 1
            GETATTRIBUTES   for 2
            DELETEGROUP     for 3
            REGISTER        for 4
            DEREGISTER      for 5
            MEMBERS         for 6
            GROUPS          for 7
            MULTICAST       for 8
            CHANGEADMIN     for 9
            CHANGEOPENATTR  for A
            CHANGEPRIVATTR  for B
            CHANGEMODER     for C
endtype     (* RequestType *)

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

(* Resulting acknowledgement and error messages from the server *)
type    AckErrorType is NaturalNumber, MIDListType, GCSinfoRecordType,GroupListType
sorts   AckError
```

```
opns
        (* Acknowledgements *)
        GROUPCREATED,
        GROUPDELETED,
        REGISTERED,
        DEREGISTERED,
        MESSAGESENT,
        ADMINCHANGED,
        MODERCHANGED,
        OPENATTRCHANGED,
        PRIVATTRCHANGED,
        SENTTOMODERATOR,
        GROUPWASDELETED,(* Multicast when a group is deleted *)

        (* Errors *)
        GROUPEXISTS,
        GROUPDOESNOTEXIST,
        MEMBERNOTINGROUP,
        NOTADMIN,
        NOTMODER,
        UNKNOWNREQUEST,
        NOADMINGROUP,
        NOMODERGROUP      : -> AckError

        (* Additional operation to encode lists of groups. *)
        GROUPSARE    : GroupList -> AckError
        (* Additional operation to encode lists of members. *)
        MEMBERSARE   : MIDList   -> AckError
        (* Additional operation to encode attributes. *)
        ATTRIBUTESARE: Msg       -> AckError

        (* Mapping on Naturals, for comparison with other AckError *)
        N            : AckError           -> Nat
        _eq_, _ne_   : AckError, AckError -> Bool
eqns
        forall  a1, a2: AckError, m: MIDList, g:GroupList, gi:Msg

        ofsort Nat
        N(GROUPCREATED)     = 0;
        N(GROUPDELETED)     = Succ(N(GROUPCREATED));
        N(REGISTERED)       = Succ(N(GROUPDELETED));
        N(DEREGISTERED)     = Succ(N(REGISTERED));
        N(MESSAGESENT)      = Succ(N(DEREGISTERED));
        N(ADMINCHANGED)     = Succ(N(MESSAGESENT));
        N(MODERCHANGED)     = Succ(N(ADMINCHANGED));
        N(OPENATTRCHANGED)  = Succ(N(MODERCHANGED));
        N(PRIVATTRCHANGED)  = Succ(N(OPENATTRCHANGED));
        N(SENTTOMODERATOR)  = Succ(N(PRIVATTRCHANGED));
        N(GROUPWASDELETED)  = Succ(N(SENTTOMODERATOR));
        N(GROUPEXISTS)      = Succ(N(GROUPWASDELETED));
        N(GROUPDOESNOTEXIST)= Succ(N(GROUPEXISTS));
        N(MEMBERNOTINGROUP) = Succ(N(GROUPDOESNOTEXIST));
        N(NOTADMIN)         = Succ(N(MEMBERNOTINGROUP));
        N(NOTMODER)         = Succ(N(NOTADMIN));
        N(UNKNOWNREQUEST)   = Succ(N(NOTMODER));
        N(NOADMINGROUP)     = Succ(N(UNKNOWNREQUEST));
        N(NOMODERGROUP)     = Succ(N(NOADMINGROUP));
        N(GROUPSARE(g))     = Succ(N(NOMODERGROUP));
        N(MEMBERSARE(m))    = Succ(Succ(N(NOMODERGROUP)));
```

```
        N(ATTRIBUTESARE(gi))= Succ(Succ(Succ(N(NOMODERGROUP))));

        ofsort Bool
        a1 eq a2    = N(a1) eq N(a2);
        a1 ne a2    = N(a1) ne N(a2);

endtype (* AckErrorType *)


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

(* Instances of messages that can be multicast (for readability). *)
(* A message type could be, for instance, a bit string. *)
type        InfoMsgType is EnumType renamedby
sortnames   InfoMsg for Enum
opnnames
            GroupIsDeleted  for Elem0 (* This one is necessary. *)
            Hello           for Elem1
            Salut           for Elem2
            GoodBye         for Elem3
            Packet          for Elem4
endtype (* InfoMsgType *)

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

(* Encoding/Decoding of messages *)
(* Several types of packets are needed. *)
type    MsgType is GCSinfoRecordType, RequestType, AckErrorType, InfoMsgType
opns
        (*************************************************************)
        (* No Message Packet.  Used for:                            *)
        (*      Group deletion                                       *)
        (*      List members                                         *)
        (*      List groups                                          *)
        (*      Deregister (from public group)                       *)
        (*************************************************************)
        NoMsg   : -> Msg

        (*************************************************************)
        (* General Packet for Group Creation:                        *)
        (*       Channel Type    (of Chan)                           *)
        (*       ChanID          (of CID)                            *)
        (*       AdminAttribute  (of Attribute)                      *)
        (*       Administrator   (of MID)                            *)
        (*       OpenedAttribute (of Attribute)                      *)
        (*       PrivateAttribute(of Attribute)                      *)
        (*       ModerAttribute  (of Attribute)                      *)
        (*       Moderator       (of MID)                            *)
        (*************************************************************)

        (* Encode  : Chan, CID, Attribute, MID, Attribute, Attribute,     *)
        (*           Attribute, MID                              -> Msg *)
        (* This packet is already defined in GCSinfoRecordType, as it     *)
        (* corresponds to the tuple that contains the GCS attributes.     *)

        (* Extraction Operations *)
        ChanType    : Msg                                       -> Chan
        ChanID      : Msg                                       -> CID
        IsAdmin     : Msg                                       -> Bool
```

```
Admin       : Msg                                        -> MID
IsOpened    : Msg                                        -> Bool
IsPrivate   : Msg                                        -> Bool
IsModerated : Msg                                        -> Bool
Moderator   : Msg                                        -> MID

(*****************************************************************)
(* Packet for Group Moderator modification:                     *)
(*      Moderator       (of MID)                                 *)
(*      ModerAttribute  (of Attribute)                           *)
(*****************************************************************)
Encode      : MID, Attribute      -> Msg
(* Modification Operations *)
SetModer    : MID, Attribute, Msg -> Msg

(*****************************************************************)
(* Packet for moderator approval:                                *)
(*      Sender     (of MID)                                      *)
(*      Message    (of Msg)                                      *)
(*****************************************************************)
ToApprove   : MID, Msg      -> AckError

(*****************************************************************)
(* Packets for multicasting                                      *)
(*****************************************************************)
Encode  : InfoMsg   -> Msg
Encode  : AckError  -> Msg
(* Modification Operations *)
GetAck  : Msg       -> AckError
GetInfo : Msg       -> InfoMsg

(*****************************************************************)
(* Packet for subscription                                       *)
(*      ChanID  (of CID)                                         *)
(*****************************************************************)
Encode  : CID       -> Msg

(*****************************************************************)
(* Packet for subscription by Administrator (for private groups)  *)
(*      NewMember   (of MID)                                     *)
(*      ChanID      (of CID)                                     *)
(*****************************************************************)
Encode      : MID, CID  -> Msg
NewMember   : Msg       -> MID

(*****************************************************************)
(* Packet for unsubscription from private group                  *)
(*      MemberID       (of MID)                                  *)
(*                                                               *)
(* Packet for Group Administrator modification:                  *)
(*      Administrator  (of MID)                                  *)
(*****************************************************************)
Encode  : MID                 -> Msg
MemberID: Msg                 -> MID
SetAdmin: MID, Attribute, Msg -> Msg

(*****************************************************************)
(* Packet for modification of Opened attribute:                  *)
(*      NewOpenAttr    (of Attribute)                            *)
```

```
        (*                                                              *)
        (* Packet for modification of Private attribute:                *)
        (*      NewPrivAttr    (of Attribute)                           *)
        (***************************************************************)
        Encode    : Attribute        -> Msg
        SetOpened : Attribute, Msg   -> Msg
        SetPrivate: Attribute, Msg   -> Msg

eqns
        forall  msg                 : Msg,
                info                : InfoMsg,
                CT                  : Chan,
                ChID                : CID,
                Adm, NewAdm, Mem,
                Mod, NewMod, Sender : MID,
                AB, OB, PB, MB,
                NewAB, NewOB,
                NewPB, NewMB        : Attribute,
                ackerr              : AckError

        ofsort Chan
        ChanType(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))    = CT;

        ofsort CID
        ChanID(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))      = ChID;
        ChanID(Encode(ChID))                                    = ChID;
        ChanID(Encode(Mem, ChID))                               = ChID;

        ofsort Bool
        IsAdmin(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))
                                                        = AB eq Administered;
        IsOpened(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))
                                                        = OB eq Opened;
            IsOpened(Encode(OB))                        = OB eq Opened;
        IsPrivate(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))
                                                        = PB eq Private;
        IsPrivate(Encode(Adm, PB))                      = PB eq Private;
        IsPrivate(Encode(PB))                           = PB eq Private;
        IsModerated(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))
                                                        = MB eq Moderated;
        IsModerated(Encode(Mod, MB))                    = MB eq Moderated;

        ofsort MID
        Admin(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))       = Adm;
        Admin(Encode(Adm))                                      = Adm;
        Moderator(Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod))   = Mod;
        Moderator(Encode(Mod, MB))                              = Mod;
        NewMember(Encode(Mem, ChID))                            = Mem;
        MemberID(Encode(Mem))                                   = Mem;

        ofsort InfoMsg
        GetInfo(Encode(info))                                   = info;

        ofsort AckError
        GetAck(Encode(ackerr))                                  = ackerr;

        ofsort Msg
        SetAdmin(NewAdm, NewAB, Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod)) =
                        Encode(CT, ChID, NewAB, NewAdm, OB, PB, MB, Mod);
```

```
        SetOpened(NewOB, Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod)) =
                        Encode(CT, ChID, AB, Adm, NewOB, PB, MB, Mod);
        SetPrivate(NewPB, Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod)) =
                        Encode(CT, ChID, AB, Adm, OB, NewPB, MB, Mod);
        SetModer(NewMod, NewMB, Encode(CT, ChID, AB, Adm, OB, PB, MB, Mod)) =
                        Encode(CT, ChID, AB, Adm, OB, PB, NewMB, NewMod);

        ofsort Nat
        (* Mapping for comparison with other Acks *)
        N(ToApprove(Sender, msg)) = Succ(Succ(Succ(Succ(N(NOMODERGROUP)))));

endtype (* MsgType *)

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

(***********************************************)
(*                                             *)
(* Buffering of requests and acknowledgements  *)
(*                                             *)
(* A generic type is created and actualized as *)
(* FIFO buffers for reqs and acks/errors.      *)
(*                                             *)
(***********************************************)

(* Generic FIFO type definition *)
type    FIFOType is Boolean
formalsorts
        Element

sorts   FIFO
opns
        Nothing :                -> FIFO
        Put     : Element, FIFO -> FIFO
        Get     : FIFO           -> Element
        Consume : FIFO           -> FIFO
        ErrorFIFO:               -> Element
        IsEmpty : FIFO           -> Bool
eqns
        forall f:FIFO, e, e2 :Element

        ofsort FIFO
        Consume(Nothing)            = Nothing;
        Consume(Put(e, Nothing))    = Nothing;
        Consume(Put(e2,Put(e,f)))   = Put(e2, Consume(Put(e, f)));
        ofsort Element
        Get(Nothing)                = ErrorFIFO;
        Get(Put(e, Nothing))        = e;
        Get(Put(e2, Put(e, f)))     = Get(Put(e, f));

        ofsort Bool
        IsEmpty(Nothing)            = true;
        IsEmpty(Put(e,f))           = false;
endtype (* FIFOType *)


(* Encoding of requests and acknowledgements as Records *)
type    BufferEncodingType is MIDType, AckErrorType, RequestType, MsgType
sorts   ReqRecord, AckErrorRecord
opns
```

```
        AckElem     : MID, AckError     -> AckErrorRecord
        ReqElem     : MID, Request, Msg -> ReqRecord

        S           : AckErrorRecord    -> MID      (* Extract Sender *)
        A           : AckErrorRecord    -> AckError (* Extract AckError *)
        S           : ReqRecord         -> MID      (* Extract Sender *)
        R           : ReqRecord         -> Request  (* Extract Request *)
        M           : ReqRecord         -> Msg      (* Extract Message *)
eqns
        forall S1:MID, A1:AckError, R1:Request, M1:Msg

        ofsort MID
        S(AckElem(S1, A1))      = S1;
        S(ReqElem(S1, R1, M1))  = S1;

        ofsort AckError
        A(AckElem(S1, A1))      = A1;

        ofsort Request
        R(ReqElem(S1, R1, M1))  = R1;

        ofsort Msg
        M(ReqElem(S1, R1, M1))  = M1;

endtype (* BufferEncodingType *)


(* Actualization (and renaming) of a Buffer type for records of requests *)
type            FIFOreqsType0 is FIFOType
actualizedby    BufferEncodingType using
sortnames
                ReqRecord for Element
endtype         (* FIFOreqsType0 *)

type        FIFOreqsType is FIFOreqsType0 renamedby
sortnames
            FIFOreqs for FIFO
opnnames
            NoReq    for Nothing
endtype     (* FIFOreqsType *)


(* Actualization (and renaming) of a Buffer type for records of acks *)
type            FIFOackerrsType0 is FIFOType
actualizedby    BufferEncodingType using
sortnames
                AckErrorRecord for Element
endtype         (* FIFOackerrsType0 *)

type        FIFOackerrsType is FIFOackerrsType0 renamedby
sortnames
            FIFOackerrs for FIFO
opnnames
            NoAckErr for Nothing
endtype     (* FIFOackerrsType *)

(*============================================*)
(*              Main behaviour               *)
(*============================================*)
```

```
behaviour

    Control_Team [mgcs_ch, gcs_ch, out_ch]

where

(*********************************************)
(*                                           *)
(*        Structure of Control Team          *)
(*                                           *)
(*********************************************)

process Control_Team [mgcs_ch, gcs_ch, out_ch] : noexit :=

    hide sgcs_ch, agcs_ch in (* Spawning and Administrative channels *)
    (
        MGCS[sgcs_ch, agcs_ch, mgcs_ch](NoGCS)           (* Management *)

        |[sgcs_ch, agcs_ch]|

        Spawn_GCS[sgcs_ch, agcs_ch, gcs_ch, out_ch]      (* GCS spawning *)
    )

endproc (* Control_Team *)


(***************************************************************************)
(*                                                                         *)
(*      Manager of Group Communication Servers (MGCS)                      *)
(*                                                                         *)
(*      Listens on the mgcs_ch channel for requests for:                   *)
(*          - the creation of a new Group Communication Server (GCS)       *)
(*          - the list of existing GCS                                     *)
(*      Uses sgcs_ch to:                                                   *)
(*          - spawn new groups                                             *)
(*      Uses agcs_ch to:                                                   *)
(*          - learn about the deletion of a group                          *)
(*                                                                         *)
(***************************************************************************)

process MGCS[sgcs_ch, agcs_ch, mgcs_ch](GCSlist:GroupList) :noexit:=

    (* Request for the creation of a new GCS *)
    (* the GID of the new group must not be used by an existing group *)

    mgcs_ch !ToMGCS ?caller:MID !CREATEGROUP ?newgroupid:GID ?infos:Msg;
    (
        [newgroupid IsIn GCSlist] ->
            mgcs_ch !FromMGCS !caller! GROUPEXISTS !newgroupid;
                (*_PROBE_*) (* P_0 *)
                MGCS[sgcs_ch, agcs_ch, mgcs_ch](GCSlist)
        []
        [newgroupid NotIn GCSlist] ->
            sgcs_ch !CREATEGROUP !newgroupid
                    !Insert(caller.ChanID(infos), NoMBR) ! infos;
                mgcs_ch !FromMGCS !caller !GROUPCREATED !newgroupid;
                    (*_PROBE_*) (* P_1 *)
                    MGCS[sgcs_ch, agcs_ch, mgcs_ch]
```

```
                                                  (Insert(newgroupid, GCSlist))
    )

    []

    (* Request for the list of existing groups *)
    mgcs_ch !ToMGCS ?caller:MID !GROUPS;
        mgcs_ch !FromMGCS !caller !GROUPSARE(GCSlist);
            (*_PROBE_*) (* P_2 *)
            MGCS[sgcs_ch, agcs_ch, mgcs_ch](GCSlist)

    []

    (* Process destruction of a GCS. All verifications were done by the GCS *)
    (* Used to update the group list database *)
    [GCSlist ne NoGCS]->
        (* allowed only if there exists at least one group *)
        agcs_ch !GROUPDELETED ?id:GID;
            (*_PROBE_*) (* P_3 *)
            MGCS [sgcs_ch, agcs_ch, mgcs_ch](Remove(id, GCSlist))

endproc (* MGCS *)

(*************************************************************************)
(*                                                                       *)
(*      Spawn Group Communication Server                                 *)
(*                                                                       *)
(*      Creates a new GCS and forwards messages to it.                   *)
(*                                                                       *)
(*************************************************************************)

process Spawn_GCS[sgcs_ch, agcs_ch, gcs_ch, out_ch] :noexit:=

    sgcs_ch !CREATEGROUP ?id:GID ?mbrL:MemberList ?infos: Msg;
        (
            GCS_Team[agcs_ch, gcs_ch, out_ch] (id, mbrL, infos)
            |||
            Spawn_GCS[sgcs_ch, agcs_ch, gcs_ch, out_ch]
        )

endproc (* Spawn_GCS *)


(****************************************)
(*                                      *)
(*        Structure of GCS Team         *)
(*                                      *)
(****************************************)

process GCS_Team[agcs_ch, gcs_ch, out_ch]
                (id:GID, mbrL:MemberList, infos:Msg) : exit :=

    hide inter_ch in
        BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id, false, NoReq, NoAckErr)
        |[inter_ch]|
        GCS[inter_ch, out_ch] (id, mbrL, infos)

endproc (* GCS_Team *)
```

```
(***************************************************************************)
(*                                                                         *)
(*      BiDirBuffer                                                        *)
(*                                                                         *)
(*      Routes messages for group ID from gcs_ch to inter_ch; ignores others*)
(*      Routes back acknowledgements to sender or to master.               *)
(*      Used for decoupling. Avoids waiting for all GCS to be ready for a  *)
(*      request to be processed. Increases concurrency.                    *)
(*                                                                         *)
(*      Requests and acknowledgements/errors are buffered.                 *)
(*      We use two infinite 2-way buffers that hold two ordered lists      *)
(*      (for acks/errors and requests).                                    *)
(*                                                                         *)
(***************************************************************************)

process BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id:GID,
                                               terminated: Bool,
                                               bufreqs:FIFOreqs,
                                               bufackerrs:FIFOackerrs ):exit:=
    (* Buffer requests from senders. *)
    RequestBuffer[agcs_ch, gcs_ch, inter_ch](id,terminated,bufreqs,bufackerrs)

    []

    (* Buffer acks and errors from GCS. *)
    AckErrorBuffer[agcs_ch, gcs_ch, inter_ch](id,terminated,bufreqs,bufackerrs)

    []

    (* Terminate after everyone has been informed of a group deletion. *)
    (* Wait for Req and Ack buffers to be empty *)
    [IsEmpty(bufreqs) and IsEmpty(bufackerrs) and terminated] ->
        inter_ch ! ToGCS !GROUPDELETED;
            (*_PROBE_*) (* P_4 *)
            exit

    where

    process RequestBuffer[agcs_ch, gcs_ch, inter_ch](id:GID,
                                                     terminated: Bool,
                                                     bufreqs:FIFOreqs,
                                                     bufackerrs:FIFOackerrs)
                                                     :exit:=
        (* Buffer requests from senders. *)
        (* Refuse them if group is deleted (terminated) *)
        [not(terminated)] ->
            (
                gcs_ch !ToGCS ?sender:MID !id ?req:Request ?msg:Msg;
                    (*_PROBE_*) (* P_5 *)
                    BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id, terminated,
                          Put(ReqElem(sender, req, msg), bufreqs), bufackerrs)
            )

        []

        (* Forward buffered requests to GCS *)
        [not(IsEmpty(bufreqs))] ->
            (
                inter_ch !ToGCS !S(Get(bufreqs)) !R(Get(bufreqs))
```

```
                                                 !M(Get(bufreqs));
                    (*_PROBE_*) (* P_6 *)
                    BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id, terminated,
                                Consume(bufreqs), bufackerrs)
            )

    endproc (* RequestBuffer *)


    process AckErrorBuffer[agcs_ch, gcs_ch, inter_ch](id:GID,
                                                      terminated: Bool,
                                                      bufreqs:FIFOreqs,
                                                      bufackerrs:FIFOackerrs )
                                                      :exit:=
        (* Buffer acks from GCS. *)
        inter_ch !FromGCS ?sender:MID ?ackerr:AckError;
            (*_PROBE_*) (* P_7 *)
            BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id, terminated,
                        bufreqs, Put(AckElem(sender, ackerr), bufackerrs))

        []

        (* Forward buffered acks to senders *)
        [not(IsEmpty(bufackerrs))] ->
            (
                [A(Get(bufackerrs)) eq GROUPDELETED] ->
                    (* Intercept this special case of ack. *)
                    (* Inform MGCS of deletion, for database update. *)
                    agcs_ch ! GROUPDELETED ! id;
                       (* Forward ack to sender, empty the request buffer, *)
                       (* and terminate listening. We could add a process *)
                       (* that would tell the senders of these buffered *)
                       (* requests that the goup was deleted... *)
                        gcs_ch !FromGCS !S(Get(bufackerrs)) !A(Get(bufackerrs))
                                                             of AckError !id;
                            (*_PROBE_*) (* P_8 *)
                            BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id,
                                         true, NoReq, Consume(bufackerrs))

                []

                [A(Get(bufackerrs)) ne GROUPDELETED] ->
                    (* Forward ack to sender but don’t empty the req buffer *)
                    gcs_ch !FromGCS !S(Get(bufackerrs)) !A(Get(bufackerrs))
                                                         of AckError !id;
                        (*_PROBE_*) (* P_9 *)
                        BiDirBuffer[agcs_ch, gcs_ch, inter_ch](id, terminated,
                                    bufreqs, Consume(bufackerrs))
            )

    endproc (* AckErrorBuffer *)

endproc (* BiDirBuffer *)


(*****************************************************************************)
(*                                                                           *)
(*      Group Communication Server (GCS)                                     *)
(*                                                                           *)
```

```
(*      inter_ch: to communicate with sender and master (for administration)*)
(*      out_ch:   to multicast messages to members                          *)
(*                                                                          *)
(*      Receives requests on inter_ch with event structure:                 *)
(*                  !MID !Request !Msg                                      *)
(*      Gives acknowledgements with event structure:                        *)
(*                  !MID !AckError                                          *)
(*                                                                          *)
(**************************************************************************)


process GCS[inter_ch, out_ch](id:GID, mbrL:MemberList, infos:Msg) :exit:=

    (* gets the request from the sender *)
    inter_ch !ToGCS ?sender:MID ?req:Request ?msg:Msg [sender ne Nobody];
    (
        (* get the GCS attributes (infos) *)
        [req eq GETATTRIBUTES] ->
            Req_GetAttributes[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* group deletion *)
        [req eq DELETEGROUP] ->
            Req_DeleteGroup[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* registration *)
        [req eq REGISTER] ->
            Req_Register[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* provides the list of group members *)
        [req eq MEMBERS] ->
            Req_Members[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* deregistration *)
        [req eq DEREGISTER] ->
            Req_DeRegister[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* multicast messages; will block until successful *)
        [req eq MULTICAST] ->
            Req_Multicast[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* change the administrator and the Administered attribute *)
        [req eq CHANGEADMIN] ->
            Req_ChangeAdmin[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* change the Opened attribute *)
        [req eq CHANGEOPENATTR] ->
            Req_ChangeOpenAttr[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* change the Private attribute *)
        [req eq CHANGEPRIVATTR] ->
            Req_ChangePrivAttr[inter_ch, out_ch](sender, msg, id, mbrL, infos)
        []
        (* change the moderator and the Moderated attribute *)
        [req eq CHANGEMODER] ->
            Req_ChangeModer[inter_ch, out_ch](sender, msg, id, mbrL, infos)
    )

    where

```

```
    process Req_GetAttributes[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                               mbrL:MemberList, infos:Msg) :exit:=

        [not(IsAdmin(infos)) or (Admin(infos) eq sender)] ->
        (* Either NonAdministered, or sender is administrator *)
            (
                inter_ch !FromGCS !sender !ATTRIBUTESARE(infos);
                    (*_PROBE_*) (* P_10 *)
                    GCS[inter_ch, out_ch](id, mbrL, infos)
            )
        []
        [(IsAdmin(infos) and (Admin(infos) ne sender))] ->
        (* Administered group, and sender is not the administrator *)
            (
                inter_ch !FromGCS !sender !NOTADMIN;
                    (*_PROBE_*) (* P_11 *)
                    GCS[inter_ch, out_ch](id, mbrL, infos)
            )

    endproc (* Req_GetAttributes *)


    process Req_DeleteGroup[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                               mbrL:MemberList, infos:Msg) :exit:=

        [IsAdmin(infos) and (Admin(infos) eq sender)] ->
        (* Administered group, and sender is the administrator *)
            (
                (* Inform members other than sender *)
                Multicast[inter_ch](sender,Encode(GROUPWASDELETED),
                                    RemoveMBR(sender, mbrL), false)>>
                    inter_ch !FromGCS !sender !GROUPDELETED;
                        inter_ch !ToGCS !GROUPDELETED;
                            (*_PROBE_*) (* P_12 *)
                            exit
            )
        []
        [IsAdmin(infos) and (Admin(infos) ne sender)] ->
        (* Administered group, and sender is not the administrator *)
            inter_ch !FromGCS !sender !NOTADMIN;
                (*_PROBE_*) (* P_13 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)
        []
        (* Non administered group *)
        [not(IsAdmin(infos))] ->
            inter_ch !FromGCS !sender !NOADMINGROUP;
                (*_PROBE_*) (* P_14 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)

    endproc (* Req_DeleteGroup *)


    process Req_Register[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                            mbrL:MemberList, infos:Msg) :exit:=

        [not(IsPrivate(infos))] ->
        (* Register only if group is public *)
            inter_ch !FromGCS !sender !REGISTERED;
            (
```

```
            [sender NotIn MembersOnly(mbrL)] ->
                (* Insert pair MID.CID *)
                (*_PROBE_*) (* P_15 *)
                GCS[inter_ch, out_ch]
                   (id, Insert(sender.ChanID(msg), mbrL), infos)
            []

            [sender IsIn MembersOnly(mbrL)] ->
                (* Modify CID only *)
                (*_PROBE_*) (* P_16 *)
                GCS[inter_ch, out_ch]
                    (id, Insert(sender.ChanID(msg),
                                RemoveMBR(sender,mbrL)), infos)
        )
    []
    [IsPrivate(infos)] ->
    (
        [sender ne Admin(infos)] ->
        (* Cannot register; group is private *)
            inter_ch !FromGCS !sender !NOTADMIN;
                (*_PROBE_*) (* P_17 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)
        []
        [sender eq Admin(infos)] ->
        (* Admin can register another member in private group *)
            inter_ch !FromGCS !sender !REGISTERED;
            (
                [NewMember(msg) NotIn MembersOnly(mbrL)] ->
                   (* Insert pair MID.CID *)
                    (*_PROBE_*) (* P_18 *)
                    GCS[inter_ch, out_ch]
                        (id, Insert(NewMember(msg).ChanID(msg),
                                    mbrL), infos)
                []
                [NewMember(msg) IsIn MembersOnly(mbrL)] ->
                    (* Modify CID only *)
                    (*_PROBE_*) (* P_19 *)
                    GCS[inter_ch, out_ch]
                        (id, Insert(NewMember(msg).ChanID(msg),
                         RemoveMBR(NewMember(msg),mbrL)), infos)
            )
    )

endproc (* Req_Register *)


process Req_Members[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                      mbrL:MemberList, infos:Msg) :exit:=

    (* Check whether group is private *)
    [not(IsPrivate(infos)) or
        (IsPrivate(infos) and (sender IsIn MembersOnly(mbrL)))] ->
        inter_ch !FromGCS !sender !MEMBERSARE(MembersOnly(mbrL));
            (*_PROBE_*) (* P_20 *)
            GCS[inter_ch, out_ch](id, mbrL, infos)
    []
    [IsPrivate(infos) and (sender NotIn MembersOnly(mbrL))] ->
        inter_ch !FromGCS !sender !MEMBERNOTINGROUP;
            (*_PROBE_*) (* P_21 *)
```

```
                GCS[inter_ch, out_ch](id, mbrL, infos)

    endproc (* Req_Members *)


    process Req_DeRegister[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                              mbrL:MemberList, infos:Msg) :exit:=

        [not(IsAdmin(infos)) or (sender ne Admin(infos))] ->
         (* When group is not administered, DeRegister only if member is in *)
         (* group. Same idea when administered and sender is not admin.     *)
            (
                [sender NotIn MembersOnly(mbrL)] ->
                    inter_ch !FromGCS !sender !MEMBERNOTINGROUP;
                        (*_PROBE_*) (* P_22 *)
                        GCS[inter_ch, out_ch](id, mbrL, infos)

                []

                [sender IsIn MembersOnly(mbrL)] ->
                    inter_ch !FromGCS !sender !DEREGISTERED;
                    (
                        [Card(mbrL) eq Succ(0)] ->
                            (* The GCS dies if no member left *)
                            (* We do not need to inform the members... *)
                                inter_ch !FromGCS !sender !GROUPDELETED;
                                    inter_ch !ToGCS !GROUPDELETED;
                                        (*_PROBE_*) (* P_23 *)
                                        exit
                        []
                        [Card(mbrL) ne Succ(0)] ->
                            (*_PROBE_*) (* P_24 *)
                            GCS[inter_ch, out_ch]
                               (id, RemoveMBR(sender, mbrL), infos)
                    )
            )
        []
        [IsAdmin(infos) and (sender eq Admin(infos))] ->
        (* When group is administered and the sender is the administrator, *)
        (* DeRegister the member named by the admin. *)
            (
                [MemberID(msg) IsIn MembersOnly(mbrL)] ->
                    inter_ch !FromGCS !sender !DEREGISTERED;
                    (
                        [Card(mbrL) eq Succ(0)] ->
                            (* The GCS dies if no member left *)
                            (* We do not need to inform the members... *)
                                inter_ch !FromGCS !sender !GROUPDELETED;
                                    inter_ch !ToGCS !GROUPDELETED;
                                        (*_PROBE_*) (* P_25 *)
                                        exit
                        []
                        [Card(mbrL) ne Succ(0)] ->
                            (*_PROBE_*) (* P_26 *)
                            GCS[inter_ch, out_ch]
                               (id, RemoveMBR(MemberID(msg), mbrL), infos)
                    )
                []
                [MemberID(msg) NotIn MembersOnly(mbrL)] ->
```

```
                        inter_ch !FromGCS !sender !MEMBERNOTINGROUP;
                            (*_PROBE_*) (* P_27 *)
                            GCS[inter_ch, out_ch](id, mbrL, infos)
                )

        endproc (* Req_DeRegister *)


       process Req_Multicast[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                               mbrL:MemberList, infos:Msg) :exit:=

        (* Is the group Opened, or is the sender a member of the group? *)
        [IsOpened(infos) or
         (not(IsOpened(infos)) and (sender IsIn MembersOnly(mbrL)))] ->
            (
                (* Not moderated, or sender is moderator *)
                [not(IsModerated(infos)) or (sender eq Moderator(infos))] ->
                    (
                        Multicast[out_ch](sender, msg, mbrL, true) >>
                            inter_ch !FromGCS !sender !MESSAGESENT;
                                (*_PROBE_*) (* P_28 *)
                                GCS[inter_ch, out_ch](id, mbrL, infos)
                    )
                []
                (* Moderated, and sender is not moderator. *)
                (* Forward to group moderator only, for approval *)
                [IsModerated(infos) and (sender ne Moderator(infos))] ->
                    (
                        inter_ch !FromGCS !Moderator(infos) !ToApprove(sender,msg);
                            inter_ch !FromGCS !sender !SENTTOMODERATOR;
                                (*_PROBE_*) (* P_29 *)
                                GCS[inter_ch, out_ch](id, mbrL, infos)
                    )
            )
        []
        [not(IsOpened(infos)) and (sender NotIn MembersOnly(mbrL))]->
            (
                inter_ch !FromGCS !sender !MEMBERNOTINGROUP;
                    (*_PROBE_*) (* P_30 *)
                    GCS[inter_ch, out_ch](id, mbrL, infos)
            )

        endproc (* Req_Multicast *)


        process Req_ChangeAdmin[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                               mbrL:MemberList, infos:Msg) :exit:=

            (* The sender has the privilege. *)
            [IsAdmin(infos) and (Admin(infos) eq sender)] ->
                (
                    [Admin(msg) IsIn mbrL] ->
                        inter_ch !FromGCS !sender !ADMINCHANGED;
                            (*_PROBE_*) (* P_31 *)
                            GCS[inter_ch, out_ch]
                               (id, mbrL, SetAdmin(Admin(msg),Administered, infos))
                    []
                    [Admin(msg) eq Nobody] ->
                        (* If it’s Nobody, then set the group to NonAdministered *)
```

```
                 inter_ch !FromGCS !sender !ADMINCHANGED;
                     (*_PROBE_*) (* P_32 *)
                     GCS[inter_ch, out_ch]
                        (id, mbrL, SetAdmin(Nobody, NonAdministered, infos))
             []
             [(Admin(msg) NotIn mbrL) and (Admin(msg) ne Nobody)] ->
                 inter_ch !FromGCS !sender !MEMBERNOTINGROUP;
                     (*_PROBE_*) (* P_33 *)
                     GCS[inter_ch, out_ch](id, mbrL, infos)
         )
    []
    (* The sender does not have the privilege to change the admin *)
    [IsAdmin(infos) and (Admin(infos) ne sender)] ->
        inter_ch !FromGCS !sender !NOTADMIN;
            (*_PROBE_*) (* P_34 *)
            GCS[inter_ch, out_ch](id, mbrL, infos)
    []
    (* The group does not have an administrator *)
    [not(IsAdmin(infos))] ->
        inter_ch !FromGCS !sender !NOADMINGROUP;
            (*_PROBE_*) (* P_35 *)
            GCS[inter_ch, out_ch](id, mbrL, infos)

endproc (* Req_ChangeAdmin *)


process Req_ChangeOpenAttr[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                      mbrL:MemberList, infos:Msg) :exit:=

    [IsAdmin(infos) and (Admin(infos) eq sender)] ->
    (* Administered group, and sender is the administrator *)
        (
            inter_ch !FromGCS !sender !OPENATTRCHANGED;
            (
                [IsOpened(msg)]->
                    (*_PROBE_*) (* P_36 *)
                    GCS[inter_ch, out_ch](id,mbrL,SetOpened(Opened, infos))
                []
                [not(IsOpened(msg))]->
                    (*_PROBE_*) (* P_37 *)
                    GCS[inter_ch, out_ch](id,mbrL,SetOpened(Closed, infos))
            )
        )
    []
    [IsAdmin(infos) and (Admin(infos) ne sender)] ->
    (* Administered group, and sender is not the administrator *)
        (
            inter_ch !FromGCS !sender !NOTADMIN;
                (*_PROBE_*) (* P_38 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)
        )
    []
    [not(IsAdmin(infos))] ->
    (* NonAdministered group *)
        (
            inter_ch !FromGCS !sender !NOADMINGROUP;
                (*_PROBE_*) (* P_39 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)
        )
```

```

endproc (* Req_ChangeOpenAttr *)


process Req_ChangePrivAttr[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                      mbrL:MemberList, infos:Msg) :exit:=

    [IsAdmin(infos) and (Admin(infos) eq sender)] ->
    (* Administered group, and sender is the administrator *)
        (
            inter_ch !FromGCS !sender !PRIVATTRCHANGED;
            (
                [IsPrivate(msg)]->
                    (*_PROBE_*) (* P_40 *)
                    GCS[inter_ch, out_ch](id, mbrL,
                                          SetPrivate(Private, infos))
                []
                [not(IsPrivate(msg))]->
                    (*_PROBE_*) (* P_41 *)
                    GCS[inter_ch, out_ch](id, mbrL,
                                          SetPrivate(Public, infos))
            )
        )
    []
    [IsAdmin(infos) and (Admin(infos) ne sender)] ->
    (* Administered group, and sender is not the administrator *)
        (
            inter_ch !FromGCS !sender !NOTADMIN;
                (*_PROBE_*) (* P_42 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)
        )
    []
    [not(IsAdmin(infos))] ->
    (* NonAdministered group *)
        (
            inter_ch !FromGCS !sender !NOADMINGROUP;
                (*_PROBE_*) (* P_43 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)
        )

endproc (* Req_ChangePrivAttr *)


process Req_ChangeModer[inter_ch, out_ch](sender: MID, msg: Msg, id:GID,
                                       mbrL:MemberList, infos:Msg) :exit:=

    (* The sender has the privilege. *)
    [(IsModerated(infos) and (Moderator(infos) eq sender)) or
     (IsAdmin(infos) and (Admin(infos) eq sender))] ->
        (
          [Moderator(msg) ne Nobody]->
            (
              [IsOpened(infos) or (Moderator(msg) IsIn mbrL)] ->
                  inter_ch !FromGCS !sender !MODERCHANGED;
                  (
                      [IsModerated(msg)]->
                          (*_PROBE_*) (* P_44 *)
                          GCS[inter_ch, out_ch]
                             (id, mbrL, SetModer(Moderator(msg),
```

```
                                                     Moderated, infos))
                          []
                          [not(IsModerated(msg))]->
                              (*_PROBE_*) (* P_45 *)
                              (* Put Nobody as new moderator *)
                              GCS[inter_ch, out_ch]
                                 (id, mbrL, SetModer(Nobody,
                                                     NonModerated, infos))
                      )
                  []
                  (* If the group is closed, *)
                  (* then moderator must be a group member *)
                  [not(IsOpened(infos)) and (Moderator(msg) NotIn mbrL)] ->
                      inter_ch !FromGCS !sender !MEMBERNOTINGROUP;
                          (*_PROBE_*) (* P_46 *)
                          GCS[inter_ch, out_ch](id, mbrL, infos)
                )
              []
              [Moderator(msg) eq Nobody]->
                  (* Special case where the group becomes NonModerated, *)
                  (* whatever the value of the new attribute *)
                  (
                      inter_ch !FromGCS !sender !MODERCHANGED;
                          (*_PROBE_*) (* P_47 *)
                          GCS[inter_ch, out_ch]
                             (id, mbrL, SetModer(Nobody,
                                                 NonModerated, infos))
                  )
            )
        []
        (* The sender does not have the privilege to change moder *)
        (* (neither admin, nor moderator)*)
        [(IsModerated(infos) and (Moderator(infos) ne sender)) and
         ( (IsAdmin(infos) and (Admin(infos) ne sender)) or
           not(IsAdmin(infos)) )] ->
                inter_ch !FromGCS !sender !NOTMODER;
                    (*_PROBE_*) (* P_48 *)
                    GCS[inter_ch, out_ch](id, mbrL, infos)
        []
        (* The group does not have a moderator *)
        (* (and the sender is not admin) *)
        [not(IsModerated(infos)) and
         not(IsAdmin(infos) and (Admin(infos) eq sender))] ->
            inter_ch !FromGCS !sender !NOMODERGROUP;
                (*_PROBE_*) (* P_49 *)
                GCS[inter_ch, out_ch](id, mbrL, infos)

    endproc (* Req_ChangeModer *)

endproc (* GCS *)


(****************************************************************************)
(*                                                                          *)
(*      Multicast                                                           *)
(*                                                                          *)
(*      Send the message to all subscribers of the group (concurrently).   *)
(*      No other message will be multicast until the first one is sent to  *)
(*      all members of the group.                                           *)
```

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

process Multicast[out](sender:MID, msg:Msg, mbrL:MemberList, UseChannel: Bool)
                      : exit :=

    [mbrL eq NoMBR] ->
         (*_PROBE_*) (* P_50 *)
         exit

    []

    [mbrL ne NoMBR] ->
        (
            (
                [UseChannel] ->
                    (
                        (* Multicasts message to members on their *)
                        (* appropriate data channel, concurrently *)
                        out !Top(mbrL) !sender !msg;
                            (*_PROBE_*) (* P_51 *)
                            exit
                        |||
                        (* loop... *)
                        (*_PROBE_*) (* P_52 *)
                        Multicast[out](sender, msg, Tail(mbrL), UseChannel)
                    )
                []
                [not(UseChannel)] ->
                    (* Use request/AckError channel, sequentially *)
                    (* (for group deletion) *)
                    out !FromGCS !MID(Top(mbrL)) !GetAck(msg);
                        (*_PROBE_*) (* P_53 *)
                        Multicast[out](sender, msg, Tail(mbrL), UseChannel)
            )
        )

endproc (* Multicast *)


(*============================================*)
(*            UCM-based Test Cases            *)
(*============================================*)

(****************************************************************************)
(*                                                                          *)
(*      Group Creation                                                      *)
(*                                                                          *)
(****************************************************************************)


(* Acceptance test : Checks group creation (3 tests) *)
process Test_1 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Creates from empty GCS list *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    success; stop
```

```
    []

    (* Creates two different groups *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    success; stop

    []

    (* Uses twice the same GID *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group1 !Encode(Video, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPEXISTS !Group1;
    success; stop

endproc (* Test_1: TestUCMcreationA *)


(* Rejection test : Checks group creation (2 tests) *)
process Test_2 [mgcs_ch, reject]:noexit :=

    (* Can't create from empty GCS list *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 ?reqack:AckError !Group1 [reqack ne GROUPCREATED];
    reject; stop

    []

    (* Uses twice the same GID *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group1 !Encode(Video, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 ?reqerr:AckError !Group1 [reqerr ne GROUPEXISTS];
    reject; stop

endproc (* Test_2: TestUCMcreationR *)



(*************************************************************************)
(*                                                                       *)
(*      List of Groups                                                   *)
(*                                                                       *)
(*************************************************************************)

(* Assumes that Creation request is operational. *)

(* Acceptance test : Checks list of groups (3 tests) *)
```

```
process Test_3 [mgcs_ch, success]:noexit :=

    (* Checks empty GCS list when starting MGCS *)
    mgcs_ch !ToMGCS !User1 !GROUPS;
    mgcs_ch !FromMGCS !User1 !GROUPSARE(NoGCS);
    success; stop

    []

    (* 1 group in list *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User1 !GROUPS;
    mgcs_ch !FromMGCS !User1 !GROUPSARE(Insert(Group1, NoGCS));
    success; stop

    []

    (* 2 groups in list *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    mgcs_ch !ToMGCS !User1 !GROUPS;
    mgcs_ch !FromMGCS !User1 !GROUPSARE(Insert(Group2, Insert(Group1, NoGCS)));
    success; stop

endproc (* Test_3: TestUCMgrouplistA *)


(* Rejection test : Checks list of groups (3 tests) *)
process Test_4 [mgcs_ch, reject]:noexit :=

    (* Checks empty GCS list when starting MGCS *)
    mgcs_ch !ToMGCS !User1 !GROUPS;
    mgcs_ch !FromMGCS !User1 ?thelist:AckError [thelist ne GROUPSARE(NoGCS)];
    reject; stop

    []
    (* 1 group in list *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User1 !GROUPS;
    mgcs_ch !FromMGCS !User1 ?thelist:AckError
                             [thelist ne GROUPSARE(Insert(Group1, NoGCS))];
    reject; stop

    []

    (* 2 groups in list *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
```

```
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 ?thelist:AckError
                [thelist ne GROUPSARE(Insert(Group2, Insert(Group1, NoGCS)))];
    reject; stop

endproc (* Test_4: TestUCMgrouplistR *)

(***************************************************************************)
(*                                                                         *)
(*      Attributes Checking                                                *)
(*                                                                         *)
(***************************************************************************)

(* Assumes that Creation is operational. *)

(* Acceptance test : Checks the attributes of a group (3 tests) *)
process Test_5 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Non-administered group, anyone can get the attributes *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User2 !ATTRIBUTESARE(Encode(Mail, Chan3,NonAdministered,
                      Nobody, Opened, Public, NonModerated, Nobody)) !Group4;
    success; stop

    []

    (* Administered group, request by admin *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3, Administered,
                       User3, Closed, Private, NonModerated, Nobody)) !Group4;
    success; stop

    []

    (* Administered, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User2 !NOTADMIN !Group4;
    success; stop

endproc (* Test_5: TestUCMattrA *)


(* Rejection test : Checks the attributes of a group (3 tests) *)
process Test_6 [mgcs_ch, gcs_ch, reject]:noexit :=

    (* Non-administered group, anyone can get the attributes *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
```

```
    gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4
           [reqack ne ATTRIBUTESARE(Encode(Mail, Chan3, NonAdministered,
                              Nobody, Opened, Public, NonModerated, Nobody))];
    reject; stop

    []

    (* Administered group, request by admin *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
           [reqack ne ATTRIBUTESARE(Encode(Mail, Chan3, Administered, User3,
                                     Closed, Private, NonModerated, Nobody))];
    reject; stop

    []

    (* Administered, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4 [reqack ne NOTADMIN];
    reject; stop

endproc (* Test_6: TestUCMattrR *)


(***************************************************************************)
(*                                                                         *)
(*      Member Registration                                                *)
(*                                                                         *)
(***************************************************************************)

(* Assumes that Creation request is operational. *)

(* Acceptance test : Checks member registration within a group (6 tests) *)
process Test_7 [mgcs_ch, gcs_ch, out_ch, success]:noexit :=

    (* 1 member in group (creator) *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Video, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !REGISTER !Encode(Chan3);
    gcs_ch !FromGCS !User2 !REGISTERED !Group1;
    success; stop

    []

    (* 2 new members in public group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User2 !REGISTERED !Group1;
    gcs_ch !ToGCS !User3 !Group1 !REGISTER !Encode(Chan3);
```

```
    gcs_ch !FromGCS !User3 !REGISTERED !Group1;
    success; stop

    []

    (* 1 new member in private, administered group, by admin *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(User2, Chan2);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    success; stop

    []

    (* 1 new member in private, administered group, not by admin *)
    (* (self and 3rd party) *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User2 !NOTADMIN !Group1;
    gcs_ch !ToGCS !User2 !Group1 !REGISTER !Encode(User3, Chan3);
    gcs_ch !FromGCS !User2 !NOTADMIN !Group1;
    success; stop

    []

    (* Change the CID of a member in a non-administered group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    (* Verification *)
    gcs_ch !ToGCS !User1 !Group1 !MULTICAST !Encode(Hello);
    out_ch !User1.Chan2 !User1 !Encode(Hello);
    gcs_ch !FromGCS !User1 !MESSAGESENT !Group1;
    success; stop

    []

    (* Change the CID of a member in an administered private group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(User2, Chan3);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(User2, Chan1);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    (* Verification *)
    gcs_ch !ToGCS !User1 !Group1 !MULTICAST !Encode(Hello);
    out_ch !User1.Chan4 !User1 !Encode(Hello); (* Any order... *)
    out_ch !User2.Chan1 !User1 !Encode(Hello);
    gcs_ch !FromGCS !User1 !MESSAGESENT !Group1;
    success; stop

endproc (* Test_7: TestUCMmembersA *)
```

```
(* Rejection test : Checks member registration within a group (4 tests) *)
process Test_8 [mgcs_ch, gcs_ch, out_ch, reject]:noexit :=

    (* 1 new member in public group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group1 [reqack ne REGISTERED];
    reject; stop

    []

    (* 1 new member in private, administered group, by admin *)
    mgcs_ch !ToMGCS !User4 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             Administered, User4, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User4 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User4 !Group1 !REGISTER !Encode(User2, Chan3);
    gcs_ch !FromGCS !User4 ?reqack:AckError !Group1 [reqack ne REGISTERED];
    reject; stop

    []

    (* 1 new member in private, administered group, not by admin *)
    mgcs_ch !ToMGCS !User4 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             Administered, User4, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User4 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(Chan3);
    gcs_ch !FromGCS !User1 ?reqack:AckError !Group1 [reqack ne NOTADMIN];
    reject; stop

    []

    (* Change the CID of a group member *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    (* Verification *)
    gcs_ch !ToGCS !User1 !Group1 !MULTICAST !Encode(Hello);
    out_ch !User1.Chan4 !User1 !Encode(Hello);  (* Should deadlock here *)
    gcs_ch !FromGCS !User1 ?reqack:AckError !Group1 [reqack ne MESSAGESENT];
    reject; stop

endproc (* Test_8: TestUCMregR *)


(**************************************************************************)
(*                                                                        *)
(*      List of Members                                                   *)
(*                                                                        *)
(**************************************************************************)

(* Assumes that Creation and Registration requests are operational. *)

(* Acceptance test : Checks member registration within a group (4 tests) *)
process Test_9 [mgcs_ch, gcs_ch, success]:noexit :=
```

```

    (* 1 member in group (creator) *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Video, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User2 !MEMBERSARE(Insert(User1,Empty)) !Group1;
    success; stop

    []

    (* 2 members in group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User3 !Group1 !REGISTER !Encode(Chan2 of CID);
    gcs_ch !FromGCS !User3 !REGISTERED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User2 !MEMBERSARE(Insert(User3, Insert(User1, Empty)))
                          !Group1;
    success; stop

    []

    (* Sender is a member of private group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(User3, Chan4);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    gcs_ch !ToGCS !User3 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User3 !MEMBERSARE(Insert(User3, Insert(User1, Empty)))
                          !Group1;
    success; stop

    []

    (* Sender is not a member of private group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User2 !MEMBERNOTINGROUP !Group1;
    success; stop

endproc (* Test_9: TestUCMmembersA *)


(* Rejection test : Checks member registration within a group (3 tests) *)
process Test_10 [mgcs_ch, gcs_ch, reject]:noexit :=

    (* 1 member in group (creator) *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group1
                            [reqack ne MEMBERSARE(Insert(User1,Empty))];
    reject; stop
```

```
    []

    (* Sender is a member of private group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan4,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(User3, Chan3);
    gcs_ch !FromGCS !User1 !REGISTERED !Group1;
    gcs_ch !ToGCS !User3 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group1
                  [reqack ne MEMBERSARE(Insert(User3, Insert(User1, Empty)))];
    reject; stop

    []

    (* Sender is not a member of private group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan1,
             Administered, User1, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group1
                            [reqack ne MEMBERNOTINGROUP];
    reject; stop

endproc (* Test_10: TestUCMmembersR *)


(***************************************************************************)
(*                                                                         *)
(*      Member DeRegistration                                              *)
(*                                                                         *)
(***************************************************************************)

(* Assumes that Creation, Registration and Members (list) *)
(* requests are operational. *)

(* Acceptance test : Checks member deregistration within a group (7 tests) *)
process Test_11 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Last member in non-administered group. Group is automatically deleted.*)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !DEREGISTER !NoMsg;
    gcs_ch !FromGCS !User3 !DEREGISTERED !Group4;
    gcs_ch !FromGCS !User3 !GROUPDELETED !Group4;
    success; stop

    []

    (* Last member in administered group. Group is automatically deleted. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !DEREGISTER !Encode(User3);
    gcs_ch !FromGCS !User3 !DEREGISTERED !Group4;
    gcs_ch !FromGCS !User3 !GROUPDELETED !Group4;
    success; stop
```

```
[]

(* 1st member in 2-members group *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
         NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan1 of CID);
gcs_ch !FromGCS !User4 !REGISTERED !Group4;
gcs_ch !ToGCS !User3 !Group4 !DEREGISTER !NoMsg;
gcs_ch !FromGCS !User3 !DEREGISTERED !Group4;
(* Verification *)
gcs_ch !ToGCS !User2 !Group4 !MEMBERS !NoMsg;
gcs_ch !FromGCS !User2 !MEMBERSARE(Insert(User4, Empty)) !Group4;
success; stop

[]

(* 2nd member in 2-members group *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
         NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan1 of CID);
gcs_ch !FromGCS !User4 !REGISTERED !Group4;
gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !NoMsg;
gcs_ch !FromGCS !User4 !DEREGISTERED !Group4;
(* Verification *)
gcs_ch !ToGCS !User2 !Group4 !MEMBERS !NoMsg;
gcs_ch !FromGCS !User2 !MEMBERSARE(Insert(User3, Empty)) !Group4;
success; stop

[]

(* Member in administered group, by admin *)
mgcs_ch !ToMGCS !User4 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
         Administered, User4, Opened, Private, NonModerated, Nobody);
mgcs_ch !FromMGCS !User4 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(User2, Chan4);
gcs_ch !FromGCS !User4 !REGISTERED !Group4;
gcs_ch !ToGCS !User4 !Group4 !MEMBERS !NoMsg;
gcs_ch !FromGCS !User4 !MEMBERSARE(Insert(User2, Insert(User4, Empty)))
                        !Group4;
gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !Encode(User2 of MID);
gcs_ch !FromGCS !User4 !DEREGISTERED !Group4;
(* Verification *)
gcs_ch !ToGCS !User4 !Group4 !MEMBERS !NoMsg;
gcs_ch !FromGCS !User4 !MEMBERSARE(Insert(User4, Empty)) !Group4;
success; stop

[]

(* Unknown member in public group *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
         NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !NoMsg;
gcs_ch !FromGCS !User4 !MEMBERNOTINGROUP !Group4;
success; stop
```

```
    []

    (* Unknown member in administered group, by admin *)
    mgcs_ch !ToMGCS !User4 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             Administered, User4, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User4 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !Encode(User2 of MID);
    gcs_ch !FromGCS !User4 !MEMBERNOTINGROUP !Group4;
    success; stop

endproc (* Test_11: TestUCMderegA *)


(* Rejection test : Checks member deregistration within a group (5 tests) *)
process Test_12 [mgcs_ch, gcs_ch, reject]:noexit :=

    (* Member in a group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan1 of CID);
    gcs_ch !FromGCS !User4 !REGISTERED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !DEREGISTER !NoMsg;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne DEREGISTERED];
    reject; stop

    []

    (* Last member in group. Group should be automatically deleted. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !DEREGISTER !NoMsg;
    gcs_ch !FromGCS !User3 !DEREGISTERED !Group4;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne GROUPDELETED];
    reject; stop

    []

    (* Member in administered group, by admin *)
    mgcs_ch !ToMGCS !User4 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             Administered, User4, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User4 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(User1 of MID, Chan2 of CID);
    gcs_ch !FromGCS !User4 !REGISTERED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !MEMBERS !NoMsg;
    gcs_ch !FromGCS !User4 !MEMBERSARE(Insert(User1, Insert(User4, Empty)))
                          !Group4;
    gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !Encode(User1 of MID);
    gcs_ch !FromGCS !User4 ?reqack:AckError !Group4 [reqack ne DEREGISTERED];
    reject; stop

    []

    (* Unknown member in public group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !NoMsg;
```

```
    gcs_ch !FromGCS !User4 ?reqack:AckError !Group4
                            [reqack ne MEMBERNOTINGROUP];
    reject; stop

    []

    (* Unknown member in administered group, by admin *)
    mgcs_ch !ToMGCS !User4 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             Administered, User4, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User4 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !DEREGISTER !Encode(User1 of MID);
    gcs_ch !FromGCS !User4 ?reqack:AckError !Group4
                            [reqack ne MEMBERNOTINGROUP];
    reject; stop

endproc (* Test_12: TestUCMderegR *)


(**************************************************************************)
(*                                                                        *)
(*      Multicast                                                         *)
(*                                                                        *)
(**************************************************************************)

(* Assumes that Creation and Registration requests are operational. *)

(* Acceptance test : Checks group multicast (6 tests) *)
process Test_13 [mgcs_ch, gcs_ch, out_ch, success]:noexit :=

    (* 3 members in public, non-moderated group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User4 !REGISTERED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
    out_ch !User2.Chan1 !User1 !Encode(Hello);     (* any order! *)
    out_ch !User3.Chan4 !User1 !Encode(Hello);
    out_ch !User4.Chan2 !User1 !Encode(Hello);
    gcs_ch !FromGCS !User1 !MESSAGESENT !Group4;
    success; stop

    []

    (* 3 members in public, non-moderated group. Different order *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User4 !REGISTERED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
    out_ch !User3.Chan4 !User1 !Encode(Hello);
    out_ch !User2.Chan1 !User1 !Encode(Hello);     (* any order! *)
    out_ch !User4.Chan2 !User1 !Encode(Hello);
    gcs_ch !FromGCS !User1 !MESSAGESENT !Group4;
```

```
    success; stop

    []

    (* Sender is a member of closed group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan4,
             Administered, User3, Closed, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !MULTICAST !Encode(Hello);
    out_ch !User2.Chan1 !User2 !Encode(Hello);
    out_ch !User3.Chan4 !User2 !Encode(Hello);     (* any order! *)
    gcs_ch !FromGCS !User2 !MESSAGESENT !Group4;
    success; stop

    []

    (* Sender is not member of close group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
    gcs_ch !FromGCS !User1 !MEMBERNOTINGROUP !Group4;
    success; stop

    []

    (* Sender is moderator of moderated group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
             Administered, User3, Opened, Public, Moderated , User2);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !MULTICAST !Encode(Hello);
    out_ch !User3.Chan1 !User2 !Encode(Hello);
    gcs_ch !FromGCS !User2 !MESSAGESENT !Group4;
    success; stop

    []

    (* Sender is not moderator of moderated group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
             Administered, User3, Opened, Public, Moderated , User2);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
    gcs_ch !FromGCS !User2 !ToApprove(User1, Encode(Hello)) !Group4 ;
    gcs_ch !FromGCS !User1 !SENTTOMODERATOR !Group4;
    success; stop

endproc (* Test_13: TestUCMmultA *)


(* Rejection test : Checks group multicast (6 tests) *)
process Test_14 [mgcs_ch, gcs_ch, out_ch, reject]:noexit :=

    (* 3 members in public, non-moderated group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
```

```
      gcs_ch !FromGCS !User2 !REGISTERED !Group4;
      gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan2);
      gcs_ch !FromGCS !User4 !REGISTERED !Group4;
      gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
      out_ch !User3.Chan3 !User1 !Encode(Hello);     (* any order! *)
      out_ch !User2.Chan1 !User1 !Encode(Hello);
      out_ch !User4.Chan2 !User1 !Encode(Hello);
      gcs_ch !FromGCS !User1 ?reqack:AckError !Group4 [reqack ne MESSAGESENT];
      reject; stop

      []

      (* MESSAGESENT ack before messages are sent (or lost of a message). *)
      mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
               NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
      mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
      gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
      gcs_ch !FromGCS !User1 !MESSAGESENT !Group4;
      reject; stop

      []

      (* Sender is a member of closed group *)
      mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
               Administered, User3, Closed, Public, NonModerated, Nobody);
      mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
      gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
      gcs_ch !FromGCS !User2 !REGISTERED !Group4;
      gcs_ch !ToGCS !User2 !Group4 !MULTICAST !Encode(Hello);
      gcs_ch !FromGCS !User2 !MEMBERNOTINGROUP !Group4;
      reject; stop

      []

      (* Sender is not member of closed group *)
      mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
               Administered, User3, Closed, Public, NonModerated, Nobody);
      mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
      gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
      out_ch !User3.Chan3 !User1 !Encode(Hello);     (* any order! *)
      gcs_ch !FromGCS !User1 !MESSAGESENT !Group4;
      reject; stop

      []

      (* Sender is moderator of moderated group *)
      mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
               Administered, User3, Opened, Public, Moderated, User2);
      mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
      gcs_ch !ToGCS !User2 !Group4 !MULTICAST !Encode(Hello);
      gcs_ch !FromGCS !User2 !ToApprove(User2, Encode(Hello)) !Group4 ;
      gcs_ch !FromGCS !User2 !SENTTOMODERATOR !Group4;
      reject; stop

      []

      (* Sender is not moderator of moderated group *)
      mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
               Administered, User3, Opened, Public, Moderated , User2);
```

```
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !MULTICAST !Encode(Hello);
    out_ch !User3.Chan1 !User1 !Encode(Hello);
    gcs_ch !FromGCS !User1 !MESSAGESENT !Group4;
    reject; stop

endproc (* Test_14: TestUCMmultR *)



(***************************************************************************)
(*                                                                         *)
(*      Group Deletion                                                     *)
(*                                                                         *)
(***************************************************************************)

(* Assumes that Creation, List (of groups), Registration, and *)
(* Multicast requests are operational. *)

(* Acceptance test : Checks deletion of groups (6 tests) *)
process Test_15 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Deletes last group in a list of groups *)
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan3,
             Administered, User2, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    gcs_ch !ToGCS !User2 !Group2 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 !GROUPDELETED !Group2;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(NoGCS);
    success; stop

    []

    (* Admin deletes first group in a list of two administered groups *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             Administered, User1, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    gcs_ch !ToGCS !User1 !Group1 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User1 !GROUPDELETED !Group1;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(Insert(Group2, NoGCS));
    success; stop

    []

    (* Admin deletes second group in a list of two administered groups *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             Administered, User1, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan3,
             Administered, User2, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    gcs_ch !ToGCS !User2 !Group2 !DELETEGROUP !NoMsg;
```

```
    gcs_ch !FromGCS !User2 !GROUPDELETED !Group2;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(Insert(Group1, NoGCS));
    success; stop

    []

    (* Non-admin tries to delete an administered group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             Administered, User1, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 !NOTADMIN !Group1;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(Insert(Group1, NoGCS));
    success; stop

    []

    (* Someone tries to delete a non-administered group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 !NOADMINGROUP !Group1;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(Insert(Group1, NoGCS));
    success; stop

    []

    (* Indicates to all members that their group was deleted *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Video, Chan1,
             Administered, User1, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User4 !Group4 !REGISTER !Encode(Chan2);
    gcs_ch !FromGCS !User4 !REGISTERED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !DELETEGROUP !NoMsg;
    (* LIFO order here... Clients should however run in parallel, thus *)
    (* fixing part of this testing problem. *)
    gcs_ch !FromGCS !User4 !GROUPWASDELETED !Group4;
    gcs_ch !FromGCS !User2 !GROUPWASDELETED !Group4;
    gcs_ch !FromGCS !User3 !GROUPWASDELETED !Group4;
    gcs_ch !FromGCS !User1 !GROUPDELETED !Group4;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(NoGCS);
    success; stop

endproc (* Test_15: TestUCMdeletionA *)


(* Rejection test : Checks deletion of groups (5 tests) *)
process Test_16 [mgcs_ch, gcs_ch, reject]:noexit :=
```

```

    (* Deletes non-existing group *)
    gcs_ch !ToGCS !User1 ?anyGroup:GID !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User1 !GROUPDELETED ?anyGroup:GID;
    reject; stop

    []

    (* Deletes a non-administered group *)
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    gcs_ch !ToGCS !User2 !Group2 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group2 [reqack ne NOADMINGROUP];
    reject; stop

    []

    (* Admin deletes an administered group *)
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan3,
             Administered, User2, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    gcs_ch !ToGCS !User2 !Group2 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group2 [reqack ne GROUPDELETED];
    reject; stop

    []

    (* Non-admin tries to delete an administered group *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             Administered, User1, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User2 !Group1 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group1 [reqack ne NOTADMIN];
    reject; stop

    []

    (* Member not advised that his group was deleted *)
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group4 !Encode(Video, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User1 !Group4 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User1 !GROUPDELETED !Group4;
    reject; stop

endproc (* Test_16: TestUCMdeletionR *)


(****************************************************************************)
(*                                                                          *)
(*      Administrator Changing                                              *)
(*                                                                          *)
(****************************************************************************)

(* Assumes that Creation and Registration requests are operational. *)

(* Acceptance test : Check the change of admin properties (5 tests) *)
process Test_17 [mgcs_ch, gcs_ch, success]:noexit :=
```

```
(* Non-administered group *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
         NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
gcs_ch !FromGCS !User3 !NOADMINGROUP !Group4;
success; stop

[]

(* Administered group, not by admin *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
         Administered, User3, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User2 !Group4 !CHANGEADMIN !Encode(User2);
gcs_ch !FromGCS !User2 !NOTADMIN !Group4;
success; stop

[]

(* Administered group, by admin, with new admin not in group *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
         Administered, User3, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
gcs_ch !FromGCS !User3 !MEMBERNOTINGROUP !Group4;
success; stop

[]

(* Administered group, by admin, with new admin in group *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
         Administered, User3, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
gcs_ch !FromGCS !User2 !REGISTERED !Group4;
gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
gcs_ch !FromGCS !User3 !ADMINCHANGED !Group4;
(* Verification *)
gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
gcs_ch !FromGCS !User2 !ATTRIBUTESARE(Encode(Mail, Chan3,Administered,
                  User2, Opened, Public, NonModerated, Nobody)) !Group4;
success; stop

[]

(* Change from administered group to non-administered, by admin. *)
mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
         Administered, User3, Opened, Public, NonModerated, Nobody);
mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(Nobody);
gcs_ch !FromGCS !User3 !ADMINCHANGED !Group4;
(* Verification *)
gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3,NonAdministered,
                  Nobody, Opened, Public, NonModerated, Nobody)) !Group4;
success; stop
```

```
endproc (* Test_17: TestUCMadminA *)


(* Rejection test : Checks the change of admin properties (6 tests) *)
process Test_18 [mgcs_ch, gcs_ch, reject]:noexit :=

    (* Non administered group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne NOADMINGROUP];
    reject; stop

    []

    (* Administered group, not by admin *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEADMIN !Encode(User2);
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4 [reqack ne NOTADMIN];
    reject; stop

    []

    (* Administered group, by admin, with new admin not in group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
                             [reqack ne MEMBERNOTINGROUP];
    reject; stop

    []

    (* Administered group, by admin, with new admin in group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne ADMINCHANGED];
    reject; stop

    []

    (* Administered group, by admin, with new admin in group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(User2);
    gcs_ch !FromGCS !User3 !ADMINCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
```

```
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4
           [reqack ne ATTRIBUTESARE(Encode(Mail, Chan3, Administered, User2,
                                    Opened, Public, NonModerated, Nobody))];
    reject; stop

    []

    (* Change from administered group no non-administered, by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEADMIN !Encode(Nobody);
    gcs_ch !FromGCS !User3 !ADMINCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
           [reqack ne ATTRIBUTESARE(Encode(Mail, Chan3, NonAdministered,
                             Nobody, Opened, Public, NonModerated, Nobody))];
    reject; stop

endproc (* Test_18: TestUCMadminR *)


(***************************************************************************)
(*                                                                         *)
(*      Moderator Changing                                                 *)
(*                                                                         *)
(***************************************************************************)

(* Assumes that Creation, Registration and Multicast requests are *)
(* operational. *)

(* Acceptance test : Checks the change of moderator properties (9 tests) *)
process Test_19 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Non-moderated and group, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 !NOMODERGROUP !Group4;
    success; stop

    []

    (* Moderated group, not by moderator (nor admin) *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User1);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User2 !NOTMODER !Group4;
    success; stop

    []

    (* Closed moderated group, by moderator, with new moderator not in group*)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Closed, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
```

```
        gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
        gcs_ch !FromGCS !User3 !MEMBERNOTINGROUP !Group4;
        success; stop

        []

        (* Closed administered and moderated group, by admin, with new *)
        (* moderator not in group *)
        mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
                 Administered, User3, Closed, Public, Moderated, User1);
        mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
        gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
        gcs_ch !FromGCS !User3 !MEMBERNOTINGROUP !Group4;
        success; stop

        []

        (* Closed moderated group, by moderator, with new moderator in group *)
        mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Video, Chan3,
                 NonAdministered, Nobody, Closed, Public, Moderated, User1);
        mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
        gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
        gcs_ch !FromGCS !User2 !REGISTERED !Group4;
        gcs_ch !ToGCS !User1 !Group4 !CHANGEMODER !Encode(User2, Moderated);
        gcs_ch !FromGCS !User1 !MODERCHANGED !Group4;
        (* Verification *)
        gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
        gcs_ch !FromGCS !User2 !ATTRIBUTESARE(Encode(Video, Chan3, NonAdministered,
                          Nobody, Closed, Public, Moderated, User2)) !Group4;
        success; stop

        []

        (* Opened moderated group, by moderator, with new moderator not in *)
        (* group *)
        mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
                 NonAdministered, Nobody, Opened, Public, Moderated, User1);
        mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
        gcs_ch !ToGCS !User1 !Group4 !CHANGEMODER !Encode(User2, Moderated);
        gcs_ch !FromGCS !User1 !MODERCHANGED !Group4;
        (* Verification *)
        gcs_ch !ToGCS !User2 !Group4 !GETATTRIBUTES !NoMsg;
        gcs_ch !FromGCS !User2 !ATTRIBUTESARE(Encode(Mail, Chan3, NonAdministered,
                          Nobody, Opened, Public, Moderated, User2)) !Group4;
        success; stop

        []

        (* Change from non-moderated (opened and administered) group to *)
        (* moderated group, by admin *)
        mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
                 Administered, User3, Opened, Public, NonModerated, Nobody);
        mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
        gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
        gcs_ch !FromGCS !User3 !MODERCHANGED !Group4;
        (* Verification *)
        gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
        gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3, Administered,
                          User3, Opened, Public, Moderated, User2)) !Group4;
```

```
    success; stop

    []

    (* Change from moderated group to non-moderated, by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
             Administered, User3, Opened, Public, Moderated, User2);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(Nobody, NonModerated);
    gcs_ch !FromGCS !User3 !MODERCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan1, Administered,
                      User3, Opened, Public, NonModerated, Nobody)) !Group4;
    success; stop

    []

    (* If the group becomes Non-moderated, insert Nobody as new moderator. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan1,
             Administered, User3, Opened, Public, Moderated, User2);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User3, NonModerated);
    gcs_ch !FromGCS !User3 !MODERCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan1, Administered,
                      User3, Opened, Public, NonModerated, Nobody)) !Group4;
    success; stop

endproc (* Test_19: TestUCMmoderA *)


(* Rejection test : Checks the change of admin properties (7 tests) *)
process Test_20 [mgcs_ch, gcs_ch, reject]:noexit :=

    (* Non-moderated and group, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne NOMODERGROUP];
    reject; stop

    []

    (* Moderated group, not by moderator (nor admin) *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4 [reqack ne NOTMODER];
    reject; stop

    []

    (* Closed moderated group, by moderator, with new moderator not in group*)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Closed, Public, Moderated, User3);
```

```
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
                              [reqack ne MEMBERNOTINGROUP];
    reject; stop

    []

    (* Closed administered and moderated group, by admin, with new *)
    (* moderator not in group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Public, Moderated, User1);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
                              [reqack ne MEMBERNOTINGROUP];
    reject; stop

    []

    (* Closed moderated group, by moderator, with new moderator in group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Closed, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !REGISTER !Encode(Chan1);
    gcs_ch !FromGCS !User2 !REGISTERED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne MODERCHANGED];
    reject; stop

    []

    (* Opened moderated group, by moderator, with new moderator not in *)
    (* group *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne MODERCHANGED];
    reject; stop

    []

    (* Change from non-moderated (opened and administered) group to *)
    (* moderated group, by admin *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan2,
             Administered, User3, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEMODER !Encode(User2, Moderated);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne MODERCHANGED];
    reject; stop

endproc (* Test_20: TestUCMmoderR *)


(***************************************************************************)
(*                                                                         *)
(*      Opened Attribute Changing                                          *)
(*                                                                         *)
```

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

(* Assumes that Creation and GetAttributes are operational. *)

(* Acceptance test : Checks the change of the Opened attribute (4 tests) *)
process Test_21 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Administered group, request by admin. From Closed to Opened *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEOPENATTR !Encode(Opened);
    gcs_ch !FromGCS !User3 !OPENATTRCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3, Administered,
                       User3, Opened, Private, NonModerated, Nobody)) !Group4;
    success; stop

    []

    (* Administered group, request by admin. From Opened to Closed *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEOPENATTR !Encode(Closed);
    gcs_ch !FromGCS !User3 !OPENATTRCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3, Administered,
                       User3, Closed, Private, NonModerated, Nobody)) !Group4;
    success; stop

    []

    (* Non-administered group, we cannot change this attribute *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEOPENATTR !Encode(Closed);
    gcs_ch !FromGCS !User3 !NOADMINGROUP !Group4;
    success; stop

    []

    (* Administered, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEOPENATTR !Encode(Closed);
    gcs_ch !FromGCS !User2 !NOTADMIN !Group4;
    success; stop

endproc (* Test_21: TestUCMopenA *)


(* Rejection test : Checks the change of the Opened attribute (3 tests) *)
process Test_22 [mgcs_ch, gcs_ch, reject]:noexit :=

```

```
    (* Administered group, request by admin *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEOPENATTR !Encode(Opened);
    gcs_ch !FromGCS !User3 !OPENATTRCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
            [reqack ne ATTRIBUTESARE(Encode(Mail, Chan3, Administered, User3,
                                    Opened, Private, NonModerated, Nobody))];
    reject; stop

    []

    (* Non-administered group, we cannot change this attribute *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEOPENATTR !Encode(Closed);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne NOADMINGROUP];
    reject; stop

    []

    (* Administered, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEOPENATTR !Encode(Closed);
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4 [reqack ne NOTADMIN];
    reject; stop

endproc (* Test_22: TestUCMopenR *)


(*****************************************************************************)
(*                                                                           *)
(*      Private Attribute Changing                                           *)
(*                                                                           *)
(*****************************************************************************)

(* Assumes that Creation and GetAttributes are operational. *)

(* Acceptance test : Checks the change of the Private attribute (4 tests) *)
process Test_23 [mgcs_ch, gcs_ch, success]:noexit :=

    (* Administered group, request by admin. From Private to Public. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEPRIVATTR !Encode(Public);
    gcs_ch !FromGCS !User3 !PRIVATTRCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3, Administered,
                       User3, Closed, Public, NonModerated, Nobody)) !Group4;
    success; stop
```

```
    []

    (* Administered group, request by admin. From Public to Private. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEPRIVATTR !Encode(Private);
    gcs_ch !FromGCS !User3 !PRIVATTRCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 !ATTRIBUTESARE(Encode(Mail, Chan3, Administered,
                        User3, Closed, Private, NonModerated, Nobody)) !Group4;
    success; stop

    []

    (* Non-administered group, we cannot change this attribute *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEPRIVATTR !Encode(Private);
    gcs_ch !FromGCS !User3 !NOADMINGROUP !Group4;
    success; stop

    []

    (* Administered, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEPRIVATTR !Encode(Private);
    gcs_ch !FromGCS !User2 !NOTADMIN !Group4;
    success; stop

endproc (* Test_23: TestUCMprivA *)


(* Rejection test : Checks the change of the Opened attribute (3 tests) *)
process Test_24 [mgcs_ch, gcs_ch, reject]:noexit :=

    (* Administered group, request by admin *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Closed, Private, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User3 !Group4 !CHANGEPRIVATTR !Encode(Public);
    gcs_ch !FromGCS !User3 !PRIVATTRCHANGED !Group4;
    (* Verification *)
    gcs_ch !ToGCS !User3 !Group4 !GETATTRIBUTES !NoMsg;
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4
            [reqack ne ATTRIBUTESARE(Encode(Mail, Chan3, Administered, User3,
                                     Closed, Public, NonModerated, Nobody))];
    reject; stop

    []

    (* Non-administered group, we cannot change this attribute *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
```

```
    gcs_ch !ToGCS !User3 !Group4 !CHANGEPRIVATTR !Encode(Private);
    gcs_ch !FromGCS !User3 ?reqack:AckError !Group4 [reqack ne NOADMINGROUP];
    reject; stop

    []

    (* Administered, not by admin. *)
    mgcs_ch !ToMGCS !User3 !CREATEGROUP !Group4 !Encode(Mail, Chan3,
             Administered, User3, Opened, Public, Moderated, User3);
    mgcs_ch !FromMGCS !User3 !GROUPCREATED !Group4;
    gcs_ch !ToGCS !User2 !Group4 !CHANGEPRIVATTR !Encode(Private);
    gcs_ch !FromGCS !User2 ?reqack:AckError !Group4 [reqack ne NOTADMIN];
    reject; stop

endproc (* Test_24: TestUCMprivR *)


(*============================================*)
(*          Simple Client Test Case           *)
(*============================================*)

(* Test case from the client viewpoint. *)
(* Tests the refusal of requests to unknown groups by the server *)
(* The client needs a timer to detect such problems and react *)
(* accordingly. *)

process Test_25[gcs_ch, success] : exit :=
    hide reject, timeout in
    (* Any request to Group4, which does not exist *)
        gcs_ch !ToGCS !User2 !Group4 ?anyreq:Request !NoMsg;
        reject; exit (* We should not be able to get here *)
        [>
        timeout;     (* timeout should be the only action possible *)
        success; stop
endproc (* Test_25: TestClientA *)


(*============================================*)
(* Complex Test Case with Pre/Post Conditions *)
(*============================================*)

(* This is a more generic format for test processes. We first bring the *)
(* system from the initial state to a specific state that satisfies a *)
(* pre-condition. Then, we execute the scenario, and finally we check *)
(* the scenario post-condition. In this example, the scenario tests *)
(* that we can register a second member to a group that contains already *)
(* one member. *)


process Test_26[mgcs_ch, gcs_ch, out_ch, success] : noexit :=

  hide reject in
  (* Preamble *)
  (* Gets the system to a state where a group contains only its creator *)
  (
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             NonAdministered, Nobody, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    gcs_ch !ToGCS !User1 !Group1 !REGISTER !Encode(Chan2);
```

```
      gcs_ch !FromGCS !User1 !REGISTERED !Group1;
      exit
    )
    >>
    (* Check pre-condition : *)
    (* (Group1 IsIn GroupList) AND (MemberList(Group1)={User1}) *)
    (
      gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
      (
          gcs_ch !FromGCS !User2 !MEMBERSARE(Insert(User1, Empty)) !Group1;
          exit
          []
          gcs_ch !FromGCS !User2 ?reqack:AckError !Group1
                                   [reqack ne MEMBERSARE(Insert(User1, Empty))];
          reject; stop (* Pre-condition not satisfied *)
      )
    )
    >>
    (* Check scenario: Register of a second user in a group (User3 in Group1) *)
    (
      gcs_ch !ToGCS !User3 !Group1 !REGISTER !Encode(Chan2);
      gcs_ch !FromGCS !User3 !REGISTERED !Group1;
      exit
    )
    >>
    (* Check post-condition: *)
    (* (Group1 IsIn GroupList) AND (MemberList(Group1)={User1, User3}) *)
    (
      gcs_ch !ToGCS !User2 !Group1 !MEMBERS !NoMsg;
      (
          gcs_ch !FromGCS !User2 !MEMBERSARE(Insert(User3, Insert(User1, Empty)))
                                  !Group1;
          success; stop
          []
          gcs_ch !FromGCS !User2 ?reqack:AckError !Group1
                      [reqack ne MEMBERSARE(Insert(User3,Insert(User1, Empty)))];
          reject; stop (* Post-condition not satisfied *)
      )
    )

  endproc (* Test_26: Test_Complex *)

  endspec (* GCS, Group Communication Server *)
```

Appendix B Example of Use of TMDL

B.1 Description of Group Creation in TMDL

The following TMDL description presents the Group Creation functionality, as interpreted from the corresponding UCM. Additional internal actions were introduced to synchronize the two paths (`SpawnSignal`) and to simulate a team instantiation (`Instantiate-GCSTeam`). TMDL descriptions are generated manually.

```
{-------------------------------------------------}
{ GCS Services Test Cases - TMDL Examples         }
{ Daniel Amyot, University of Ottawa              }
{ Version 0.1                                     }
{ January 7, 1997                                 }
{-------------------------------------------------}

Map GCSTestCases is

INTERNAL
    SpawnSignal, InstanciateGCSteam

INTERACTIONS
    GroupCreation, SpawnNewGroup on SpawnSignal;

DESCRIPTIONS

Timethread GroupCreation is

   Internal
      CheckDB,          { Check an abstract (and virtual!) database }
      AddNewIdInGCSlist

   Trigger (CreateGroup)
   Wait (CheckDB ? Exist)

   OrFork
       Guard (Exist eq IsNotInDB)
       Continue  { The new group does not already exist }
     OR
       Guard (Exist eq IsInDB)
       { The new group already exists! }
       Path
          Result (GroupExists)  { Error message. Exit here }
       EndPath
   EndOrFork

   Sync (SpawnSignal)
   Action (AddNewIdInGCSlist)
   Result (GroupCreated) { Acknowledge }
EndTT { GroupCreation }

Timethread SpawnNewGroup is
   Trigger (SpawnSignal)
   Result (InstanciateGCSteam)
EndTT { SpawnNewGroup }

EndMap { GCSTestCases }
```

B.2 Corresponding LOTOS Specification

We automatically generated this specification (`GCSTestCases`) using the *tmdl2lot* compiler [Amyot 1994a], a tool that translates TMDL descriptions into equivalent LOTOS specifications. Events are mapped onto gates and tags onto a single data type.

```
(* TMDL-to-LOTOS Compiler, version 0.9. *)

specification GCSTestCases[CreateGroup, GroupCreated, GroupExists]:noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition *)

type Tag is Boolean, NaturalNumber
sorts Tag
opns dummy_val, IsInDB, IsNotInDB : -> Tag
     N : Tag -> Nat
     _eq_, _ne_ : Tag, Tag ->Bool
eqns forall x, y: Tag
     ofsort Nat
     N(dummy_val)= 0;   (* dummy value *)
     N(IsInDB) = Succ(N(dummy_val));
     N(IsNotInDB) = Succ(N(IsInDB));
     ofsort Bool
     x eq y = N(x) eq N(y);
     x ne y = not(x eq y);
endtype

behaviour

hide InstanciateGCSteam, SpawnSignal in

   (
      GroupCreation[CreateGroup, GroupCreated, GroupExists, SpawnSignal]
      |[SpawnSignal]|
      SpawnNewGroup[InstanciateGCSteam, SpawnSignal]
   )

where

   process GroupCreation[CreateGroup, GroupCreated, GroupExists, SpawnSignal]:noexit :=

   hide AddNewIdInGCSlist, CheckDB in

      CreateGroup;
      (
         CheckDB ? Exist:Tag;
         (
            (
               [Exist eq IsInDB]->
               GroupExists; stop  (* No recursion *)
            )
            []
            (
            [Exist eq IsNotInDB]->
               SpawnSignal;
               AddNewIdInGCSlist;
```

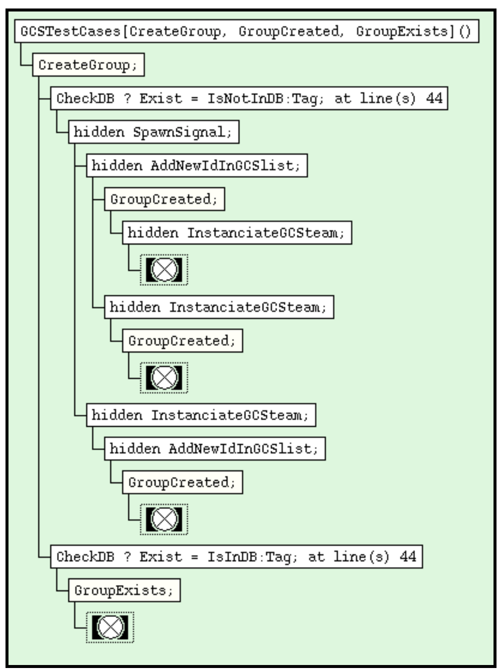
```
               GroupCreated; stop  (* No recursion *)
            )
         )
      )
   endproc  (* Timethread GroupCreation *)

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

   process SpawnNewGroup[InstanciateGCSteam, SpawnSignal]:noexit :=
      SpawnSignal;
      (
         InstanciateGCSteam; stop  (* No recursion *)
      )
   endproc  (* Timethread SpawnNewGroup *)

endspec (* Map GCSTestCases *)
```

B.3 Expansion of this Specification

This specification being somewhat simple and short, we can expand its complete behaviour using tools such as *SELA* (part of the *XEludo* toolkit) or LOLA. Figure 34 illustrates by a tree all the alternative global executions of `GCSTestCases`. We obtained this figure with *XEludo* using step-by-step execution of all possible behaviours.

As expected, a **GROUPCREATED** message results when the group identifier is not already in the database. While a new GCS team is created, the new identifier is added to the database and the acknowledgement is sent back. When the identifier is in the database, a **GROUPEXISTS** error results.

FIGURE 34. Exhaustive Simulation of GCSTestCases with XEludo



Appendix C Other Multicast Processes

We present three alternative Multicast processes that could substitute the one in our specification. The two first ones (*Sequential* and *Best Effort Sequential*) can be *plugged-in* as is in the specification. The third one (*Broadcast*) would require some modifications in the specification as it has a rather different approach.

C.1 Sequential Multicast

This protocol sends the message to the receivers in a sequential way (LIFO order). The process in the specification created different concurrent threads for each sending.

```
(**************************************************************************)
(*                                                                        *)
(*      MulticastS (Sequential)                                           *)
(*                                                                        *)
(*      Send the message to all subscribers of the group (sequentially).  *)
(*      No other message will be multicast until the first one is sent to *)
(*      all members in the group.                                         *)
(*                                                                        *)
(**************************************************************************)

process MulticastS[out](sender:MID, msg:Msg, mbrL:MemberList, UseChannel:Bool): exit :=

    [mbrL eq NoMBR] ->
         exit

    []

    [mbrL ne NoMBR] ->
        (
            [UseChannel] ->
                (* Multicasts message to members on their appropriate *)
                (* data channel, sequentially *)
                out !Top(mbrL) !sender !msg;
                    (* loop... *)
                    Multicast[out](sender, msg, Tail(mbrL), UseChannel)
            []
            [Not(UseChannel)] ->
                 (* Use request/ack channel, sequentially (for group deletion) *)
                 out !FromGCS !MID(Top(mbrl)) !GetAck(msg);
                    (* loop... *)
                    Multicast[out](sender, msg, Tail(mbrL), UseChannel)
        )

endproc (* MulticastS *)
```

C.2 Best Effort Sequential Multicast

This protocol sends the messages in a sequential way (as in C.1), but it counts the number of messages that have been successfully sent (or even received, if we consider the sending to be a synchronous interaction between the system and a receiver). A time-out mechanism ensures that the system does not block while sending a message. This process could be enhanced such that it would retry to send a number of times before declaring a failure.

```
(*************************************************************************)
(*                                                                       *)
(*      MulticastE (Best Effort)                                         *)
(*                                                                       *)
(*      Send the message to all subscribers of the group (sequentially). *)
(*      Best effort, without retry. Could be extended to include retries.*)
(*      We count the number of Pass and the number of Fail.              *)
(*                                                                       *)
(*************************************************************************)

process MulticastE[out](sender:MID, msg:Msg, mbrL:MemberList, UseChannel:Bool): exit :=

    hide P, F in (* Counter gates: P for Pass, F for Fail *)
        (
            BestMulticast[out, P, F](sender, msg, mbrL, UseChannel)
            |[P, F]|
            Counter[P, F](0 of Nat, 0 of Nat)
        )
        >>
        accept pass:Nat, fail:Nat in
            exit (* “pass” and “fail” could be made available to the calling process *)

    where

    process BestMulticast[out, P, F](sender:MID, msg:Msg, mbrL:MemberList, UseChannel: Bool)
                                : exit(Nat, Nat):=

        hide TimeOut in

        [mbrL eq NoMBR] ->
             exit (any Nat, any Nat) (* for synchronisation with Counter *)
        []
        [mbrL ne NoMBR] ->
            (
                [UseChannel] ->
                    (* Multicasts message to members on their appropriate *)
                    (* data channel, sequentially *)
                    (
                        out !Top(mbrL) !sender !msg;
                            P; (* Successfully sent! We loop... *)
                                BestMulticast[out, P, F](sender, msg, Tail(mbrL), UseChannel)
                        []
                        Timeout;
                            F; (* Problem while sending; continue with next *)
                                BestMulticast[out, P, F](sender, msg, Tail(mbrL), UseChannel)
                    )
                []
                [Not(UseChannel)] ->
                    (
                        (* Use request/ack channel *)
                        out !FromGCS !MID(Top(mbrl)) !GetAck(msg);
                            P; (* Successfully sent! We loop... *)
                                BestMulticast[out, P, F](sender, msg, Tail(mbrL), UseChannel)
                        []
                        Timeout;
                            F; (* Problem while sending; continue with next *)
                                BestMulticast[out, P, F](sender, msg, Tail(mbrL), UseChannel)
                    )
             )
    endproc (* BestMulticast *)
```

```

    process Counter[P, F](pass:Nat, fail:Nat) : exit (Nat, Nat) :=

        P; Counter[P, F](succ(pass), fail) (* one more Pass *)
        []
        F; Counter[P, F](pass, succ(fail)) (* one more fail *)
        []
        exit(pass, fail)                   (* final results *)

    endproc (* Counter *)

endproc (* MulticastE *)
```

C.3 Broadcast

This protocol assumes an underlying broadcast mechanism (such as IP broadcast) to be used for sending messages. Using the group identifier sent with every message, each receiver would need to filter the messages that are relevant from the others.

```
(***************************************************************************)
(*                                                                         *)
(*      MulticastB (Broadcast)                                             *)
(*                                                                         *)
(*      Send the message to all subscribers of the group using an          *)
(*      underlying multicast or broadcast mechanism (such as IP broadcast) *)
(*      Receiver’s channels and identifiers are irrelevant in this         *)
(*      situation. They know, by looking at the group identifier, whether  *)
(*      the message is addressed to them or not. Each receiver is          *)
(*      responsible for keeping a list of group identifiers in which it    *)
(*      has registered. Encryption mechanisms could be added for security  *)
(*      and privacy.                                                       *)
(*                                                                         *)
(*      We removed the notification of group deletion from this            *)
(*      hypothetical multicast process.                                    *)
(*                                                                         *)
(***************************************************************************)

process MulticastB[out](gid:GID, sender:MID, msg:Msg)
                  : exit :=

        (* Broadcast message to group *)
        out !gid !sender !msg;
            exit

endproc (* MulticastB *)
```

Appendix D Example of MSC Generation

In Section 6.2.2 on page 46, we introduced the notion of strict event structure in order to enhance the traceability between execution traces (or trees) and messages exchanged between components. We present here an overview of an application of such information. Our intention is to give an idea, using a simple example, of how Message Sequence Charts (MSCs) can be obtained from LOTOS traces and a description of the topology of components. A MSC will be derived from a test process, optionally considering the users as contextual information instead as message parameters. Then, we will consider the architectural information of the GCS in order to decompose the MSC and look at internal messages. Dynamic creation of processes will also be considered.

Readers have to be careful with the semantics associated to the following MSCs. In the standard, messages are asynchronous, while LOTOS descriptions are based on a synchronous interactions mechanism. Therefore, we assume here that messages are synchronous, i.e., the transmission takes no time, there are no message queues, and messages between two given components cannot cross each other.

D.1 Generation of Traces from the Specification

Traces can be generated from LOTOS specifications in many ways. These methods usually include step-by-step execution, simulation, random walk, expansion, goal-oriented execution, and testing. Traces can be used for problem diagnostics or for documentation.

In the following example, we chose to extract a trace from the composition of a test case and the main behavior of the specification. LOLA has a functionality, called *OneExpand*, that allows the user to generate a random trace from the specification, optionally constrained by a composition with a test. For instance, we chose to use *Test_15* (acceptance test for a group deletion), which includes a test case that checks whether or not the system successfully deletes the second and last group from its list.

```
process Test_15 [mgcs_ch, gcs_ch, success]:noexit :=
...
    (* Admin deletes second group in a list of two administered groups *)
    mgcs_ch !ToMGCS !User1 !CREATEGROUP !Group1 !Encode(Mail, Chan3,
             Administered, User1, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User1 !GROUPCREATED !Group1;
    mgcs_ch !ToMGCS !User2 !CREATEGROUP !Group2 !Encode(Video, Chan3,
             Administered, User2, Opened, Public, NonModerated, Nobody);
    mgcs_ch !FromMGCS !User2 !GROUPCREATED !Group2;
    gcs_ch !ToGCS !User2 !Group2 !DELETEGROUP !NoMsg;
    gcs_ch !FromGCS !User2 !GROUPDELETED !Group2;
    (* Verification *)
    mgcs_ch !ToMGCS !User3 !GROUPS;
    mgcs_ch !FromMGCS !User3 !GROUPSARE(Insert(Group1, NoGCS));
    success; stop
...
endproc (* Test_15: TestUCMdeletionA *)
```

We used "*OneExpand -1 Success Test_15 2 1 -v -i*" in order to get the following trace, where internal actions are commented:

```
1    mgcs_ch  ! tomgcs ! user1 ! creategroup ! group1 !
               encode(mail,chan3,administered,user1,opened,public,nonmoderated,nobody);
2    i; (* sgcs_ch ! creategroup ! group1 ! insert(user1 . chan3,nombr) !
               encode(mail,chan3,administered,user1,opened,public,nonmoderated,nobody) *)
3    mgcs_ch  ! frommgcs ! user1 ! groupcreated ! group1;
4    mgcs_ch  ! tomgcs ! user2 ! creategroup ! group2 !
               encode(video,chan3,administered,user2,opened,public,nonmoderated,nobody);
5    i; (* sgcs_ch ! creategroup ! group2 ! insert(user2 . chan3,nombr) !
               encode(video,chan3,administered,user2,opened,public,nonmoderated,nobody) *)
6    mgcs_ch  ! frommgcs ! user2 ! groupcreated ! group2;
7    gcs_ch  ! togcs ! user2 ! group2 ! deletegroup ! nomsg;
8    i; (* inter_ch ! togcs ! user2 ! deletegroup ! nomsg *)
9    i; (* exit *)
10   i; (* inter_ch ! fromgcs ! user2 ! groupdeleted *)
11   i; (* agcs_ch ! groupdeleted ! group2 *)
12   gcs_ch  ! fromgcs ! user2 ! groupdeleted ! group2;
13   mgcs_ch  ! tomgcs ! user3 ! groups;
14   i; (* inter_ch ! togcs ! groupdeleted *)
15   mgcs_ch  ! frommgcs ! user3 ! groupsare(insert(group1,nogcs));
16   success;
```

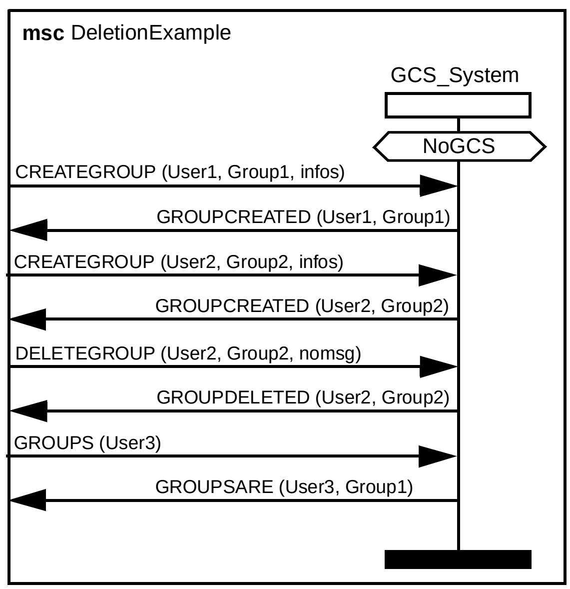
Our goal is to show how this trace would look like as a MSC. Note that we are not necessarily constrained to traces; because MSCs now include an *alternative* operator in its notation, full LTSs (trees with choices and sequences) could also be represented.

D.2 MSC from a Test Process

First, we can represent the test process itself as a MSC. The test acts as the *environment* of the system. Since we know the direction of the messages within the channels (Section 6.2.2), we can straightforwardly generate the MSC in Figure 35. We replaced the information parameter (`Encode(...)`) by `infos` in order to simplify the diagram.

FIGURE 35. MSC DeletionExample as Derived from the Test Process

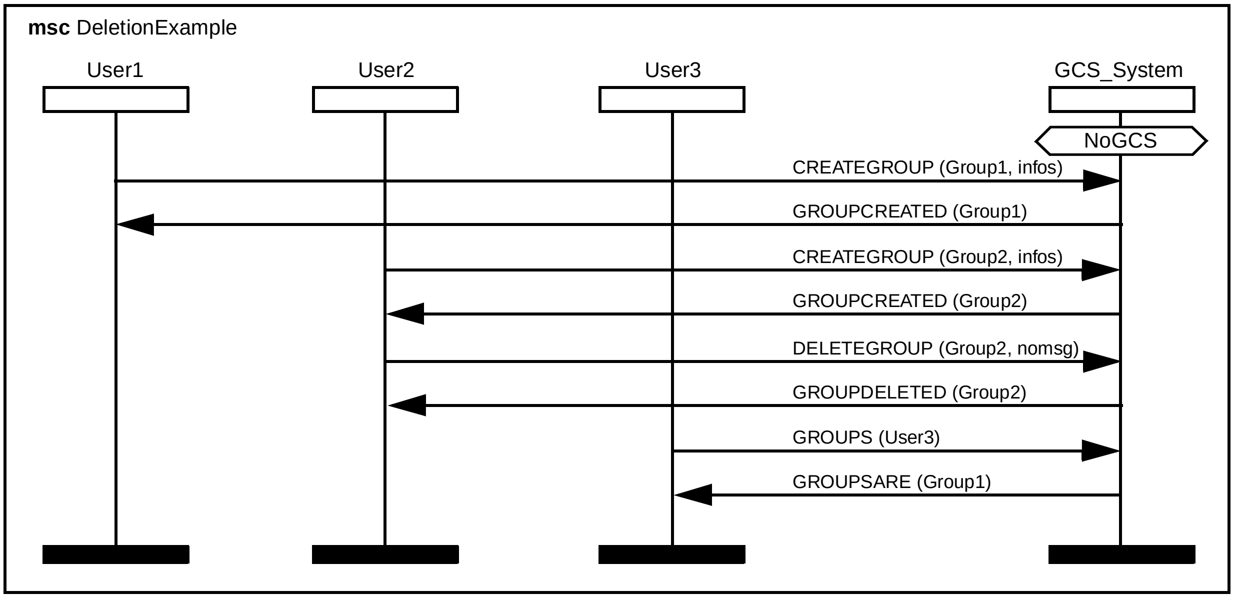
```
msc DeletionExample; inst GCS_System;

    instance GCS_System;
        condition NoGCS shared;
       in CREATEGROUP (User1, Group1, infos) from env;
        out GROUPCREATED (User1, Group1) to env;
       in CREATEGROUP (User2, Group2, infos) from env;
        out GROUPCREATED (User2, Group2) to env;
       in DELETEGROUP (User2, Group2, nomsg) from env;
        out GROUPDELETED (User2, Group2) to env;
        in GROUPS (User3) from env;
        out GROUPSARE (User3, Group1) to env;
    endinstance

endmsc;
```



Users could optionally be represented as instances of some client process. This would lead to a more complex MSC where all component instances, including those involved in the tester process, are represented (see Figure 36). However, the sender identifier would not need to be stated as a parameter, because we now have a visual representation of the users. Gates such as `mgcs_ch` and `gcs_ch` could also be explicitly used to further refine the MSC representation. As for the textual form, we always have a choice between a MSC description that is component-oriented (instances defined separately; this is what we use here) or system-oriented (references to instances are attached to each message) [ITU, 1996].

FIGURE 36. MSC DeletionExample with Users Explicitely Represented

```
msc DeletionExample;
inst GCS_System, User1, User2, User3;

    instance GCS_System;
        condition NoGCS shared;
        in CREATEGROUP (Group1, infos) from User1;
        out GROUPCREATED (Group1) to User1;
        in CREATEGROUP (Group2, infos) from User2;
        out GROUPCREATED (Group2) to User2;
        in DELETEGROUP (Group2, nomsg) from User2;
        out GROUPDELETED (Group2) to User2;
        in GROUPS from User3;
        out GROUPSARE (Group1)to User3;
    endinstance;

    instance User1;
        condition NoGCS shared;
        out CREATEGROUP (Group1, infos) to GCS_System;
        in GROUPCREATED (Group1) from GCS_System;
    endinstance;

    instance User2;
        condition NoGCS shared;
        out CREATEGROUP (Group2, infos) to GCS_System;
        in GROUPCREATED (Group2) from GCS_System;
        out DELETEGROUP (Group2, nomsg) to GCS_System;
        in GROUPDELETED (Group2) from GCS_System;
    endinstance;

    instance User3;
        condition NoGCS shared;
        out GROUPS to GCS_System;
        in GROUPSARE (Group1) from GCS_System;
    endinstance;

endmsc;
```



D.3 MSC from an Execution Trace and the Structure

Components often include sub-components, as we can observe from the GCS structure (Figure 11 on page 25). The designer may decide that the messages exchanged between those sub-components are hidden to the external world. LOTOS traces reflect this fact with internal actions (**i**). In the traces generated with LOLA, we can see the internal actions and their associated message, in comments. We can use this information to describe MSCs at various levels of abstraction related to the structure of the components.

First Level: GCS_System

We will use the decomposition functionality of MSCs to describe these levels of abstractions. At the top (system) level, the MSC we get from our trace is exactly the same as the one we generated from the test process (Figure 35). There would be however a minor difference at line 2 of the textual description: we need to have the "**decomposed**" keyword at the end, stating that the instance *GCS_System* is to be refined further in another MSC.

Second Level: Control_Team & GCS_Team (within GCS_System)

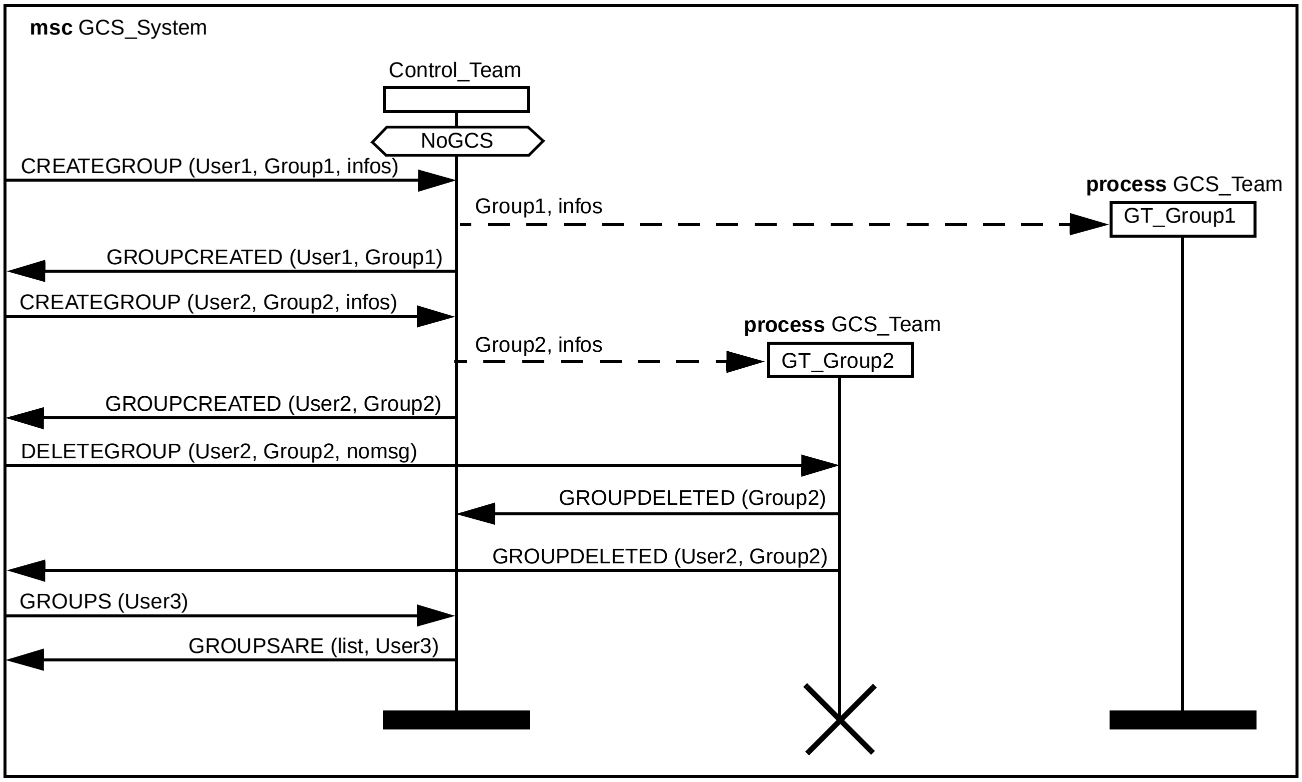
The second level of abstraction (with respect to the structure) would allow us to look at *GCS_System* from an internal viewpoint, i.e., the MSC is to be refined in terms of two classes of components: *Control_Team* and *GCS_Team* (multiple instances). Figure 37 shows this MSC. There are two interesting aspects to this figure:

- The messages to/from the environment correspond to the ones we had in the first MSC. There is only one new internal message between *Control_Team* and the second *Group_Team* (named *GT_Group2*).

- Dynamic creation/destruction of process instances is part of the MSC notation, and we use it for the management of GCS teams. A dotted arrow indicates the creation of an instance, and a “X” its termination.

FIGURE 37. MSC GCS_System

```
msc GCS_System;
inst Control_Team decomposed,
     GT_Group1: process GCS_Team decomposed,
     GT_Group2: process GCS_Team decomposed;

    instance Control_Team decomposed;
        condition NoGCS shared;
        in CREATEGROUP (User1, Group1, infos) from env;
        create GT_Group1 (Group1, infos);
        out GROUPCREATED (User1, Group1) to env;
        in CREATEGROUP (User2, Group2, infos) from env;
        create GT_Group2 (Group2, infos);
        out GROUPCREATED (User2, Group2) to env;
        in GROUPDELETED (Group2) from GT_Group2;
        in GROUPS (User3) from env;
        out GROUPSARE (list, User3) to env;
    endinstance;

    instance GT_Group1: process GCS_Team decomposed;
    endinstance;

    instance GT_Group2: process GCS_Team decomposed;
        in DELETEGROUP (User2, Group2, nomsg) from env;
        out GROUPDELETED (Group2) to Control_Team;
        out GROUPDELETED (User2, Group2) to env;
        stop;
    endinstance;

endmsc;
```

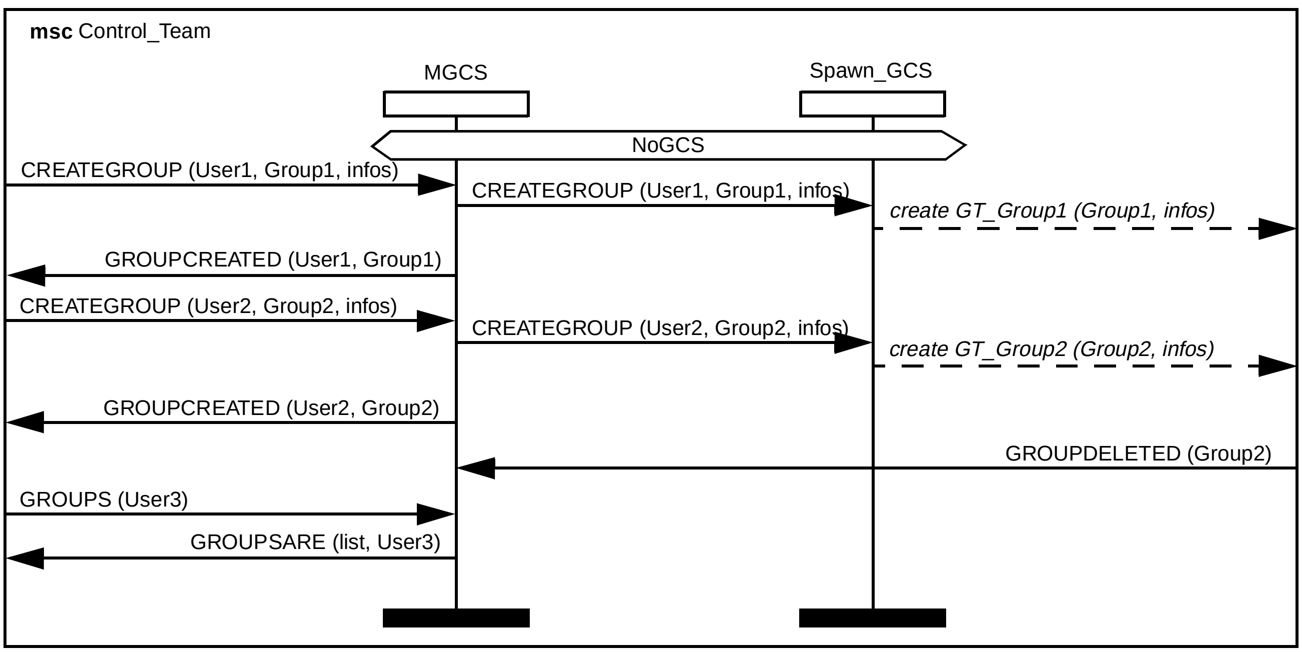


Third Level: MGCS & Spawn_GCS (within Control_Team)

The three instances from the previous figure can be further decomposed. We know that *Control_Team* is composed of two communicating entities: *MGCS* and *Spawn_GCS*. Figure 38 shows the corresponding MSC. A *GCS_Team* contains two sub-components: a *BiDiBuffer* and a *GCS*. Figure 39 and Figure 40 respectively show the MSCs of our two GCS teams, *GT_Group1* (empty in our case) and *GT_Group2*. Several internal exchanges of messages between the sub-components are shown. They correspond to the internal events in the LOTOS trace.

FIGURE 38. MSC Control_Team

```
mcs Control_Team; inst MGCS, Spawn_GCS;

    instance MGCS;
        condition NoGCS shared all;
        in CREATEGROUP (User1, Group1, infos) from env;
        out CREATEGROUP (User1, Group1, infos) to Spawn_GCS;
        out GROUPCREATED (User1, Group1) to env;
        in CREATEGROUP (User2, Group2, infos) from env;
        out CREATEGROUP (User2, Group2, infos) to Spawn_GCS;
        out GROUPCREATED (User2, Group2) to env;
        in GROUPDELETED (Group2) from env;
        in GROUPS (User3) from env;
        out GROUPSARE (list, User3) to env;
    endinstance;

    instance Spawn_GCS;
        condition NoGCS shared all;
        in CREATEGROUP (User1, Group1, infos) from MGCS;
        create GT_Group1 (Group1, infos);
        in CREATEGROUP (User2, Group2, infos) from MGCS;
        create GT_Group2 (Group2, infos);
    endinstance;

endmsc;
```



Third Level: BiDirBuffer & GCS (within GCS_Team)

FIGURE 39. MSC GT_Group1

```
msc GT_Group1;
inst Buffer_Group1: process BiDiBuffer, GCS_Group1: process GCS;

    instance Buffer_Group1: process BiDiBuffer;
    endinstance;

    instance GCS_Group1: process GCS;
    endinstance;

endmsc;
```

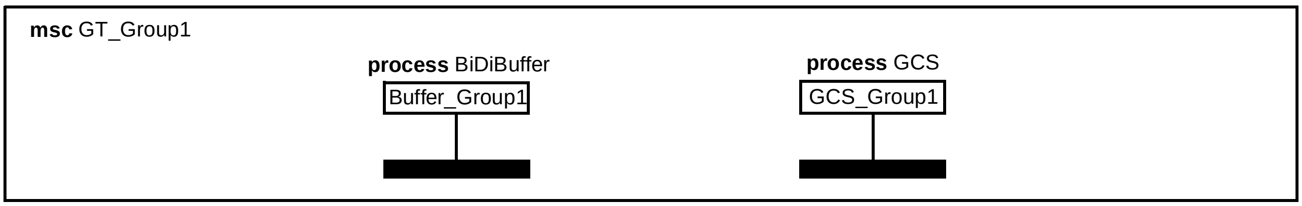
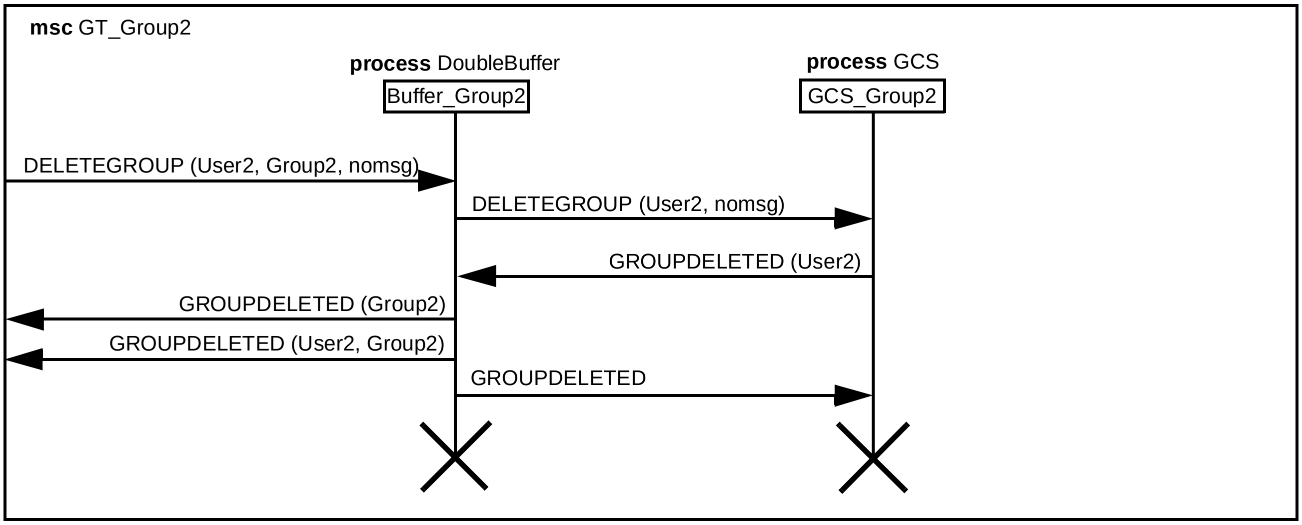


FIGURE 40. MSC GT_Group2

```
msc GT_Group2;
inst Buffer_Group2: process BiDiBuffer, GCS_Group2: process GCS;

    instance Buffer_Group2: process BiDiBuffer;
        in DELETEGROUP (User2, Group2, nomsg) from env;
        out DELETEGROUP (User2, nomsg) to GCS_Group2;
        in GROUPDELETED (User2) from GCS_Group2;
        out GROUPDELETED (Group2) to env;
        out GROUPDELETED (User2, Group2) to env;
        out GROUPDELETED to GCS_Group2;
        stop;
    endinstance;

    instance GCS_Group2: process GCS;
        in DELETEGROUP (User2, nomsg) from Buffer_Group2;
        out GROUPDELETED (User2) to Buffer_Group2;
        in GROUPDELETED from Buffer_Group2;
        stop;
    endinstance;

endmsc;
```



Flattened MSC

Instead of having multiple local views, resulting from the levels of abstraction, one could decide to *flatten* the structure and provide a global transparent view of the system. In that case, we would see all the sub-components in one MSC. However, we believe that two many components shown at once clutters the understanding. Nevertheless, when the structure is relatively simple, a flattened view might provide better insights in the understanding of an execution trace.

Event Oriented Textual Representations

The MSC textual notation allows two styles of description. The one we used in our example is component oriented, and it describes a scenario in terms of the messages sent and received by system components. There is another style that seems closer to the execution traces and the UCM. This textual format describes an end-to-end scenario with messages that explicitly state the sender in addition the receiver. This is another viewpoint that fits well in an approach based on UCMs.

Note that both styles are equivalent and we can go from one to the other automatically.

D.4 MSC Usage

In our approach, MSCs can be used for testing and documentation:

- **Testing**: Execution traces can be visualized with MSCs. This graphical notation is certainly more appealing than a plain textual trace. Moreover, we can better understand the relationships between the components for a specific scenario, because of our access and integration of architectural information. Viewing executions is useful when unexpected scenarios or test results occur. We can then compare the test MSC with a problematic execution MSC to visually diagnose where and why the problem occurred. A view where instances (e.g., users, as in Figure 36) are explicitly represented seems best suited for representing these MSCs.
- **Documentation**: Typical executions can be derived from the model and use as part of the system documentation, for illustrative purpose. Hierarchical views, based on the architecture, can help focusing on component behaviour.

Note that synchronous MSCs based on LOTOS traces, such as the one we used here, seem very useful for requirements engineering, but perhaps not so much for detailed design. Using LOTOS, we tend not to address all the issues at once. Having realistic channels with delays and losses/reordering of messages increases significantly the complexity of system validation. These issues can be addressed at a later time, in the detailed design, with a LOTOS model that includes buffer processes (to simulate realistic asynchronous channels) leading to true MSC semantics.