RUN-TIME MONITORING AND ON-LINE TESTING OF MIDDLEWARE BASED COMMUNICATION SERVICES

THÈSE N^{O} 2137 (2000)

PRÉSENTÉE AU DÉPARTEMENT DE SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Xavier LOGEAN

Ingénieur électricien diplômé EPF
de nationalité suisse et originaire d'Hérémence (VS)

acceptée sur proposition du jury:

Prof. J.-P. Hubaux, directeur de thèse
M. S. Grisouard, rapporteur
Dr Y.-J. Lin, rapporteur
Prof. A. Schiper, rapporteur

Lausanne, EPFL
2000

à mon père,
à ma mère,
à Tit

Acknowledgements

Were it not for the aid, support and encouragement of many people, this thesis would not be what it is. I would therefore like to formally recognize some of those who have helped me along the way, towards the realization of this goal.

My deepest thanks are to my advisor Professor Jean-Pierre Hubaux who welcomed me in his laboratory. I especially appreciated his constant concern for bringing research face to face with the needs of the industry, the large academic freedom that was a source of motivation, and his valuable support.

I would like to express a deep gratitude to Professor Simon Znaty and Dr. Shawn Koppenhoefer for their contribution to this work.

I would like to thank Falk Dietrich with whom I had the privilege to work with on the same project, as well as in the same office. His availability, good humor, constructive criticism, suggestions and above all his facility to solve problems have been of great help. The positive atmosphere of our office contributed to motivate me when necessary.

Nothing could have been done without the advice of Dr. Pierre-Alain Etique, who introduced me to the problematic of validating and testing communication services. I thank him for his support and for all the fruitful discussions we had.

I am especially grateful to all the members of ICA, and particularly towards Raffaele Noro, Daniel Lungu, Holly Cogliati and Danielle Alvarez.

Sylvain Grisouard, Thierry Labbé and Stéphane Betgé-Brezetz from Alcatel Research, deserve recognition for all the discussions we had on applying my research in their industrial environment.

I thank Swisscom AG, Switzerland, and Alcatel Research, France, for providing the funds for my research.

Last but definitely not least, I acknowledge my parents for their continuous confidence and invaluable support. I express a special thank to my sister Christine and her husband Dan who always helped me with sage advice, from the other side of the world. Finally, my gratitude goes to my wife Séverine who always encouraged me by her confidence and her understanding. This work owes a lot to her presence and love.

Abstract

With the present day's exponential growth of the (tele-)communications market, the explosion of the number of mobile communication customers, and the tremendous growth of the number of IP hosts, ensuring the reliability of communication services is one of the most challenging tasks in today's software engineering.

This Thesis addresses this problem by providing a new method for increasing confidence in the implementation of communication services. We developed an approach for *monitoring* and *testing* communications services that are built on top of a middleware. Specifically, we show how to monitor a communication service and how to use the monitored information to test at runtime whether the behavior of the service conforms to constraints. The constraints are extracted from the requirements, expressed using a formal method. The approach advocated in this Thesis consists in combining the power of formal methods, which have reached a high degree of maturity in the academic domain, with a testing methodology, that fit the industrial needs, in order to test the implementation of a communication service. The developer can thus *focus on the testing rather then on how the testing is performed.*

To solve this problem (1) we designed a *monitoring* mechanism that is able to observe at run-time, the events occuring in the system under scrutiny, (2) designed a *testing* mechanism that checks at runtime if properties expressed using Temporal Logic, are violated by the execution of the application, (3) implemented a prototype *tool*, MOTEL (MOnitoring and TEsting tooL).

With our approach the tester need only to specify the properties. The other steps involved in the testing processes are performed automatically. According to the content of the properties the instrumentation is added in the system to be tested. This instrumentation reports a consistent order of the events occurring to a central Observer. Within this Observer, the properties, which are independent of the implementation language, are translated into deterministic automata. At run-time these automata are used, according to the events reported by the instrumentation to the Observer, to check if the properties are violated. Thus, our monitoring and testing mechanism fits well in the development process because it does not add noticeable new steps for the developer.

In this thesis we put together many concepts and solutions from various domains (distributed systems, middleware, object-oriented development, CORBA, communication services, formal method, Temporal Logic, automata theory, monitoring, testing). By enhancing, developing and combining, and integrating them together, we reached a high level of integration that provides a new, powerful and useful functionality to test the implementation of communication services.

We believe that there is not a more valuable argument for the application and practical relevance of our approach in the industry, than the introduction and use of our method and tool in and by the industry. Our partner, Alcatel, is currently integrating MOTEL in the distributed platform PERCO.

Version Abrégée

Dû à l'évolution exponentielle du marché des télécommunications, à l'explosion du nombre de clients de la communication mobile et à l'incroyable évolution du nombre d'hôtes IP, garantir la fiabilité des services de communications est un des grands défis actuels du génie logiciel appliqué aux télécommunications.

Cette Thèse propose une nouvelle approche pour améliorer la qualité de l'implantation des services de communications. Nous avons développé une méthode pour l'observation et le test des services de communications qui ont été développés à l'aide d'un "middleware". Plus spécifiquement, nous montrons comment observer un service de communications et comment utiliser cette information pour tester, durant l'exécution du service, si le comportement respecte ce qui est exprimé dans des propriétés. Les propriétés sont établies grâce au cahier des charges du service, indépendamment du langage d'implantation et à l'aide d'un langage formel.

Dans notre approche, (1) nous avons développé un mécanisme *d'observation* qui intercepte les événements durant l'exécution du service, (2) nous avons mis au point un mécanisme pour le *test*, qui vérifie si des propriétés exprimées à l'aide de la logique temporelle et de notre modèle événementiel, sont violées durant l'exécution du service, (3) nous avons implanté un *prototype*, appelé MOTEL (MOnitoring and TEsting tooL).

En appliquant notre méthode, le "testeur" se concentre sur le test plutôt que sur les moyens par lesquels le test est accompli: il exprime les propriétés à vérifier et les autres étapes impliquées dans la procédure de test sont accomplies automatiquement. Le code est instrumenté pour détecter et rapporter les événements, en respectant l'ordre d'occurrence, à un Observeur central. Au sein de l'Observeur, les propriétés sont traduites sous la forme d'automates déterministes. Durant l'exécution du service, les automates sont alimentés à l'aide des événements observés, pour vérifier si les propriétés sont violées. Notre approche s'intègre bien dans le processus de développement logiciel puisque les étapes normales restent identiques, uniquement l'expression des propriétés est ajoutée.

Dans cette thèse, nous combinons l'efficacité des méthodes formelles, qui sont matures dans le domaine académique, avec une méthode de test qui correspond aux besoins industriels; nous avons intégré les concepts et solutions de différents domaines (systèmes répartis, CORBA, développement orienté-objet, services de communications, méthodes formelles, théorie des automates, observations et test de systèmes). En les améliorant, développant, combinant et intégrant ensemble, nous avons atteint un haut niveau d'intégration qui apporte une nouvelle et puissante méthode pour tester l'implémentation des services de communications.

Nous pensons qu'il n'y a pas de meilleurs arguments pour l'application et l'intérêt pratique de notre travail dans l'industrie, que son introduction et utilisation, dans et par l'industrie. Notre partenaire, Alcatel, intègre actuellement MOTEL dans sa plate-forme répartie PERCO.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Chapter Overview

In this chapter we give a general description of the problem addressed in this thesis, the reasons for our work and we sketch the approach taken. Finally, we outline the structure of the document.

1.2 Addressed Problem

The design and implementation of telecommunication services is a complex task. This task is filled with pitfalls where errors may manifest themselves only when the final implementation is observed at run-time. Often, there are many inconspicuous symptoms that pass unperceived by the human overseer whose job is to constantly watch for imminent disaster. Entire telecommunication systems can be brought to their knees by a misimplemented piece of the greater puzzle or an unintended feature interaction. This puzzle, and thus the risk of errors, is likely to get even more complicated in the future. Services are becoming increasingly complex and are provided in a more and more heterogeneous environment.

In response to the complex task of service provision, in the eighties, the telecommunications world proposed the Intelligent Network (IN). With the IN, users were offered more sophisticated services such as call forwarding or the '800' free call service. But the solutions provided by the IN could not easily integrate with the emerging multimedia communications, nor the advancing software engineering. Therefore, a world-wide consortium, composed mainly of huge telecommunication companies and national telecommunication operators, proposed the Telecommunications Information Networking Architecture (TINA). The TINA architecture encompasses the features of the IN, the concepts of the Telecommunication Management Network (TMN), the object-oriented approach and the distributed technology. Due to the complexity of the proposal, and mainly to the emergence of Internet technologies,

the TINA consortium proposals were not welcomed by the different actors of the communication world.

Now, the trend is *hybrid services*. A hybrid service is defined as a service that spans several network technologies, especially the Public Switched Telephone Network (PSTN), the cellular networks, and the IP networks. An example of a simple hybrid service is Click-To-Dial, which enables a user to request, from a Web browser, a connection to be set up between telephones connected to the PSTN.

With installations that will soon reach one billion wireline phones, 300 million mobile phones and 200 million IP hosts, assuring hybrid service reliability is one of the most challenging tasks in software engineering. Many academics see formality (formal specifications, formal methods, formal description techniques etc.) as an inevitability for a high degree of software reliability and scalability.

After many years of research in the testing domain, there are still unsolved or unsuspected problems that may ensue situations such as the billing problem of Deutsche Telekom in January 1996[1]. This thesis aims to master one part of the obstacles on the road to providing communication services, built on top of a middleware, which behavior can be trusted by developers as well as by users.

Formal Methods do not have a high degree of acceptance in the industry. This thesis also aims to make testing based on Formal methods more appealing to the industry.

Our ultimate goal is to provide a solution that fits the needs and the reality of communication systems, and to provide a more valuable than purely "academic" solution. We propose an approach for *monitoring* and *testing* distributed applications that are built on top of a middleware. Specifically, we show how to monitor a communication service and how to use the monitored information to test at runtime whether the behavior of the service conforms to constraints. The constraints are extracted from the requirements, expressed using a Formal Technique.

As we are interested in a solution/method that corresponds to the needs of the industry, our collaboration and interactions with Swisscom and with Alcatel Alsthom Research Marcoussis (France), are of great importance. As a result, our ideas, that led to an invention that we patented [LDH+99b], are implemented within the PERCO [LBGD99] platform, jointly developed by Alcatel Research and Thomson.

Although our focus is on communication services, the approach proposed can be applied for the monitoring and testing of any middleware based distributed application; this could carry implications to domains such as fault tolerance, service management, and security.

[1]The company that implemented the billing software forgot to implement January 1st as a day off (which implies a cheaper billing policy), and billed users like a working day. The financial impact of that error was a lost of 100 million DM! [Som96]

1.3 Methodology Adopted in this Work

We aim to provide methods to increase confidence in the implementation of communication services. For this, understanding the nature of the communication services is of great importance. How can a communication service be defined?; what is its structure?; what are the similarities or genericities between communication services?; what infrastructure/environment do they run on?; etc. We carefully analyze these questions to establish the background for this dissertation.

All the communication services that are offered today are distributed: some part of the software is running on the client side, while some other parts are running in one or many providers' sites. For example, in the case of a Telebanking service, the client is running a software on his computer that will ask the bank server to perform some actions on his account. Among the different Distributed system standards, CORBA (Common Object Request Broker) is the leader today.

For decades the development of software has evolved and Object-Oriented technology has proven to be useful and suitable for industrial developments. Moreover, the process of developing Object-oriented software has been formalized and many methods and notations have been developed: OMT, FUSION and recently UML (Unified Modelling Language). Combined with the Object-Oriented technology, distributed systems offer the building-blocks for the software of communication services.

A large part of the method proposed in this work is based on the object-oriented distributed environment.

The approach taken in this Thesis consists in combining the power of Formal Methods (FM), which have reached a high degree of maturity in the academic domain, with a testing methodology, that fit the industrial needs, in order to test the implementation of a communication service.

Specifically, the Formal Method that is used in this Thesis is a Temporal Logic, which allows the specification of properties on the temporal order of the states of a system. Associated with the Automata theory, Temporal Logic properties can be represented in a form that is suitable to be checked either by a model checker based on a state-space exploration or by an on-line checker.

This thesis implies the merge of several domains, as depicted in Figure 1.1.

1.4 Requirements for this Work

The work presented in this Thesis was done with the collaboration of Swisscom AG, Corporate Technology, Bern, Switzerland and Alcatel Recherche, Marcoussis, France. As this work was partially funded by these two industrial collaborations, we agreed on the following requirements:

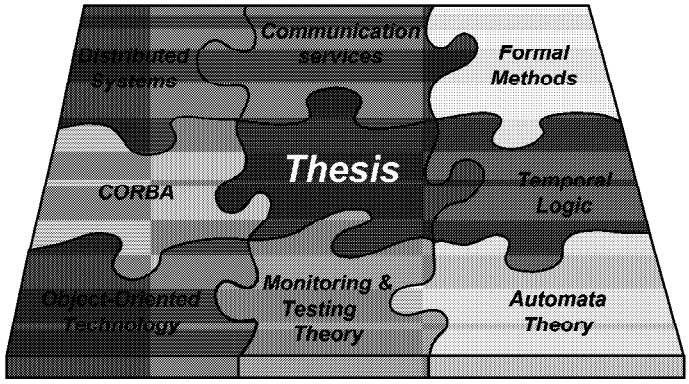


Figure 1.1: Domains involved in this Thesis

- *Applicability to industrial framework* The development of testing methods based on Formal Methods often leads to simple, demonstrative and academic examples. We intend to provide a method to fit today's industrial needs for testing communication services. For example, we must consider the scalability and the heterogeneity of the environment.

- *Applicability to all communication services* A communication service may be deployed on different network infrastructures and different middleware environments. Our goal is to provide a monitoring and a testing method that is as open as possible to avoid restrictions on its applicability to only a few environments. One of the common denominators of the communication services is the middleware. In this work, *we assume that the services are deployed by the means of a middleware.*

- *Testing of the implementation* Given a competitive market, with multiple service providers and heterogeneous equipment, it is rather unlikely that detailed specification of the services and/or the operational environment can be obtained. Therefore our testing methodology must focus on the runtime stage of the service: what has to be tested is the actual service implementation in an operational environment. With this approach, we ensure that what we test is what is (or will be) provided to the customers. The testing of an abstract "complete" specification of the service is outside the scope of our work, as it is unfeasible in today's communication services. We would like to avoid focusing our test too early in the software development process without any guarantee that no new errors or bugs will be added when we develop the final implementation. The testing methods should also permit the checking of a large scope of constraints: feature interactions, internal correct behavior, etc.

- *Integration in the software development process* In order to optimize the efforts of development and testing, our method should require minimal specific work for testing purposes. In other words, it must fit well in the software development process.

- *Provision of a tool support* As our method is intended for use by industrial developers, it is definitely relevant to provide a tool support. Typical communication system developers are neither trained in, nor comfortable with, the formality and mathematical sophistication of most FMs. Therefore, the machinery involved for testing (FMs, automata, monitoring, etc.) should be hidden from the service developer. All functionality should appear in some kind of human-friendly Graphical User Interface. The developer can thus *focus on the testing rather then on how the testing is performed.*

In addition to the points that are enumerated above, the generation of test scenarios and test cases is another important aspect in the testing process. Nevertheless, that part is left for further study.

The work we performed in order to fulfil the requirements presented above, was done in strong collaboration with Falk Dietrich. He concentrated mainly on the aspects of modelling the behavior of middleware based applications with events, whereas, in this thesis we concentrate in how to monitor and test middleware based applications. The thesis of Falk Dietrich "Modelling and Testing Object-Oriented Communication Services with Temporal Logic" [Die00] and this thesis are part of the same project, each solving one part of the problem.

1.5 Advocated Solution

Two concepts are at the heart of the solution advocated in this work, *monitoring* and *testing*. Testing is defined as [IEE97]:

> *The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspects of the system or component.*

Testing and debugging are often lumped under the same headings and often used together as "testing and debugging". Nevertheless, the purpose of testing is to show that a program is free of faults (commonly called bugs). The purpose of debugging is find error or misconception that led to the program's failure and to design and implement the program changes that correct the error.

A Monitor is defined as [IEE97]:

> *A tool that executes concurrently with another program and provides detailed information about the execution of the other program.*

Joyce et. al detail the definition of monitoring [J+87]:

> *The Monitoring of distributed systems involves the collection, interpretation, and display of information concerning the interactions among concurrently executing processes.*

Monitoring and Testing are two different activities. Monitoring consists in observing a system whereas testing consist in finding errors of a system.

In Figure 1.2 we present a comprehensive view of the approach advocated in this thesis.

We also indicate in each box the chapter in which we developed the corresponding concepts. The left side of the figure represents the usual distributed software

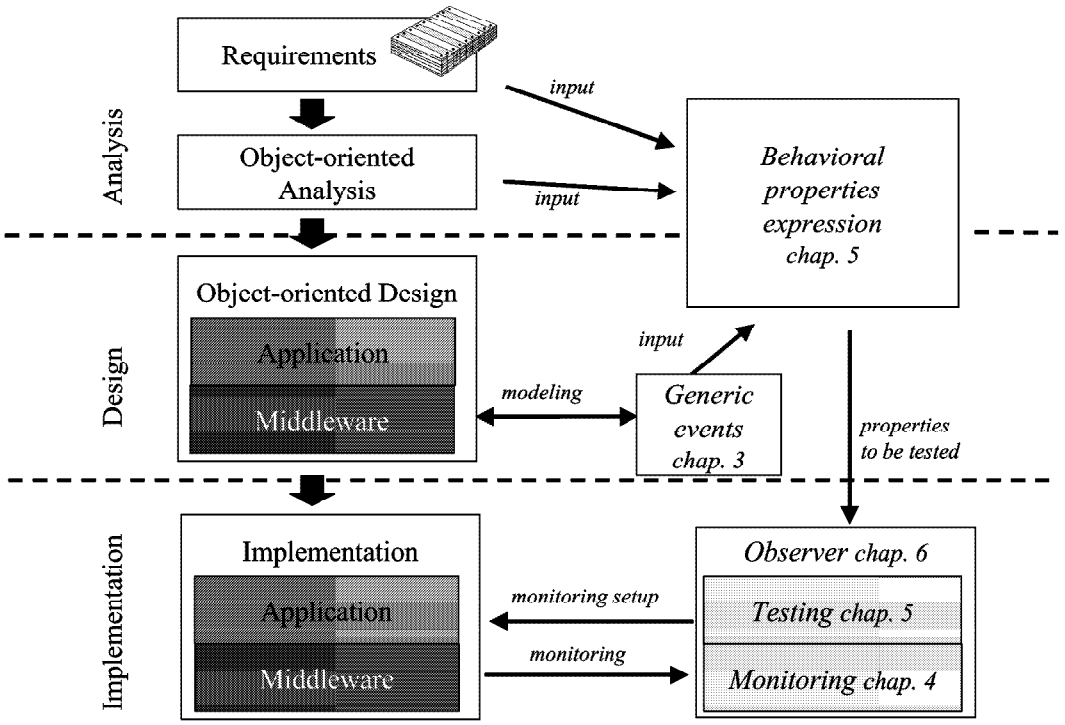


Figure 1.2: Monitoring and Testing approach

development process. From the requirements, the developer proceeds to the analysis step, usually using an object-oriented method/notation such as Fusion or UML. The following step, where the distribution issues are to be solved, is the design. Finally the implementation code of the application is written according to the analysis and the design accomplished.

The method we propose in this Thesis consists in expressing properties on an abstract level (i.e., not on the implementation code) and checking for the violation of these properties at the implementation level and during execution. For this, we first use a set of generic events that is used to model the behavior of any middleware based application. Based on the requirements, the analysis step of the application and the set of events, the developer expresses properties that represents the behavior that the application has to fulfil.

These properties are given to an Observer that is composed of two parts: a Testing part that handles the properties and checks for violations at run-time and a Monitoring part that observes the running execution and forwards the observation

to the Testing part.

The Monitoring is performed on the final implementation of the application and observes occurring events.

It is important to note that the testing mechanism is dynamic: the tester can change at run-time the properties to check. A second aspect is of great importance: the method proposed is fully automatizable and thus does not require from the tester a complete understanding of the code of the application. Therefore, our proposal is applicable even if the tester is not the developer of the application and thus, fits well in industrial practice.

1.6 Main Contributions

In this work we:

- *designed a method for the monitoring of communication services.* From our discussions with our industrial partners, a set of events that is judged adequate to model the behavior of communication services built on top of a middleware has been identified. This set of events is generic and can thus be used to model the behavior of "any" distributed system. We developed a method to catch these events at the middleware layer, during the execution of the services. The observed events are sent to an observer via notifications, for which, we designed a set of mechanisms ensuring the consistency and the pertinence of the information received at the observer site. A specific format was created for those notifications.

  In order to be able to catch the events and send notifications, a strategy to instrument the code was designed. As extra operations are added in the execution of the service, we also measured the impact of our method on the system: from a theoretical, as well as from a performance measurement, point of view.

- *designed a method for the testing of communication services.* Based on the set of events that was chosen and the requirements of the service, we express behavioral constraints using Linear Time Temporal Logic.

  In order to be used for testing, the properties have to be transformed into a form that can be interpreted by a machine. We adapted a tableau algorithm for the translation of properties into automata (Finite State Machines). To avoid some state-space explosion, special care must be put in the managing of the number of state and number of transitions of the generated automata. Thus, our translation method produces deterministic Automata and optimizes their number of states.

  The testing scenario was defined: the monitored information is to be used to update the state of each automata to reflect the current state of the system.

- *designed a method that fits in the software development.* The method we advocate in this thesis is mostly automatizable. The instrumentations that have to

be added in order to catch and send the events that occur, can be introduced automatically, without the need for the developer to look into the implementation code. This implies that the software development process is not modified. Our method focuses on constraints that the developer judges important and, thus, does not necessitate the definition of a large, time-consuming, formal specification of the entire system. Adding our testing method to the development process of the software only requires the execution of a tool that adds the instrumentation and, then, the expression of behavioral constraints by the software developer.

- *designed and implemented a method that is efficient for industrial applications.* With the goal of hiding all the internal testing-monitoring machinery to the user, we developed and implemented a prototype that is called MOTEL (MOnitoring and TEsting tooL). With it, the user can concentrate on the expression of properties and the rest is automatically performed for him.

  While our tool is primarily designed for testing, it can also be used for monitoring only, thus serving as a pseudo-debugger for distributed applications. To handle the typical large size of industrial systems, our tool scales well with the number of properties.

- *designed a method that is independent of the implementation language and platform.* Due to the evolution of the communication service market and because there is no longer only one standard for the provision of services, such as the Intelligent Network in the eighties, we developed a method that is open. The method is not restricted to a specific middleware or a specific implementation language.

1.7 Structure of the Document

In Figure 1.3 the structure of this document is summarized.

In Chapter 2, we present the basic concepts needed for this work. We define a communication service and its role, and we present different distributed system standards. We focus on CORBA but also briefly present DCOM, DCE and JAVA RMI. A section is devoted to a state of the art of the use of middlewares in the (tele-)communication world.

In Chapter 3, we introduce the notion of event behavioral abstraction that leads to the definition of generic events to model the behavior of distributed system. The defined events are the basics for the monitoring process. This Chapter has been written in tight collaboration with Falk Dietrich.

Chapter 4 is devoted to the Monitoring of distributed Systems. We first present the monitoring concepts and definitions. Then our proposal for monitoring is detailed: architecture, mechanism, performance, etc. At the end of the Chapter, we present the related work on Monitoring.

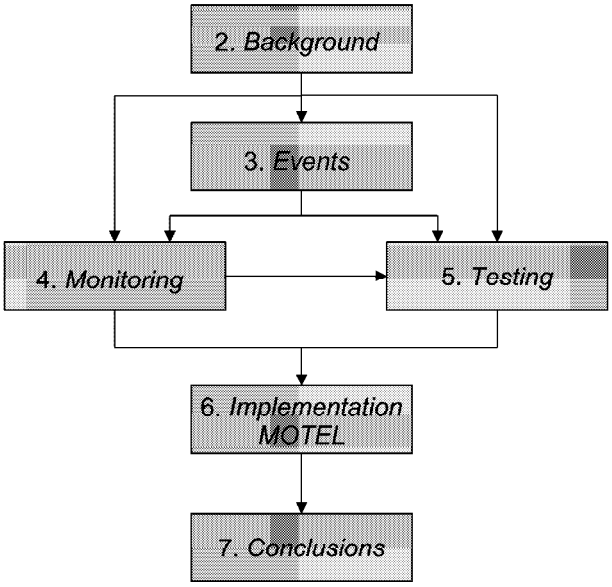


Figure 1.3: Structure of the document

In Chapter 5 the testing mechanisms are presented. The expression of the properties is presented, followed by the way the properties are represented to be used for on-line testing. The different testing processes and scenarios are explained. At the end of the Chapter, we present the related work on Testing.

Chapter 6 is dedicated to the presentation of the prototype tool that we implemented, which is called MOTEL (MOnitoring and TEsting tooL). The architecture and working of the prototype is analyzed and some examples of use are given. We also explain how MOTEL can be used in and by industry and how it is being introduced in the industrial platform PERCO.

We conclude this work in Chapter 7, some other approaches to continue this work are also proposed.

Chapter 2

Framework and Background

Parts of this Chapter were published in [LDH+99a]

2.1 Chapter Overview

In this chapter we present the framework of this thesis and the necessary background. We first define what are *services* and *communication services*, and what is *service engineering*; then we set the focus of this work. We define on which platform services are deployed and we present the concept of middleware and some standards and implementations. We introduce CORBA, and some products on the market. We also present several projects in the area of communication systems, making use of distributed platforms.

2.2 Introduction

The creation, validation, testing and maintenance of communications service is a complex task that can be summarized by the term of *service engineering*. In the framework of service engineering, we are interested in the monitoring and testing of services: How can we test if a service is behaving as expected? For this, two concepts are absolutely essential: what is a service? and how is it provided?

The word *service* has many meanings, depending on the domains in which it is used. In Section 2.3 we define a service and we differentiate telecommunication, communication, teleinformation and hybrid services. In this thesis, we are mainly interested in communication services. In Section 2.4, we clearly set the focus of this work.

As the services that are offered today are becoming increasingly complex, they require the use of the advances in software engineering -especially Object-Oriented technology and distributed systems. The intrinsic nature of a service is distributed: there is always one provider and one consumer. Todays, distributed softwares are

built on top of distributed platforms that are commonly called *middlewares*. In Section 2.5 we present different middlewares that are used for the provision of services. We focus on CORBA and some of its implementations. CORBA is a standard and at the time these lines are written the leader in the distributed processing environments.

To illustrate the use of middlewares in industry, and particularly in the telecommunication companies, we present, in Section 2.7, several industrial projects involving the use of middlewares.

As already explained in Chapter 1, in this thesis we design a method for *monitoring* and *testing* communication services. The related work on monitoring can be found in Chapter 4; the related work on testing in Chapter 5.

2.3 Service & Service Engineering

The word *service* is one of the words that have numerous meanings in the communication domains as well as in the computer science domains. The goal of this section is to examine the different meanings and define, in our context, what we mean by a *communication service*, which is at the heart of our approach.

2.3.1 Computer Science

In the computer science domain, a service is often defined as a set of features on which applications are run. For example in the OSI reference model, a layer N, is offering a service to layer $N+1$ via a Service Access Point.

2.3.2 Internet

In the "Internet Domain", as defined by the IETF, a service is usually related to the transport function provided by a network. For example, the word service is used in the resource reservation domain. One speaks about Resource Reservation Protocol (RSVP), differentiated services, etc.

2.3.3 Telecommunication Service

In the ISDN (Integrated Services Digital Network) world, three types of services are defined:

1. *Support services*: They define the characteristics of the lowest layer of the OSI model. They correspond to the "bearer capability".

2. *Teleservices*: They are applications built on top of the support services. They define layers 4 to 7 of the OSI model. Telephony, audio, fax, video telephony are some of the teleservices foreseen in narrow-band ISDN.

3. *Supplementary services*: Sometimes called features, they complement one of the teleservices or support services. Most well known supplementary services are related to the telephony teleservices (Call Forwarding, Call Screening, Credit Card Calling, etc.)

Sometimes, one speaks about *value-added services*. This concept generally corresponds to anything added to a teleservice and that can be used to earn money. The term *Supplementary services* can be added in this definition, but it is often used to refer to services offered through the network, like telebanking. In this case, the network is simply used as an access medium, and only its capacity to establish a connection between two access points is used.

The ISDN refers to teleservice and, to some extent to support services. The Intelligent Network addresses the problem of supplementary services.

2.3.4 Teleinformation Services

In the context of communications, which is the context of this work, we define a service as :

A contractual relationship between two organizations: the service provider and the service client; in this relationship, the provider commits to provide facilities, based on communication networks, with a given level of Quality of Service (QoS)

A service can be of two kinds:

- *Communication services*: The information exchanged between the users are generated by a terminal and sent to another, or to several, terminal(s),
- *Information services* : The information is stored in a server and accessed by a terminal.

A *Teleinformation service* is thus the association of terminal(s), a contract on the QoS, a connectivity and, a distributed environment and/or a networking mechanism

For example, end-to-end voice, Virtual Private Network (VPN), Telebanking, E-commerce, credit card calling, etc. are teleinformation services.

2.3.5 Hybrid Services

A *hybrid service* is defined as a service that spans many network technologies [GHHT99], especially the Public Switched Telephone Network (PSTN), the cellular networks and the IP networks. An example of a simple hybrid service is Click-To-Dial, which enables a user to request, from a Web browser, a connection to be set up between telephones connected to the PSTN. Another example is Voice over IP networks. The subscribers can use the regular (called "black") phone to call somebody but the call is routed through an IP network instead of a connection-oriented network. Note that both networks might be involved in the call delivery.

The ITU-T H.323 recommendation was adopted to guide the provisioning of hybrid services and the related functionality [GHHT99, TO99].

2.3.6 Service Engineering

Service Engineering is the art of providing new teleinformation services over communications networks. It encompasses the following tasks:

- Service Definition: how can a service provider and an end-user have a common understanding of the service?
- Service Creation: how to create a new service?
- Service Testing: how to test new services, how to detect and solve service interactions?
- Quality of Service: how to guarantee a chosen QoS for the end-user as well as within the network?
- Trust: how can the trust between end-user and between service providers be defined and ensured?
- Billing: how to define appropriate billing models?
- Service Management: how to organize the procedure and the data related to the end-users and to the services? how to track the problem and assess customer satisfaction?

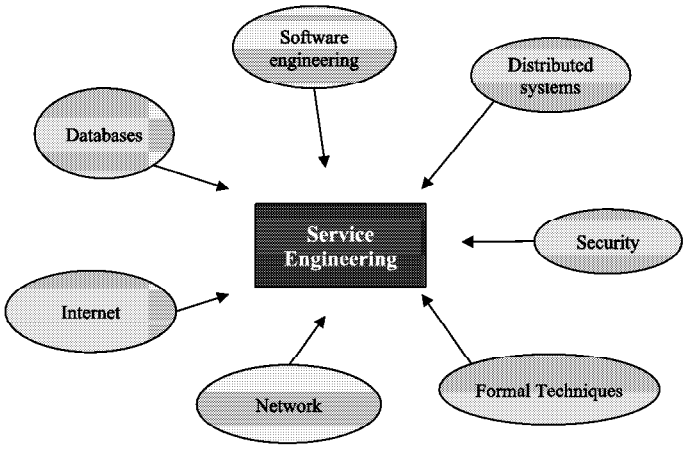


Figure 2.1: Service Engineering

As depicted in Figure 2.1, service engineering is at the intersection of many domains: software engineering, distributed systems, the Internet, network, etc. and thus necessitates multidisciplinary competences.

2.4 Focus and Position of this work

Among the different tasks of services engineering, this Thesis addresses *the testing of teleinformation services and, in particular communication services.*

Formerly, services were provided by the national operator(s) of each country. Since the privatisation of most of these operators and the subsequent market competitions, the time-to-market for the introduction of new services has decreased immensely. There is also a challenge to create new services that will attract customers with interesting and useful functions. With these requirements, there is a need for new methods for the creation of services. Since this topic is relatively new, there has been little work in the domain of testing the "new" services but much on how to create them (new architectures [AFFH96][Gba99], new components [Jub98] , etc.).

Consequently, two industrial partners, Swisscom and Alcatel, were interested in the area of monitoring and testing communication services. Therefore, we have investigated this area, trying to provide a method that fits the industry's needs.

Even with the most liberal meaning of formal methods, it is safe to say that formal methods are rarely used in the industry [Gri96]. Academics see formal methods as inevitable in the future of the software profession but practitioner see formal methods as "irrelevant" to what they do [Gla96]. As formal methods can bring interesting concepts (possibility of verification and validation, unambiguity of specifications, etc.), in this thesis, we try to make Formal Methods more appealing to industry and for that, we develop a method for monitoring and testing middleware based communication services, that:

- *Applies to industrial frameworks.* Often, methods involving the use of Formal Methods are only applied to simple academic such as the dining philosopher.
- *Applies to a broad number of services, middleware platforms and implementation languages.*The method should be as generic as possible. It can be applied to a large number of services, platforms and implementation languages; thus the effort of development amortises.
- *Tests the implementation.* The method should contribute to the quality of the actual implementation; proving the correctness of highly abstract models is not desirable.
- *Hides the formalism.* Usually industrial developers are not familiar with Formal Methods, by hiding the formalism the developer can focus to the test rather on how the testing is performed.
- *Integrates in the software development process.* The time-consuming development of large formal specifications should be avoided. Formal methods should be used as add-ons in the normal development process; if desired, they can be gradually introduced.
- *Provides a tool support.* We developed a prototype tool MOTEL that hides much of the formalism and automatizes several steps of testing.

The interested reader can find a survey of the use of Formal Methods for communications services in [Die00]. In [Die00], there is also an interesting discussion on the application of Formal Methods according to the peculiarities of communications services.

We position our work and present the related works at the end of Chapter 4 and 5. In order to set the basis for our work we present in the next sections the different middlewares that can be used for the provision of communication services and their use in the industry.

2.5 Middleware

Distributed applications are built with the help of Distributed Processing Environment (DPE), or in other words, with *middlewares*. With such environments, the specifier of the service does not need to deal with distribution issues when specifying the distributed application. The RPC mechanism or sophisticated socket mechanisms that ensure distributed features of the application are completely hidden from the programmer. Thus, the design of the distributed application consists of:

1. identifying the objects required for the application
2. defining the interfaces to those objects
3. implementing the different interfaces
4. writing a server program that creates instances of the implementation classes
5. writing a client program that accesses the server objects.

There are different middlewares, the following are widespread:

Sockets Sockets define a mechanism for the client to find and pass data to the server, and also for the server to listen to a specific point for the client requests. Sockets are the basic technology for making client-server applications.

RPC and DCE The Open Software Foundation (OSF) has defined the Distributed Computing Environment (DCE). DCE provides some "services" such as time services, RPC (Remote Procedure Call), security, threads, etc. DCE defines an API to remote function calls from a client to a server running on a remote host. In [Gba95, GGM95], a Virtual Private Network is designed as a distributed application by the means of DCE. Note that DCE does not encompass the object-oriented concepts and thus, is not of much interest for us.

COM - DCOM The Component Object Model (COM) is Microsoft's binary standard that defines how objects can interact. DCOM is COM distributed across the network. The client of a DCOM object sees the object as if it is in its own address space, whether it is on a local machine or a remote machine.

DCOM also encompasses security features. DCOM is briefly described in Section 2.5.1.

JAVA RMI Java was written for the Internet and Java Remote Method Invocation (RMI) is a mechanism for accessing the methods of a remote Java class. The remote object has to be Java because RMI depends on Java serialization. Java serialization allows the objects to be transmitted as stream. Java RMI is briefly presented in Section 2.5.2.

CORBA CORBA is the standard for distributed computing developed by the Object Management Group (OMG) for distributed systems. CORBA is described in Section 2.6.

JAVA IDL Sun has provided a series of Java classes that are CORBA-compliant along with an IDL compiler that creates CORBA stubs and skeletons in Java.

2.5.1 COM - DCOM [Ses98]

COM is in several points similar to DCE RPC. The client requests that an object is activated on the server, and is passed back an interface on that object. This interface may be the generic interface that all COM objects support (`IUnknown`) or it may be another interface that the client requested.

The COM architecture is depicted in Figure 2.2. The client accesses the interface by the means of proxy. This proxy plays the role of the interface that is requested. The interface method invocations are sent, via the channel, to the stub code on the server machine. The channel is either Windows messages (server on the local machine) or RPCs (server on remote machine).

DCOM can be described as COM with a longer wire. Let us take an example: a client in one process space wants to access an interface on an object in another process space. When the client activates the object, COM loads the proxy in the client's address space and a stub object in the object's address space. The proxy talks to the stub, marshalling the method calls across the network.

COM and DCOM are widely spread in the Microsoft's community. Unfortunately they are not standardised, which dissuades big companies from investing in this technology to offer services to their customer.

COM-DCOM is very well suited to a Microsoft Windows environment because it integrates well with all the development tools and with the OSs. However, it is not practical in other situations.

2.5.2 JAVA RMI

JAVA Remote Methods Invocation (RMI) is a mechanism for accessing methods of a remote Java class.

RMI depends on three components. The first of these components involves defining an interface that extends the `Remote` interface. This interface has methods that allow an object to be accessed outside of the current virtual machine (even on another machine).

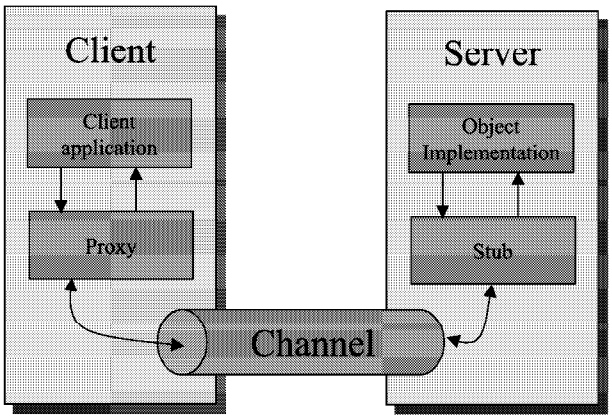


Figure 2.2: COM architecture

The second component is an implementation of the Java `interface`. The task of the developer is to extend this class with a class that must provide the implementation of the methods declared.

The third component is a Naming mechanism, this is used by the client to locate the objects and by the server to publicize itself so that the client can find it. For this, there is a *registry* running on the server machine that handles information about the available objects.

Note that with RMI, objects are named using URL.

RMI is simple and fits well in today's "Web-compliant" technology. For big applications, robustness, etc. RMI is less powerful than CORBA.

2.6 CORBA

The major telecommunication companies, as well as the providers of software for computers, have been working with various organizations to promote common standards. For example, many telecommunications are participating as active members of the Object Management Group (OMG) and almost all have envisaged the use of CORBA in order to facilitate linking different systems together with a minimum of fuss. In 1994 the international telecommunication companies association voted to make CORBA one of their standards [Har98].

2.6.1 OMG Standard

The Object Management Group (OMG) was formed in 1989 in order to create standards and thus produce specifications for the interoperability and portability of distributed object-oriented applications. These specifications were created using ideas and technology from OMG members who respond to Request for information(RFI) and Request For Proposal(RFP) issued by the OMG [Sie96, OMG98].

The OMG's Object Management Architecture (OMA) attempts to define, at a high level, the various facilities necessary for distributed computing. In the Object Management Architecture(OMA) object model, objects provide services, and clients issue requests for those services to be performed on their behalf. As depicted in Figure 2.3 the OMA model comprises four component categories: the Object Request Broker, Object Services, Common Facilities and Application Objects. The core of

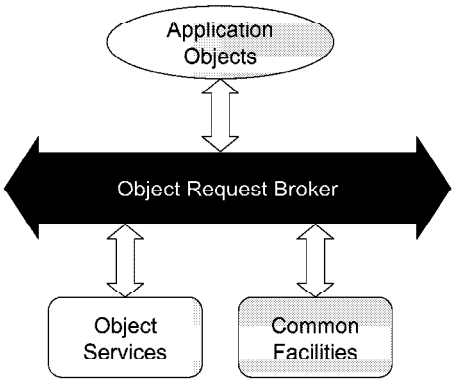


Figure 2.3: The OMA Reference Model

the OMA is the Object Request Broker (ORB), a mechanism that provides transparency of object location, activation, and communication. The ORB provides the basic interaction capability, and the Object Services provide the basic services. For example, the ORB provides ways for requests to be delivered to servers, but does not define how the servers are found, this is the role of the "trader service" of the Object Services layer. Common Facilities also provide services, but these services are typically at a higher level. They are closer to the application levels than are Object Services. For example, compound document management and help systems are Common Facilities. Application objects are objects that implement programmer-defined IDL interfaces.

Object Services The Object Services define a set of low-level services that allow application objects to communicate in a standard way. These services include the following:

- The *Naming Service.* Before using an object, a client program must get an identifier for the object, known as an *object reference.* This service allows a client to locate object references based on abstract, programmer-defined object names.
- The *Trader Service.* This service allows a client to locate object references based on the desired properties models.
- The *Object Transaction Service.* This service allows programs to interact using transactional processing models.
- The *Security Service.* This Service allows programs to interact using secure communications.

- The *Event Service.* This service allows objects to communicate using decoupled, event-based semantics, instead of the basic ORB function-call semantics.

Common facilities The Common Facilities define a set of high-level services that applications frequently require when manipulating distributed objects. The Common Facilities are divided into two categories: horizontal and vertical. The horizontal facilities consist of user interface, information management, system management, and task management facilities. The vertical facilities standardise IDL (Interface Definition Language) specifications for market sectors such as healthcare and telecommunications.

The Common Object Request Broker Architecture (CORBA) defines the architecture of ORB-based environment. This architecture is the basis of any OMG component and is the enabler of the OMG vision. It was the first component to receive a formal specification and is continuously being enhanced to support the technology emerging from the work of the OMG and its members.

In Figure 2.4, a request passing from a client to an object implementation in the CORBA architecture is presented. Two aspects of this architecture stand out:

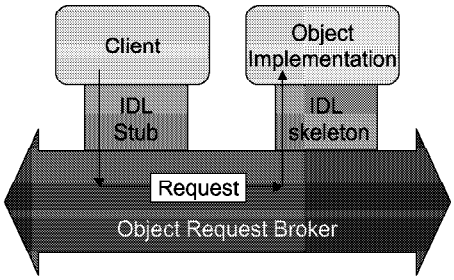


Figure 2.4: A Request passing from client to object implementation

- Both client and object implementation are isolated from the ORB by an Interface Definition Language (IDL) Interface.

  "CORBA requires that every object's interface be expressed in OMG IDL. Clients see only the object's interface, never any implementation detail. This guarantees substitutability of the implementation behind the interface - and plug-and-play component software environment" [Sie96].

- The request does not pass directly from client to object implementation; instead, request is handled by an ORB.

  "Every invocation of a CORBA object is passed to the ORB; the form of the invocation is the same whether the target object is local or remote. Distribution details remain in the ORB where they are handled by software you bought, not software you built. Application code, freed of this administration burden, concentrates on the problem at hand" [Sie96].

The typical development process of a CORBA based application is depicted in Figure 2.5. The shaded boxes indicate code that the developer has to write and the

boxes with dotted lines indicate generated components. The task of the developer is first to write the IDL specification (i.e. define the different interfaces of the components of the applications) and then, to implement the interfaces that have been defined. The IDL specification of the interfaces is passed to an IDL compiler, associated with the ORB, which generates stub code and skeleton files that are then linked to the implementation of the interfaces thereby shielding the developer of the distributed application from the difficult task of handling the distribution issues.

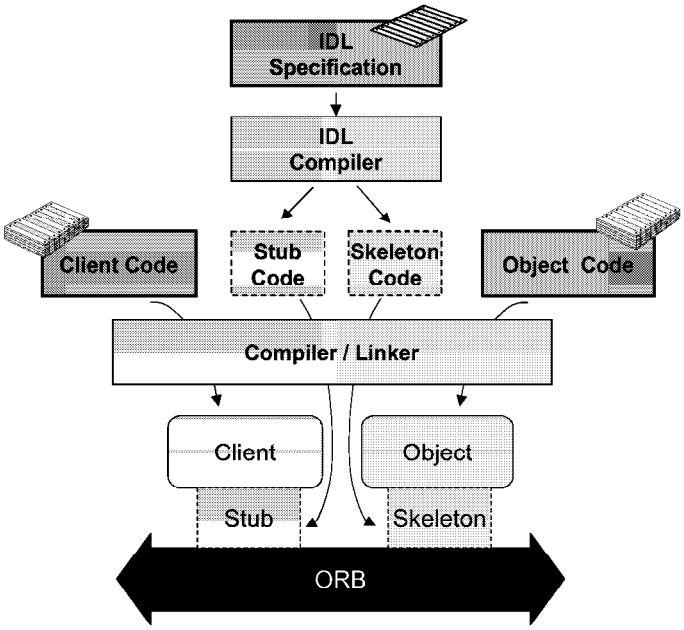


Figure 2.5: From IDL to client and object implementation

The CORBA architecture is composed of five major components:

- ORB Core
- Interface Definition Language
- Dynamic Invocation Interface
- Interface Repository
- Object Adapters

2.6.2 ORB Core

The purpose of the ORB is to deliver request to objects and return any output values back to clients. The ORB accomplishes this task completely unbeknownst to the clients. The clients do not need to know where objects reside on the network, how they communicate, how they are implemented, how they are stored, nor how they execute.

Before a client can issue a request to an object, it must hold an object reference for that object. An ORB uses object references to identify and locate objects so that it can direct requests to them. Object reference can be made persistent by first asking the ORB to convert them to strings. Clients can store these string object references, in their own private data files and later retrieve them, ask the ORB to change them back into object references, and use them to make requests. This capability can be used to maintain persistent links between objects and the applications that use them.

The CORBA specifies two different ways in which clients can issue requests to objects:

- static invocation via interface-specific stubs
- dynamic invocations via the Dynamic Invocation Interface(DII)

Regardless of which of these methods the client uses to make request, the ORB locates the desired object, activates it if it is not already executing, and delivers the request to it. The object has no knowledge of whether the request was made by the static stub or the DII, nor does it know about where it came from (security aspect is taken care by the ORB). It performs the requested service and returns any output values back to the ORB which then in turn returns it to the client.

2.6.3 Interface Definition Language(IDL)

Before an application can make use of an object, it must know what services the object provides. In CORBA, object interfaces are described in IDL, a declarative language with a syntax resembling that of C++. IDL provides basic data types (such as short, long, float, double, and boolean), constructed types (such as struct and unions), and template types (such as sequences and strings). These are used in operation declarations to define arguments types and return types. The operations are used in interface declarations to define the services provided by objects. IDL also provides a module construct that can hold interfaces, type definitions, and other modules for name scoping purposes.

The type Interface describes CORBA objects, they are also used as object reference types. Operations can be declared to return object references and to take object references as arguments by using interface names. IDL provides interface inheritance in which derived interfaces inherit the operations and types defined in their base interfaces. The fact that CORBA IDL is a declarative language heightens the separation of interface and implementation that is emphasized in OO systems development. CORBA IDL mapping to many languages such as C,C++, and smalltalk is supported through the IDL compiler that are bundled with the ORB products. When IDL files are compiled, the compiler generates stubs and skeleton programs. The stub provides an interface to the ORB that performs marshalling and un-marshalling of operation parameters transparent to the client. The skeleton is the corresponding server side implementation of the IDL interface. In CORBA all a client needs to know is the IDL interface definition and the description of what the object does and

at runtime only one other piece of information is needed: the object reference.

2.6.4 Interface Repository(IR)

The ORB stores all the IDL definitions for all the objects it knows about in the Interface Repository. IR is used mainly to provide interoperability between two ORBs, to check inheritance graphs and to provide type checking. A runtime object interface details could be retrieved on-line from the IR that will be exploited in dynamic interface invocations. An IR may be implemented on top of a database.

2.6.5 Dynamic Invocation Interface(DII)

The DII gives the client object the capability to invoke operations at run time. Server object may not detect whether the invocation came into the ORB via the SII or the DII since the ORB passes the message as it was done in the case of SII. The DII is useful in the case of server objects that are added to the system at a later stage and are discovered through a trading service. DII allows deferred synchronous i.e. non-blocking with return results in addition to the synchronous and asynchronous invocations. Clients will usually consult a Trading service to obtain the object reference (object handle). Then client invoke the necessary operations on the ORB interface to get the newly found object's interface operation name, and their parameters. As to the semantical interpretation of these operations the client find them from the trader.

2.6.6 Object Adaptor(OA)

CORBA allows object implementations to vary widely. In some cases multiple IDL interfaces may be implemented by a single server program, whereas in other cases an IDL interface may be implemented by a series of shell scripts, one for each operation. Some of these may be legacy applications while others may be OO systems developed specifically to work with the ORB. The OA provides the flexibility to permit the integration of legacy applications by the use of a wrapper mechanism without setting implementation criteria.

An Object Adaptor provides the means by which various types of object implementations utilize ORB services such as:

- object reference generation
- object method invocation
- security
- activation and deactivation of object implementations.

Each ORB is expected to provide a general OA called the Basic Object Adaptor (BOA), which is intended to support objects implemented as separate programs. It

is expected that most object implementations will work with the BOA because its services are flexible enough to accommodate different object implementations, such as:

- persistent implementations which are activated by something other than BOA
- shared implementations in which multiple objects coexist in the same program
- ensured implementations in which only one object implementation exists per program
- server-per-method implementations in which each operation supported by an object is a separate program.

Object Adaptors other than BOA may exist. The Library Object Adaptor(LOA) may be used with lightweight object implementations that are co-resident with client applications, for example an object oriented database.

2.6.7 CORBA Interoperability

The CORBA specification contains definitions of ORB domains, bridges, and Interoperable Object References (IORs). The approach of the architecture is to identify the things that can be used as common representation between domains and to suggest ways in which ORB domains can communicate using common representation. An IOR contains the same information as a single domain object reference, but adds a list of protocol profiles indicating in which communication protocols the domain of origin can accept a request . The protocol interoperability problem is addressed in the General Inter-ORB Protocol (GIOP) .

The GIOP defines a linear format for the transmission of CORBA requests and replies, without requiring a particular network transport protocol.

The *Internet Inter-ORB Protocol* (IIOP) is a specialization of the GIOP that specifies the use of TCP/IP. It defines some primitives to assist in the establishment of TCP connections. For example, JAVA ORB are implemented using IIOP. Other ORB support proprietary protocols and IIOP. With the use of a GIOP as ORB-

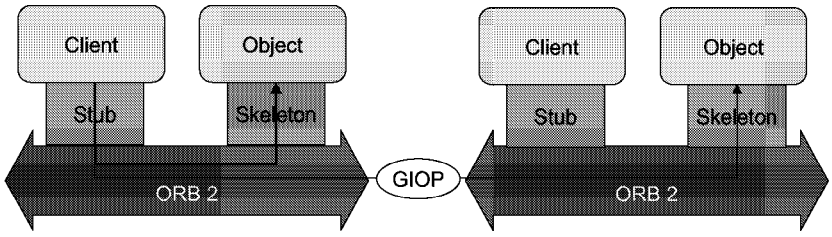


Figure 2.6: Interoperability between two ORBs with GIOP

to-ORB communication, the client does nothing different for a remote invocation compared to the local case (Figure 2.6). It passes its usual IDL-based invocation to its local ORB. If the invocation contains object an reference of a local object

implementation, the ORB routes it to its target object; if not, the ORB routes the invocation to a remote ORB. The remote ORB then routes the invocation to the target object.

2.6.8 Interceptors

An interceptor [OMG99] is responsible for the execution of one or more ORB services. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and a target object. When several ORB services are required, several interceptors may be used.

Two types of interceptors are defined in this specification:

- Request-level interceptors, which execute the given request.
- Message-level interceptors, which send and receive messages (unstructured buffers) derived from the requests and replies.

Interceptors provide a highly flexible means of adding portable ORB Services to a CORBA-compliant object system. The flexibility derives from the capacity of a binding between client and target object to be extended and specialized to reflect the mutual requirements of client and target. The portability derives from the definition of the interceptor interface in OMG IDL.

The kinds of interceptors available are known to the ORB. Interceptors are created by the ORB as necessary during binding, as described next.

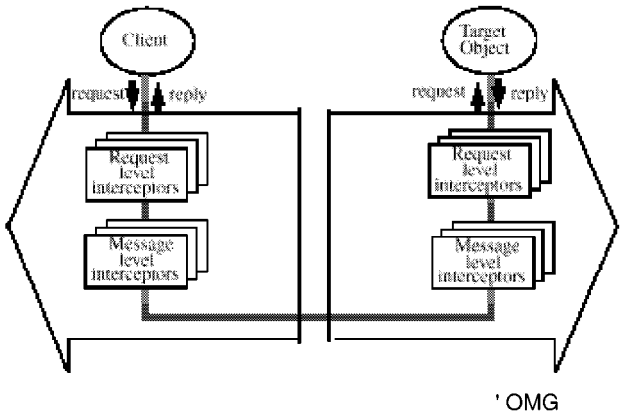


Figure 2.7: Interceptors called during invocation path.

2.6.8.1 Request-Level Interceptors

Request-level interceptors are used to implement services which may be required regardless of whether the client and target are collocated or remote. They resemble the CORBA bridge mechanism in that they receive the request as a parameter, and subsequently re-invoke it using the Dynamic Invocation Interface (DII). An example

of a request-level interceptor is the Access Control interceptor, which uses information about the requesting principal and the operation in order to make an access control decision.

Request-level interceptors could be used for services such as transaction management, access control, or replication. Services at this level process the request in some way. For example, they may transform the request into one or more lower-level invocations or make checks that the request is permitted. The request-level interceptors, after performing whatever action is needed at the client (or target), reinvoke the (transformed) request using the Dynamic Invocation Interface (DII) `CORBA::Request::invoke`. The interceptor is then stacked until the invocation completes, when it has an opportunity to perform further actions, taking into account the response before returning.

Interceptors can find details of the request using the operations on the request as defined in the Dynamic Skeleton interface in CORBA 2. This allows the interceptor to find the target object 1 , operation name, context, parameters, and (when complete) the result. If the interceptor decides not to forward the request, for example, the access control interceptor determines that access is not permitted, it indicates the appropriate exception and returns. When the interceptor resumes after an inner request is complete, it can find the result of the operation using the result operation on the Request object, and check for exceptions using the exception operation before returning.

2.6.8.2 Message-Level Interceptors

When remote invocation is required, the ORB will transform the request into a message, which can be sent over the network. As functions such as encryption are performed on messages, a second kind of interceptor interface is required.

The ORB code invokes each message-level interceptor via the `send_message` operation (when sending a message, for example, the request at the client and the reply at the target) or the `receive_message` operation (when receiving a message). Both have a message as an argument. The interceptor generally transforms the message and then invokes send. Send operations return control to the caller without waiting for the operation to finish. Having completed the `send_message` operation, the interceptor can continue with its function or return.

When remote invocation is required, the ORB will transform the request into a message that can be sent over the network. Message-level interceptors operate on messages in general without understanding how these messages relate to requests (for example, the message could be just a fragment of a request). Note that the message interceptors may achieve their purpose not by just transforming the given message, but by communicating using their own message (for example, to establish a secure association). Fragmentation and message protection are possible message-level interceptors.

`send_message` is always used when sending a message, so it is used by the client

to send a request (or part of a request), and by the target to send a reply.

When a client message-level interceptor is activated to perform a `send_message` operation, it transforms the message as required, and calls a send operation to pass the message on to the ORB and hence to its target. Unlike invoke operations, send operations may return to the caller without completing the operation. The interceptor can then perform other operations if required before exiting. The client interceptor may next be called either using `send_message` to process another outgoing message, or using `receive_message` to process an incoming message. A target message-level interceptor also supports send_message and `receive_message` operations, though these are obviously called in a different order from the client side.

2.6.9 CORBA Implementations

There are a lot of implementations of the CORBA standard. The leaders in the market are Orbix^{TM} from IONA and Visibroker. Recently a new ORB appeared, ORBACUS from Object Oriented Concepts, Inc (OOC^{TM}). Unlike Orbix and Visibroker, Orbacus is licensed as "free for non-commercial use".

2.6.9.1 Orbix

Orbix [ION98], from IONA Technologies, was the first ORB to fully support the CORBA standard and the first to support C++. Initially, Orbix supported the C++ language, and support has since been added for other languages and programming environments, including Smalltalk, Ada, OLE, and JAVA. The IONA product supporting the JAVA mapping is called OrbixWeb. Orbix is an ORB that implements all of the elements of the CORBA 2.3 including the stub approach (Static Invocation Interface or SII), the Dynamic Invocation Interface (DII), the implementation Repository and the Interface Repository. By default, all Orbix components and applications communicate using the CORBA IIOP protocol.

The components of Orbix are as follows:

- The *IDL compiler* parses the IDL definitions and produces the implementation code of the mapping language of the product (C++, Smalltalk, JAVA, etc.).
- The *Orbix library* is linked against each Orbix program and implements several components of the ORB, including the DII, the DSI, and the core ORB functionality.
- The *Orbix daemon* is a process that runs on each server host and implement several ORB components, including the Implementation Repository.
- The *Orbix Interface Repository server* is a process that implements the Interface Repository.

Orbix also includes functionality that are not yet stable in the CORBA standard that extends the capabilities of the ORB; for example, Callbacks and Filters. Today,

Orbix is the market leader of the ORBs. The Filter mechanism is critical to the proposed methodology and is worth a detailed description.

Orbix Filters Orbix/OrbixWeb allows you to specify that additional code to be executed before or after the normal code of an operation or attribute. This support is provided by allowing applications to create filters, which can perform security checks, provide debugging traps or information, maintain an audit trail, etc.

There are two forms of filters in Orbix/OrbixWeb:

- per-process filters.
- per-objects filters.

Per-process filters monitor all operation and attribute calls leaving or entering a client's or server's address space, irrespective of the target object. Per-object filters apply to individual objects.

Per-Process Filters monitor all incoming and outgoing operation §and attribute requests to and from an address space. Each process can have a chain of such filters, with each element of the chain performing its own actions. Each filter of the chain can monitor ten individual points during the transmission and reception of an operation or attribute request, as shown in Figure 2.8. The four most commonly used filter

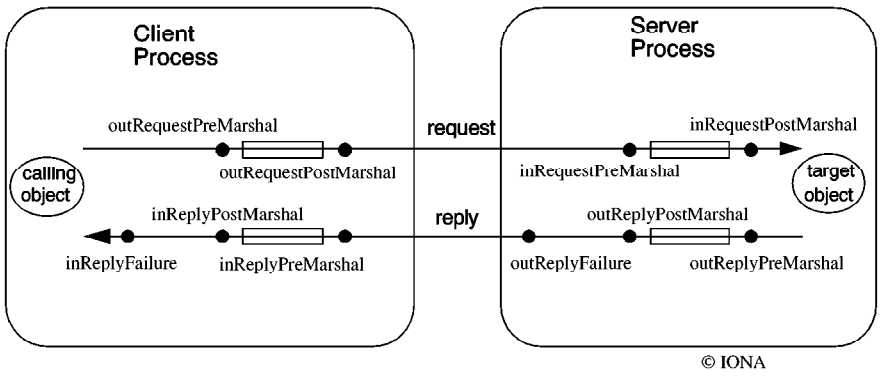


Figure 2.8: Per-Process filters points

points are:

- *outRequestPreMarshal*
- *inRequestPreMarshal*
- *outReplyPreMarshal*
- *inReplyPreMarshal.*

The Post marshalling Filter points are:

- *outRequestPostMarshal*

- *inRequestPostMarshal*
- *outReplyPostMarshal*
- *inReplyPostMarshal.*

"Post" and "Pre" marshaling indicate if the parameters have already been removed from the request.

The Failure Filter points are:

- *outReplyFailure*
- *inReplyFailure.*

Per-object filters are associated with a particular object and not with all objects in an address space as in per-process filters; per-object filters apply to intra-process operation requests. The following filtering points are supported:

- *Per-object pre* this filter applies to operation invocations on a particular object- before they are passed to the target object.
- *Per-object post* This filter is applied to operation invocations on a particular object - after they have been processed by the target object.

2.6.9.2 Visibroker

Visibroker [Inp98] is a complete CORBA ORB and supports a development environment for building, deploying, and managing distributed object applications that are interoperable across platforms. Visibroker encompass both C++ and JAVA mapping. Objects built with Visibroker for JAVA are easily accessed by Web-based applications that communicate using the OMG's IIOP.

Visibroker has the following features:

- Interface Repository
- Dynamic Invocation Interface
- Dynamic Skeleton Interface
- Smart Binding
- Smart Agent
- Object Activation Daemon
- Enhanced Thread and Connection Management
- Location Service
- Enhanced idl2java Compiler
- Object request Debugger
- Smart Stubs
- Interceptors

- Communication Event Handler
- Gatekeeper and SSL Pack (optional)

The interested reader can refer to [Inp98] for a complete explanation of all these features.

Visibroker Interceptors The interceptor is a powerful feature that allows you to view under-the-cover communications between clients and servers. Architecturally, the interceptor code sits between the client and server. Visibroker ORB messages sent between the client and the server pass through the interceptor at either end, and may be viewed or altered. With the interceptors you can determine exactly how the ORB is processing requests.

The designers of Visibroker recommend to be proficient with system-level programming. Interceptors work below the client and server application code, and therefore lack the comprehensive error checking of higher-level code. As a result, effects of errors can be both serious and difficult to diagnose. Some interceptor methods, such as those that change the messages between server and client, provide more potential for damage than others.

2.6.9.3 ORBacus

ORBacus [Obj98] is an ORB that is compliant with the CORBA specification. The highlights of ORBacus are:

- Full CORBA IDL support.
- Complete IDL to C++ and JAVA mapping.
- Include the basic CORBA Services: Naming, Event and Property.
- Pluggable protocols with IIOP as the default plug-in.
- Single- and Multi-Threaded support with several concurrency models: Blocking, Reactive, Thread-per-Client, Thread-per-Request and Thread Pool
- Nested method invocations, even in the single-threaded version
- Support for timeouts
- Seamless event loop integration with X11 and Windows
- Full support for dynamic programming: Dynamic Invocation Interface, Dynamic Skeleton Interface, Interface Repository and DynAny.
- IDL-to-HTML and IDL-to-RTF translators for generating "javadoc" documentations

For the moment (version 3), ORBacus proposes only persistent servers, i.e., manually launched.

2.6.10 Discussion

In Section 2.5 we presented different middleware. As there are many to chose from, when implementing a distributed application, in Table 2.1 we summarise the advantages and disadvantages of the principal middlewares.

	Advantages	Disadvantages
Sockets	-lightweight -fast communications -total control for developer	-few functionality -integration of legacy appl. very hard -all responsibilities to develop.
JAVA RMI	-simple & easy -excellent RPC	-integration of legacy appl. hard -not fast -not robust -not a standard (Sun) -lack of support for true distribution
CORBA	-standard -implementation language independence -true support for distribution - well suited for integration of legacy appl.	-heavy -complex infrastructure -current perfor. (speed, scalability) -the need of IDL
DCOM	-full integrat. in Microsoft tools -support for true distribution	-not a standard (Microsoft) -complex unless efficient use of Microsoft tools

Table 2.1: Comparison of Middlewares

The main advantage of CORBA is that it is a standard. This fact makes CORBA well suited to the telecommunications world because it does not rely on a specific company: Microsoft in the case of DCOM and Sun in the case of JAVA RMI. In addition, it is evident that most of the services that are developed can be realized on top of any middleware. Thus making the choice of a specific middleware an implementation design decision: Do you privilege simplicity?, high level functionality?, scalability?, integrability of legacy applications?, evoluability?, etc.

It is not our intent to respond to these questions. In this thesis, we attempt to give some methods for monitoring and testing middleware based software built on top of middleware, without restricting to a specific middleware. Nevertheless, we took CORBA as the basis for our work.

2.7 Distributed Systems in the Telecommunication World

Major telecommunications companies have been working with various organizations to promote common standards. Many, for example, have been active members of

the OMG and most have explored the use of CORBA in order to facilitate linking different systems together with a minimum of trouble. Different projects illustrate the emergence of distributed computing in the telecommunication world. Some of them are presented in the following sections.

2.7.1 Distributed Computing At Bell Laboratories, Lucent Technologies [MBR+98]

At Lucent, they developed a highly available distributed call processing system as a solution for the next generation of switching systems. They proposed an architecture based on modular pieces that can be used to customize the functionality of the Call center. These different modules communicate by the means of CORBA ORB.

The focus is put on the fault-tolerant aspects as the requirements on the availability of a switching system are extremely high (Call processing software must process each call request within a few hundred milliseconds, and the entire switching system should not be out of service for more then a few minutes per year).

They implemented a Mobile switching System (MSC) following their concepts and architecture. They evaluated the availability of their system and concluded, assuming a call arrival rate of 120,000 calls per hour, each and every day, 52 weeks a year, an availability of 0.999958 (which represents a loss of approximately 50,000 calls per year).

2.7.2 Distributed Computing at Swisscom

At Swisscom many projects for the provision of communication services are using a distributed environment. Among them two retained our attention: SPOT and CLUE.

SPOT - Service Pilot On TINA [MBHL97] The Service Pilot On TINA (SPOT), a project between Swisscom, (formerly Swiss Telecom PTT), Telia, KPN and Telefonica (The UNISOURCE consortium, Figure 2.9) and the two manufacturers Alcatel Alsthom and Ericsson, was devoted to the experiment and provision of multimedia services over the Pan-European ATM (Asynchronous Transfer Mode) network using the TINA (Telecommunication Information Networking Architecture) architecture and principles.

Specifically the goals were:

- Apply, validate and experiment TINA principles
- Use TINA compliant product from key vendors
- Experiment with real multimedia service(s) on top of TINA

To comply with the TINA architecture, they used a distributed processing environment that was CORBA. CORBA was used to ensure the provision of all the

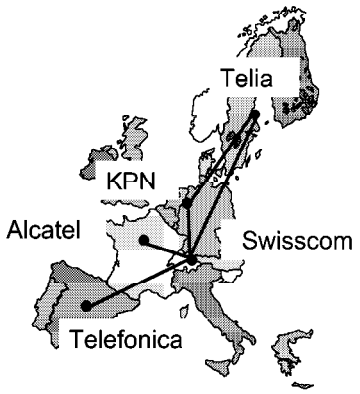


Figure 2.9: The SPOT project within the UNISOURCE consortium

distributed infrastructure for the connection management, the service management and the session management. For the stream connection (voice and video) they were using the network API provided by the ATM switches directly.

The project was more or less successful, the partners implemented a desktop video conferencing service over the ATM network. The practical use of CORBA was demonstrated. However, the project did not demonstrate the practical superiority of TINA (advocated by the aficionados of TINA) for the provision of services.

CLUE - Closed User Group Environment [SBC+99] The fundamental idea of CLUE is to give each user the ability to select or to create suitable environments for the use of services that prevent other users outside these environments from interfering with the service sessions. Users achieve their security objectives simply through running the desired telecommunication service in the closed user group environment. For this, a prototype was implemented (Figure 2.10). The backbone of

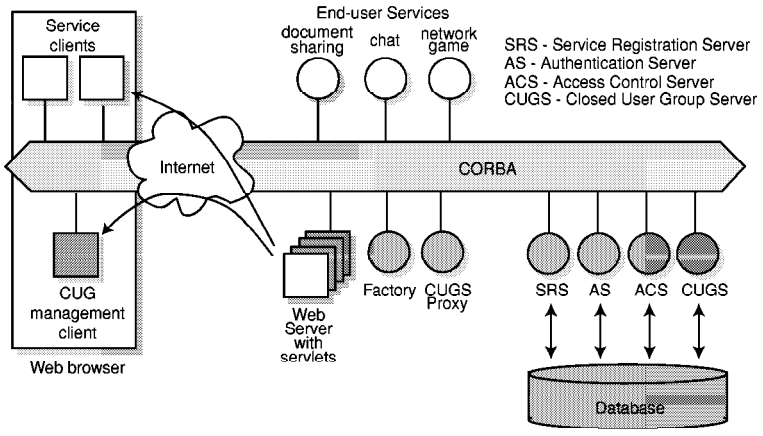


Figure 2.10: Architecture of CLUE

the architecture is CORBA, together with WWW technology and JAVA Applets as access technology.

The end-user services are depicted in the upper part of Figure 2.10. They are realised as, or wrapped up by, CORBA server objects. The service clients are implemented as JAVA applets, downloaded to the users via the Internet using WWW technology, and executed by the users' web browsers. The service clients use the underlying CORBA layer, which is either built into the browser or downloaded together with the service client from the service center, to interact with the end-user services. The components that composed the CLosed User group Environment are depicted in the lower part of the figure. They are implemented as CORBA server objects, JAVA applets and JAVA servlets.

2.7.3 Distributed Computing at Alcatel [MCL99, LBGD99]

At Alcatel, they developed the PERCO (Plate-forme d'Exécution Répartie COmmune) platform. PERCO is an industry-provided software platform that specifically addresses the requirements of highly available dependable autonomous systems. Its basic tenet is to use UML specifications to build as much of the application as possible, while integrating real-time and fault-tolerant properties at the architectural level.

The PERCO approach is to take advantage of the potential of UML to integrate the application and its environment at a high level. It makes a broad use of CORBA and its integration within UML to solve the aspects related to portability distribution, reliability, openness and heterogeneity.

The architecture is presented in Figure 2.11. PERCO can be considered as a CORBA extension: CORBA plus fault tolerance, plus load balancing, plus real-time. As PERCO makes an extensive use of CORBA and adds requirements and directions for future evolutions, Alcatel uses this experience to be involved in the CORBA standardization process, notably for minimum CORBA, Realtime CORBA and Fault-Tolerant CORBA.

PERCO relies on CORBA 2.3 for portability and heterogeneity. The current implementation is based on ORBACUS^{TM} and Visibroker^{TM}.

Alcatel wants to integrate a support for monitoring and testing the applications deployed on top of PERCO. Therefore, the work presented in this dissertation is currently integrated in the PERCO Platform (see Chapter 6).

2.7.4 Distributed Computing at Motorola and IRIDIUM, Inc

One of the most impressive examples of large distributed telecommunications application is IRIDIUM[1]. IRIDIUM is an international consortium of telecommunication companies that have joined together to fund a satellite-based global cellular network. The system consists of 66 satellites. Motorola is the leading IRIDIUM developer. It

[1]IRIDIUM is currently in a very difficult financial situation due mainly to the lack of customers (because of the price of the terminals.)

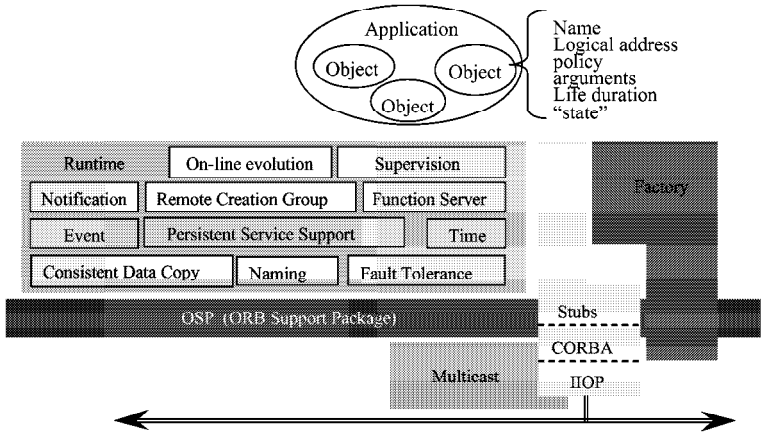


Figure 2.11: The PERCO platform

chose CORBA Orbix software to build and control the ground station portion of the system.

Dr. David Casstillo, lead software designer of the System Control of Iridium said:

> *" Early on in this project, we realized we would need to take a distributed, object-oriented approach in developing critical software and quickly saw CORBA as the basis for the best solution. Following an exhaustive examination of existing CORBA implementations, Motorola opted for Orbix as an easy-to-use, robust, reliable, and faithful implementation of the CORBA specification. Orbix presents CORBA in a natural way to C++ programmers. The principles CORBA and Orbix are based on, are well established. With Orbix, our programmers can develop distributed, object oriented applications quickly, following a consistent and straightforward, standard-based model, without compromising performance. Orbix handles many of the complexities of distribution, leaving our programmers to focus on functionality."*

The idea of IRIDIUM is to allow any user, assuming he possesses the adequate user-terminal, to call to and receive calls from anywhere on the earth.

2.7.5 Distributed Computing at Telefonica [Gom98]

In 1995, Telefonica started to develop a Center for Advanced Service Provisioning (CPSA). The CPSA has a number of key requirements:

- Fast delivery of services,
- Capacity to evolve,
- Ability to handle large-scale, distributed telecom networks,
- Independence from Switching Network suppliers

- Improvement in performance despite a large number of users
- Ability to manage the system in its multiple configurations and control the entire life cycle of services
- The same platform to support different services at the same time.

To fulfil all these requirements, Telefonica chose CORBA to provide the communication and management mechanism. CORBA is everywhere in the system. All the processes developed are CORBA servers and every single user call uses CORBA transactions between them.

Table 2.2 presents the services that are already deployed with CSPA. From this table one can notice the variety of services that the platform simultaneously supports.

Service	Description	CORBA invoc./day (in million)
002/004	Commercial assistance to Caller	50
009	Automatically assisted Calls	3.5
Wake-up service	Automatic wake-up service	1
Spain-Direct	International calls to Spain	4.8
005/008	Automatic assisted int'l calls	1.5
CIAC	Int'l customers attention center service	1.5
TPA	Advanced personal Card service	5
SIC	Consumption information service	0.15
Funobuzon	Personal telephonic Agenda service	0.15
PARI	Auxiliary service to IN access	N/A
Direct-Peru	Int'l calls to Peru	N/A
Activa Roaming	Int'l roaming with Activa GSM cards	0.8
Moviline Family	Single mobile number for all the family	2

Table 2.2: Services currently available from Telefonica using CSPA

Gonzalo Gomez, Telefonica, stated:

> *"The end in developing these [code generation from formal specifications and use of IIOP] is not purely to augment the efficiency of CSPA, although this is a constant goal, but also to add greater functionality to the platform that will allow us to provide more services for customer. For example, there is no theoretical boundary on using the CSPA to provide multimedia and Internet services to the customer and to Telefonica itself. These are clearly the next step and are obviously being developed by our competitors. But What CSPA gives us over these competitors is an incredibly short time-to-market, huge saving in human resources optimization and massive efficiency in infrastructure with one extensible platform serving all."*

CSPA is the first time that CORBA has been introduced to support Intelligent Network service. It shows that CORBA is reliable, performant and scalable. CSPA is more flexible and decreases the service development time compared to the traditional IN solutions.

2.8 Conclusions

In the first part of this chapter, we defined a service, and the different tasks involved for the provision of new services. Among these tasks, we are investigating the testing of services. For this, we carefully looked at the infrastructures and environments on which services are deployed. We presented different middlewares that are used for this purpose. CORBA has a big advantage compared to DCOM and JAVA RMI; it is a standard and does not depend solely on one software manufacturer. CORBA is also well suited for the integration of legacy applications.

As reported by the industry and presented in Section 2.7, CORBA is the leader middleware in the Telecommunication domain. The main reason for this is probably the long experience in standards in the telecommunications industry where they used to make standards first and then implement. Another reason of the success of CORBA in the telecommunication world might be the strong investment of these companies in the TINA consortium where CORBA was advocated as the distributed processing environment. Even if TINA was not successful as was promised, these companies gained a strong experience in the use of that type of middleware.

Having defined a service and specified on which environment it is deployed, we have examined the frameworks for the testing and monitoring of services, which are presented in the following chapters.

Chapter 3

Events

The content of this Chapter was published in [DLH98]

3.1 Chapter overview

As we discussed in the Introduction (Chapter 1) of this dissertation, the method we developed for monitoring and testing middleware based communication services, consists in expressing properties (or constraints) on an abstract level and then, checking for the violation of these properties, at run-time, when the services execute.

For this, we need to model the system under test to be able to:

1. express properties in an abstract way (using a Formal Method),
2. observe at run-time, in the system under test, what is expressed in the properties,
3. match what is occurring in the system with what is expressed in the abstract properties.

The purpose of this Chapter is to show how it is possible and to present our solution. In this chapter we present the modelling of communication services by the means of events. We first explain the concept of event-based behavioral abstraction and detail the set of generic events that we use for modelling distributed applications. This chapter has been written in tight collaboration with Falk Dietrich.

3.2 Introduction

There are different methods to model distributed systems. According to need we can model the system in terms of:

- *flow of control.* In this model the flow of control allows following what is happening in the system. This kind of model is often used in debugging tools where

the execution is repeatedly stopped in order to examine states and variables, and then either continuing or re-executing some blocks of the program.

Using such models, if an undesirable behavior occurs with very low probability, the programmer or tester may never be able to recreate the error situation. This has been referred to as the "Heisenberg Uncertainty" [LP85] principle applied to software or the "Probe effect" [Gai85].

- *events.* In this model the behavior of the system is modeled as a succession of events. An event is understood as *something that happens in the system*: The change of a variable value, the sending of a message, the creation of an object, etc.

  In an event-based monitoring or debugging approach, all the events occurring in the system can be stored in a database and therefore be used to exactly replay the execution. The event-based approach therefore does not suffer from the probe effect. Furthermore, event-based modelling might also allow the detection of concurrent events and not only the attribution of a sequential order to all events.

3.3 Event Based Modelling

Considering the advantages of the event-based model, our model, presented in Section 3.4, is based on the notion of *events.*

In [Bat95], Bates states, on the usefulness of Event-Based Behavioral Abstraction (EBBA), that

> *"Event-based Behavioral Abstraction changes the emphasis on detailed state examination to one in which a user is presented with abstractions of program behavior and is provided mechanisms for structuring and manipulating those views. EBBA is a systematic approach to the modelling process that a tool user can use to focus on developing behavior models that explain program activity rather than directing activity at obtaining information necessary to verify the models. EBBA-based tools provide means to obtain actual behavioral information, compare the behavior to user models, and show the user how well the models fit the actual behavior"*

The use of events as the basis information unit makes EBBA largely independent of target system implementation. As we will see in Chapter 5, the set of events is used to express properties or behavioral constraints.

3.4 The Set of Events

The question we answer in this section is the following: How can we use event-based behavioral abstraction to represent the behavior of distributed multimedia services developed on top of middleware platforms, such that (i) the chosen events allow a

faithful representation of today's industrial implementations of distributed systems and (ii) the identified events are easily observable in these systems?

To respond to this question we derive a set of twenty events (Table 3.1) that is appropriate for modelling distributed services. These twenty events satisfy the requirement of being easily observable as we will show later in this paper. We consider observable events at four different levels: the object-, thread-, process- and system level. The classification of events into these four groups is mainly intended to facilitate the presentation. For a detailed and formal description of all events we refer the interested reader to [DLKH98b].

The set of events has been determined by collaborating with several industrial players and by taking into account the tradeoffs between flexibility and complexity.

We assume that the service is built on top of a middleware architecture like CORBA. CORBA is a standardized architecture for object-oriented distributed systems with transparent distribution and easy access to components. CORBA requires that every object's interface be expressed in the Interface Definition Language (IDL). Clients only see the object's interface but never any of the implementation details. Every invocation of a CORBA object is passed to the Object Request Broker (ORB); even when the object is local. All distribution issues like parameter transfer to the remote object, are handled by the ORB.

The complete definitions, semantics and details can be found in the Thesis of Falk Dietrich [Die00].

3.4.1 Syntax

The events are described as *tuples*, and thus the syntax is as follows:

$$(event\_id, i_1, i_2, \ldots, i_n)$$

where the first element *event_id* is the type of the event, and the other elements i_n ($n \geq 1$) are information or "parameter" of the event. For example, for the event modeling the creation of an object, i_1 would be the id of the object. The precise syntax of each event type is given in the following sections.

3.5 Object Level

3.5.1 o_outReq (Outgoing Operation Request)

Description: An event of type *o_outReq* occurs at the instant when an object has finished sending a request to execute an operation on another object.
Syntax: $(o\_outReq, (o_1, o_2, oper, prm))$

Table 3.1: Events summary

Name	Level	Description
o_outReq	object	outgoing operation request
o_inReq	object	incoming operation request
o_outRep	object	outgoing operation reply
o_inRep	object	incoming operation reply
p_inReq	process	incoming operation request
p_newO	process	object creation
p_delO	process	object deletion
p_newT	process	thread creation
p_delT	process	thread deletion
p_reqRef	process	request for an object reference
p_recRef	process	receipt of an object reference
t_assT	thread	thread assignment
t_relT	thread	thread release
t_outReq	thread	outgoing operation request
t_outRep	thread	outgoing operation reply
t_inRep	thread	incoming operation reply
s_oReg	system	object registration
s_oDereg	system	object deregistration
s_newP	system	process creation
s_delP	system	process deletion

o_1 object identifier of object that calls the operation
o_2 object identifier of object that offers the operation
oper operation name
prm parameter list

3.5.2 o_inReq (Incoming Operation Request)

Description: An event of type *o_inReq* occurs at the instant when an object has just started executing an operation as requested by another object.
Syntax: $(o\_inReq, (o_1, o_2, oper, prm))$
o_1 object identifier of object that calls the operation
o_2 object identifier of object that offers the operation
oper operation name
prm parameter list

3.5.3 o_outRep (Outgoing Operation Reply)

Description: An event of type *o_outRep* occurs at the instant when the object has completed processing the request. At this event, the underlying architecture starts

transferring the result back to the object that requested the execution of the operation.
Syntax: $(o\_outRep, (o_1, o_2, oper, prm))$
o_1 object identifier of object that calls the operation
o_2 object identifier of object that offers the operation
$oper$ operation name
prm parameter list

3.5.4 o_inRep (Incoming Operation Reply)

Description: An event of type o_inRep occurs at the instant when the calling object has completed the reception of the reply for the execution of an operation from the called object.
Syntax: $(o\_inRep, (o_1, o_2, oper, prm))$
o_1 object identifier of object that calls the operation
o_2 object identifier of object that offers the operation
$oper$ operation name
prm parameter list

3.6 Thread Level

3.6.1 t_assT (Thread Assignment)

Description: An event of type t_assT occurs at the instant when an operation request has just been assigned to a thread.
Syntax: $(t\_assT, tid, (o_1, o_2, oper, prm))$
tid thread identifier
o_1 object identifier of object that calls the operation
o_2 object identifier of object that offers the operation
$oper$ operation name
prm parameter list

3.6.2 t_relT (Thread Release)

Description: An event of type t_relT occurs at the instant when a thread has processed an operation request to completion.
Syntax: $(t\_relT, tid, (o_1, o_2, oper, prm))$
tid thread identifier
o_1 object identifier of object that calls the operation
o_2 object identifier of object that offers the operation
$oper$ operation name
prm parameter list

3.6.3 t_outReq (Outgoing Operation Request)

Description: An event of type *t_outReq* occurs at the instant when, during the execution of an operation request, a request to invoke another operation on another object has been sent.
Syntax: $(t\_outReq, tid, (o_1, o_2, oper, prm))$

tid	thread identifier
o_1	object identifier of object that calls the operation
o_2	object identifier of object that offers the operation
oper	operation name
prm	parameter list

3.6.4 t_outRep (Outgoing Operation Reply)

Description: An event of type *t_outRep* occurs at the instant when a thread has completed the execution of an operation, i.e., when the underlying infrastructure starts sending the result back to the calling object.
Syntax: $(t\_outRep, tid, (o_1, o_2, oper, prm))$

tid	thread identifier
o_1	object identifier of object that calls the operation
o_2	object identifier of object that offers the operation
oper	operation name
prm	parameter list

3.6.5 t_inRep (Incoming Operation Reply)

Description: An event of type *t_inRep* occurs at the instant when the response for a previous *t_outReq* has just arrived and the thread continues to execute the original operation.
Syntax: $(t\_inRep, tid, (o_1, o_2, oper, prm))$

tid	thread identifier
o_1	object identifier of object that calls the operation
o_2	object identifier of object that offers the operation
oper	operation name
prm	parameter list

3.7 Process Level

3.7.1 p_inReq (Incoming Operation Request)

Description: An event of type type *p_inReq* occurs at the instant when an operation request has just arrived at the process.

Syntax: $(p\_inReq, (o_1, o_2, oper, prm))$

o_1	object identifier of object that calls the operation
o_2	object identifier of object that offers the operation
oper	operation name
prm	parameter list

3.7.2 p_newO (Object Creation)

Description: An event of type *p_newO* occurs at the instant when the creation of an object has just taken place.
Syntax: (p_newO, oid)

oid	object identifier for the newly created object

3.7.3 p_delO (Object Deletion)

Description: An event of type *p_delO* occurs at the instant when an object has just been deleted.
Syntax: (p_delO, oid)

oid	object identifier of the deleted object

3.7.4 p_newT (Thread Creation)

Description: An event of type *p_newT* occurs at the instant when the creation of a thread has just taken place.
Syntax: (p_newT, tid)

tid	thread identifier for the newly created thread

3.7.5 p_delT (Thread Deletion)

Description: An event of type *p_delT* occurs at the instant when a thread has just been deleted.
Syntax: (p_delT, tid)

tid	thread identifier of the deleted thread

3.7.6 p_reqRef (Request of an Object Reference)

Description: An event of type *p reqRef* occurs at the instant when an object reference has just been requested.
Syntax: $(p\_reqRef, tid, cn, pn, n, m)$

tid thread identifier for the thread requesting the object reference.
cn class name for the class the requested object is derived from.
pn process name of process the object should reside in.
n used to identify specific objects.
m used to identify specific objects.

3.7.7 p_recRef (Receipt of an Object Reference)

Description: An event of type *p_recRef* occurs at the instant when an object reference has just been received.
Syntax: (*p_recRef*, *tid*, *oid*)
tid thread identifier for the thread receiving the object reference.
oid object reference returned.

3.8 System Level

3.8.1 s_oReg (Object Registration)

Description: An event of type *s_oReg* occurs at the instant when an object registration has just taken place.
Syntax: (*s_oReg*, *oid*)
oid object identifier of the object being registered.

3.8.2 s_oDereg (Object Deregistration)

Description: An event of type *s_oDereg* occurs at the instant when an object deregistration has just taken place.
Syntax: (*s_oDereg*, *oid*)
oid object identifier of the object being deregistered.

3.8.3 s_newP (Process Creation)

Description: An event of type *s_newP* occurs at the instant when the creation of a process has just taken place.
Syntax: (*s_newP*, *pid*)
pid process identifier of the newly created process

3.8.4 s_delP (Process Deletion)

Description: An event of type *s_delP* occurs at the instant when the deletion of a process has just taken place.
Syntax: (s_delP, pid)
pid process identifier of the deleted process

3.9 Use of the Model

3.9.1 The Model and the Reality

The goal of a model is to be as close to reality as possible, easy to use and not too complicated. The EBBA model presented above, focuses on the middleware layer.

At the middleware layer, we have two different types of objects as depicted in Figure 3.1: application objects, but also objects representing the underlying infrastructure. By focusing on the the middleware layer we can take advantage of the fact that both application objects and objects representing resources [LBK95] are represented in a similar way: all of these objects are represented by their interfaces which are described in the Interface Definition Language (IDL).

IDL provides an implementation language independent representation of the system, more specifically, of the interface templates that the objects in the distributed system support.

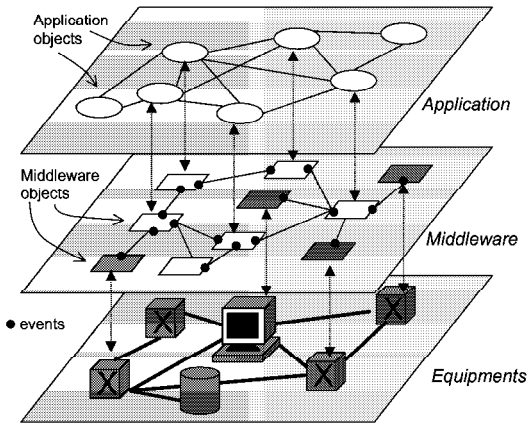


Figure 3.1: middleware abstraction layer

In order to use our model, we can, to a large extent, rely on the information given in the IDL specification. The advantages of expressing properties at the abstraction level given by IDL are appealing: a property, making reference to the items of an IDL specification, inherits the implementation language independent character from

IDL. The standardized mapping from IDL to implementation languages enables us to automate the process of finding all the IDL information at the implementation level. Therefore, when expressing properties at the IDL level, we do not need to have information about the actual implementation language.

3.9.2 An Example

We consider a simple system (see Figure 3.2) with two processes p_1 and p_2, each of them containing a single object o_1 and o_2 respectively. The IDL of the system is as follows:

```
// IDL
interface ServerObj {
  integer read( );
  void write(in integer value);
};
```

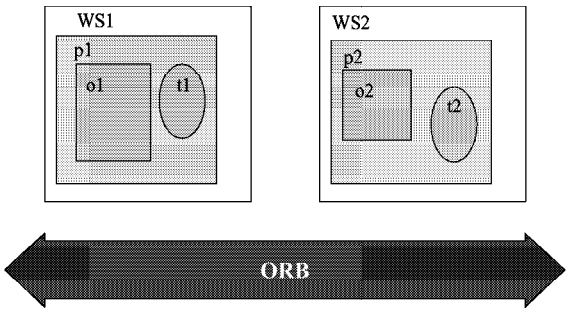


Figure 3.2: Example

A user interacts with process p_1 (e.g., by means of a graphical user interface). We consider the following scenario: A user invokes an operation on object o_1. During the execution of the operation, object o_1 requests to set the value of the attribute on object o_2 by executing the operation *write* on object o_2.

The scenario we consider leads to the events as listed in Table 3.2[1] and depicted in Figure 3.3. First, the user request the execution of the operation *set_value* on the GUI. This request arrives at the process (1). The request is assigned to thread t_1 that is available in this process (2). During the execution of the operation, object o_1 needs to obtain a reference to the remote object. It therefore requests such a reference by specifying that the object to which it wants to get a reference has to be derived from the class with class name *cn* (3). The system returns the requested object reference (4) which, in this case, points to object o_2. We can now request the execution of the operation which leads to an *t_outReq* event at object o_1 (5) and an *p_inReq* event at process p_2 (6). At this moment, in process p_2 there is no thread

[1]Note that the event in square brackets mean that this is another name for the same event

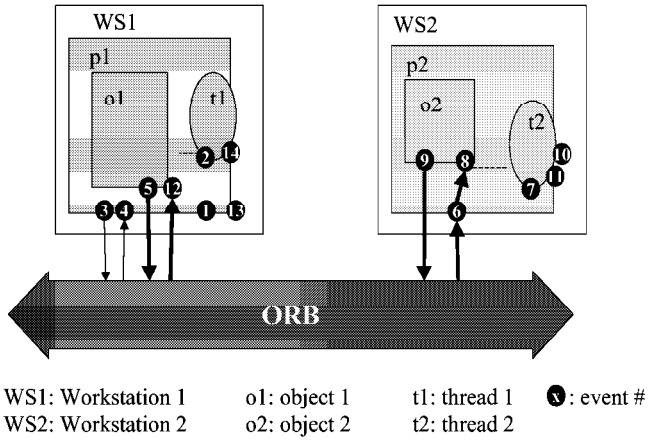


Figure 3.3: Example with events

#	Event
0	-
1	$(p\_inReq, (*, o_1, write, *))$
2	$(t\_assT, t_1, (*, o_1, write, *))$
	$[(o\_inReq, (*, o_1, write, *))]$
3	$(p\_reqRef, p_1, *, cn, *, *)$
4	(p_recRef, p_1, o_2)
5	$(t\_outReq, t_1, (o_1, o_2, write, *))$
	$[(o\_outReq, (o_1, o_2, write, *))]$
6	$(p\_inReq, (o_1, o_2, write, *))$
7	(p_newT, t_2)
8	$(p\_assT, t_2, (o_1, o_2, write, *))$
	$[(o\_inReq, (o_1, o_2, write, *))]$
9	$(t\_outRep, t_2, (o_1, o_2, write, *))$
	$[(o\_outRep, (o_1, o_2, write, *))]$
10	$(t\_relT, t_2, (o_1, o_2, write, *))$
11	(p_delT, t_2)
12	$(t\_inRep, t_1, (o_1, o_2, write, *))$
	$[(o\_inRep, (o_1, o_2, write, *))]$
13	$(t\_outRep, t_1, (*, o_1, write, *))$
	$[(o\_outRep, (*, o_1, write, *))]$
14	$(t\_relT, t_1, (*, o_1, write, *))$

Table 3.2: Example: Overview

that can handle the request. The process therefore creates a thread (7) and assigns the thread to the operation request (8). Once the operation has been processed to completion, the reply is sent back to o_1 (9). At process p_2, the thread is released (10)

and eventually deleted (11). When the reply for the operation arrives at t_1 (12), the operation `write` can be terminated (13), and thread t_1 can be released (14).

Table 3.3 gives the sequence of events that model the example. As we will see in Chapter 4, the events we defined can easily be observed at run-time. In Chapter 5, we will show that the set of events can be used to specified properties, which the system should not violate and that are checked at run-time.

3.10 Conclusion

In this Chapter we presented the set of events that was defined with our industrial partners Swisscom and Alcatel for the modelling of distributed applications built on top of a middleware. The set of events is defined at the middleware layer and thus, allows an abstraction that is independent of the implementation language. In other words, by changing the implementation language, the set of events modelling the application remains valid.

The abstraction level that is used for establishing the set of events is "abstract enough", as we will see in Chapter 5, to express properties using a formal language and "not too abstract" to model the final implementation.

In the next chapter we explain the method for monitoring, by the detection of the set of events that we defined in this Chapter.

Chapter 4

Monitoring

The contents of this Chapter were published in [LDKK98] and parts of it in [LDH98]

4.1 Chapter Overview

In this chapter, we present the architecture and method we developed in order to monitor a distributed application. The different elements of the architecture are described. The interworking of the different structural pieces of the architecture is also explained. A special section is devoted to the study of the intrusive effect of the monitoring approach adopted in this thesis.

4.2 Introduction

Today, distributed applications are built with the help of a Distributed Processing Environment (DPE). With such an environment, the specifier of the service does not need to deal with distribution issues when specifying the distributed application. The RPC mechanism or sophisticated socket mechanisms that ensure distributed features of the application are completely hidden from the programmer. Thus, the design of the distributed application consists of:

1. identifying the objects required for the application,
2. defining the interfaces to those objects,
3. implementing the different interfaces,
4. writing a server program that creates instances of the implementation classes,
5. writing a client program that accesses the server objects.

The development of a distributed application that exhibits a desired functionality is a complex task. And, when the application has to guarantee specific performances this problem becomes even more complicated. Therefore, the construction of *monitoring* tools to assist in the development and testing steps of complex distributed

application is of practical significance. Monitoring is an essential means for obtaining the information required about the components of a distributed application, their communications, their computations, their state variables, etc.

Monitoring can be used for various purposes [MSS94]: debugging, testing, program visualization and animation, general management activities (performance management, configuration management, fault management, security management, etc.). Much work has been devoted to the monitoring of distributed applications as presented in the monitoring survey by Mansouri-Samani and Sloman [MSS94]. Less work on the subject has been accomplished since the emergence of middleware platform such as CORBA.

The monitoring approach that we present in this chapter is specific to distributed applications built on top of an object-oriented middleware platform. The proposal we developed is generic enough to be applied on any such middleware platform; nevertheless, it is specific enough to capture the needed and useful information for the use of the monitoring application.

4.3 Monitoring Background

4.3.1 Definitions

Joyce, et al. give a definition of monitoring [J+87]:

> *The Monitoring of distributed systems involves the collection, interpretation, and display of information concerning the interactions among concurrently executing processes.*

The monitoring approaches can be classified as *software, hardware* and *hybrid.* The principal difference is that the hardware approach separates the monitoring task from the target system's work load, whereas the software approach mixes them together. Hybrid monitoring is situated in between the two others.

4.3.2 Software Monitoring

The software monitoring system consists of an instrumentation software that is part of the system to be monitored. The instrumentation is added into the code by the means of an instrumentation software. The generated events of interest are detected, as well as handled, by the system to be monitored. As a result, the instrumented program has an execution speed penality, which is referred to as monitoring intrusion. The steps for software monitoring are:

1. The system to be monitored is instrumented by the instrumentation software.
2. The instrumented program is executed and thus generates events.
3. The events are detected and processed by the monitoring software.

The main disadvantage of software monitoring is intrusion. As the instrumentation is part of the system to be monitored, it will intrude into the system processing time and memory space. This intrusion can change the event ordering. The advantage of the software monitoring is that is does not require extra hardware and thus is very flexible.

4.3.3 Hardware Monitoring

The monitoring hardware consists of the monitoring hardware and control module and is separated from the system to be monitored. The execution of the target programs is monitored directly via the monitoring hardware, which is connected to the system buses without any embedded instrumentation code. Thus the monitoring hardware catches each message that passes through the bus. The different steps for hardware monitoring are as follows:

1. The hardware monitoring equipment is connected to the system to be monitored.
2. The monitoring hardware modules and systems to be monitored are executed.
3. The events are detected and processed by the monitoring hardware modules.

The advantage of the hardware monitoring is that it does not share any computational resources. Therefore, it causes no monitoring intrusions. On the other hand, it necessitates an extra hardware equipment.

4.3.4 Hybrid Monitoring

In hybrid monitoring, both software and hardware are used to perform monitoring activities: software is used to specify the event to be detected in conjunction with hardware devices in order to identify and process those events. In this approach, the instrumentation code are executed within the system to be monitored, while event detection and processing are performed by separate monitoring hardware.

4.4 Our Monitoring Approach

The distributed systems that we are interested in monitoring consist of applications developed on top of a middleware (see Chapter 2). With the use of middleware all the communication issues are hidden from the developer. And thus, the developer might not be aware of the (proprietary) protocol or message format of a specific middleware technology. Using a hardware monitoring approach in such a case is extremely difficult. For example, it is easy to intercept packets transmitted from one workstation to another, but it is almost impossible to reconstruct the message (e.g., from the transport layer to the application layer) and thus the events from these packets.

Therefore we prefer to develop a software monitoring approach. Software monitoring is flexible and fits well in the usual development process. *The design of a software monitoring mechanism consists in finding an instrumentation method that is able to detect the needed events and is as unintrusive as possible.*

In [MSS93], Mansouri-Samani and Sloman present a functional model of monitoring in terms of the generation, processing, dissemination, and presentation of information. This functional model can help design a monitoring mechanism appropriate for the facilities needed for its purpose (debugging, testing, managements, etc.). They identify the different elements of a monitoring model:

1. Generation of monitoring information: Important events are detected, and event and status reports are generated. These monitoring reports are used to generate monitoring traces, which represent historical views of system behavior.
2. Processing of monitoring information: A generalised monitoring mechanism provides common processing functionalities:
   - Merging of traces and multiple trace generation
   - Validation of monitoring information
   - Database updating
   - Combination, correlation and filtering of monitoring information.
   - Analysis
3. Dissemination of monitoring information: Monitoring reports are distributed to entities who require them.
4. Presentation: gathered and processed information is displayed to the users in an appropriate form.
5. Implementation issues: The way the monitoring mechanism is designed can imply different intrusiveness properties (software, hardware or hybrid monitor). The use of clock synchronization or logical clocks exhibits the use of different algorithms or protocols.

In the following sections, we will show how we designed the monitoring mechanism, corresponding to the above criteria.

For the design of the distributed application, different languages are available to choose from: IDL (Interface Definition Language) [COR97, OMG98], which is specific to CORBA and ODL (Object Description Language), which is specific to the Telecommunication Information Networking Architecture (TINA) [KLMR95]. ODL can be considered as a super set of IDL, but is not as common as IDL.

These different languages of specification make no assumptions about the implementation details. The application can be implemented in C++, JAVA, Smalltalk, etc. But thanks to the IDL specification, all the operations will be present (implemented) in either implementation language. This interesting property allows us to propose a method independent of the implementation language.

Consider Figure 4.1 for an overview of the development process of distributed applications in the CORBA framework. The white boxes depict the normal devel-

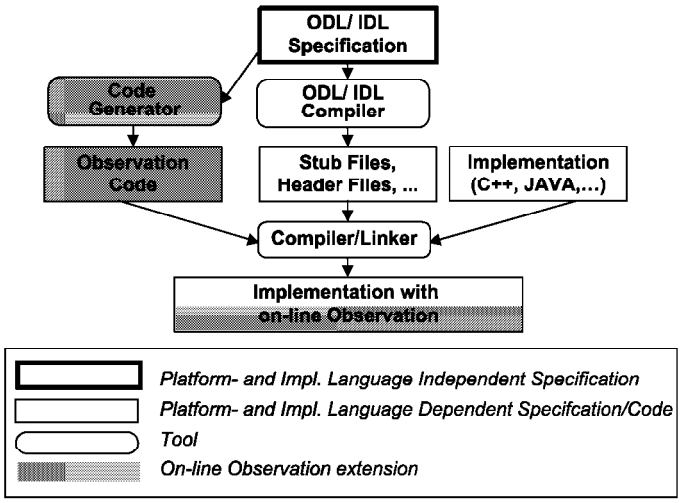


Figure 4.1: Global framework

opment process of distributed applications; the gray boxes describe the extensions we propose in order to allow the automatic instrumentation for run-time monitoring. Rounded boxes denote tools.

The IDL specification of the interfaces is passed to an IDL compiler associated with the DPE which generates stub code and header files that are then linked to the actual implementation code thereby shielding the developer of the distributed application from the difficult task of handling the distribution issues.

In addition to passing the IDL specifications to the IDL compiler, the *code generator* is fed with the IDL specifications. This code generator tool generates some generic observation code, which also needs to be linked to the actual implementation, and it forms the on-line observer part of the implementation. This process of introducing features that facilitate the testing process is frequently referred to as part of a Design For Testability process [DKPR96]. In [CG94], Chechik and Gannon propose annotating C code in order to check whether the code satisfies the requirements. Sankar [San89] and Mandal [SM90] annotate ADA programs with a formal specification language in order to compare the run-time behavior against its formal specification while the program is executing. The monitoring of applications based on the manual instrumentation of the code can be very efficient for the developer of the application, but rather inappropriate for a tester who does not have a full understanding of the implementation. This tester would have to understand all (at least part of) the code to figure out how and where to instrument it. Manual instrumentation is therefore currently an ad hoc solution for specific cases, but is not systematic and methodical enough to be applied in general cases. In our proposal, the system of trace generation is automatically added by the code generator without the need for understanding or manually instrumenting the code of the distributed application.

The traces are collected by an *Observer*. The interface of the Observer is specified and normalized, and then the Observer is implemented once and for all. By using its interface, and the standardized protocol GIOP (General Inter-ORB Proto-

col) [OMG98] it is irrelevant if the implementation of the Observer is in the same implementation language as the distributed application or not. The Observer can be used in any system as long as the object of the system uses its interface. This means that the code generator introducing the trace generation mechanism, which is used to send *notifications* to the Observer via its interface, is the only piece that depends on the implementation language used for developing the application.

The observable information (list of objects, operations and attributes) that the user can select from is provided to the Observer by parsing the IDL specification with an appropriate tool. The user has only to select what to observe, then the remaining work, including the instrumentation, is done automatically.

Unlike other approaches (e.g., [AAM97, SM90]) the definition of the different monitorable information is automatically done by the compilation process by analyzing the IDL specification. For example, each operation and attribute[1] in the specification will be associated with observable events (e.g., out_ request, in_request, out_reply, in_reply); each object implies two events (creation and deletion).

The Observer is notified of each creation and deletion of an object active in the system and assigns an Id to each object. The Observer maintains a table with the list of objects currently existing in the system and their respective Ids. This table is used to perform selective monitoring, according to the selections of the user.

In Section 4.5, we present the Observer, which collects the information gathered by the instrumentation code and display it to the user. In Section 4.6, the scenario for monitoring is explained. The Section 4.7 is devoted to the presentation of the instrumentation: How the instrumentation code is added? How the events are detected? What are the notifications sent by the instrumentation to the Observer?. In order for the Observer to have a consistent view of the behavior of the system, it is necessary to guarantee that the ordering of the notifications received is correct (Section 4.8). The intrusive effect of the instrumentation is analyzed in Section 4.9.

4.5 The Observer

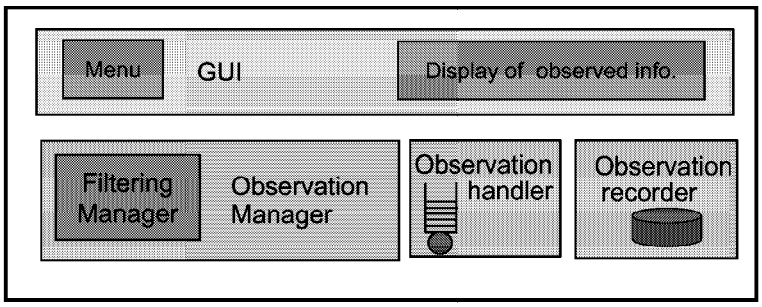


Figure 4.2: Structure of the Observer

[1] Two operations are automatically defined to act on the attributes defined in the specification: the *set* and *get* operations.

Figure 4.2 illustrates the global design of the Observer. The Observer is made up of four modules:

- The **Graphical User Interface** provides the user with a list of objects and operations of the system (a list of events that might happen). Through the GUI the user selects what to observe and the GUI hands the selection to the Filtering Manager. The GUI also displays all the notifications received.
- The **Observation Manager** makes the interface between the system under scrutiny and the Observer. It receives all the notifications from the instrumentation and also, when needed, parameterizes the instrumentation.

  Within the Observation Manager, the Filtering Manager is in charge of activating and parameterizing the filters (instrumentation code used to detect events (Section 4.7))involved by the selections made by the user. This is done by invoking an operation on all the objects in which selected events are occurring.
- The **Observation Handler** receives all the observed information from the Observation Manger and is responsible for the correct ordering of them. Based on a timestamp algorithm (Section 4.8), it reorders the notifications and gives them to the GUI and The Observation recorder.
- The **Observation recorder** provides the functionality of storing notifications in a database. This database could then be used to analyze a crash, a special behavior or to replay an execution trace. The Observation recorder is an optional part of the Observer.

4.6 Monitoring Scenario

Let us assume that the generic instrumentation code has been added into the implementation of the distributed application. The scenario of run-time monitoring is sketched in Figure 4.3.

The user first selects the events that he wants to monitor. The events available to monitor are given to the user by means of menus (the way these menus are generated is explained in Section 4.7.2). This selected information is passed to the Filtering Manager (1). The Filters are first activated and parameterized (2) according to the selections of the user. Then, at run-time the notifications are sent to the Observer (3) whenever an observable event (i.e., an event that has been selected for monitoring) occurs in the computation. At run-time the user can change the configuration of the Monitoring system by selecting other objects/operations to monitor. The Observation Manager receives the notifications and passes them (4) to the Observation Handler that guarantees a consistent order (5), by applying a specific algorithm, of the information. The Observation Handler gives the information to the Observation Recorder (6) and to the GUI (7) as requested by the user. The monitored information is then displayed for the user in a kind of time line diagram, updated at runtime.

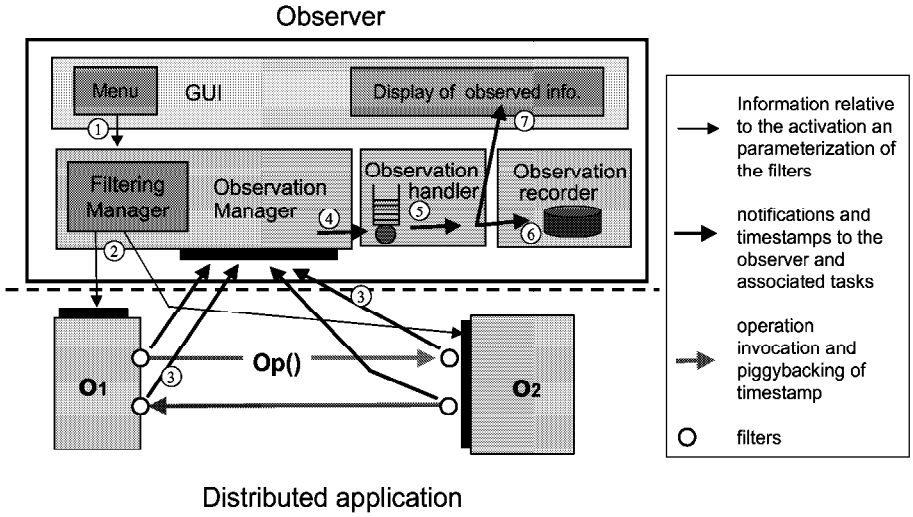


Figure 4.3: Scenario of run-time monitoring

4.7 Instrumentation

In order to monitor the events occurring in the system, it would be useful to be able to interpose a mechanism on the invocation (and response) path(s) between a client and a server. In the CORBA specification, there is a standardized notion that can be used for precisely this purpose: *the interceptors* [OMG98, OMG99].

The Orbix (from IONA) implementation of CORBA rovides the Interceptors by means of a *filtering* mechanism. A filter allows a programmer to specify that an additional code is to be executed before or after the normal code of an operation. The Orbacus OOC distributed platform will soon offer a mechanism similar to Orbix filters, termed *interceptors*. In our work we make the assumption that a run time observation mechanism, that we term with the generic word '*Filter*'[2], is provided by the distributed platform. This assumption is not restrictive. In the case where such a mechanism is not offered by the distributed platform, a proxy object that plays the role of the Filter can be added for each object within the system. Such work is performed in the IRI project [AAM97].

We use the "breakpoints" provided by the Filters to send notifications to the observer. With this, there are different approaches for achieving run-time observation:

- The Filters are put everywhere (around every object and every process) and all the information passing through the Filters is collected. The relevant events are masked out and the unneeded information discarded. This approach creates a huge overhead because all information is sent to the Observer and heavily influences the system. But, it gives us a complete idea about what is going on.

[2]Note that our Filtering is not the same as the definition given in [Per94]: 'The removal of irrelevant information before processing begins'

- Filters are put everywhere but activated and parameterized only as needed. This would allow the selection of specific information to observe, without requiring the program to be recompiled. However, in the case that the tester wants to remove the Filters completely, he still needs to recompile the code.

Our approach is based on the second description, because it appears to give the best compromise between flexibility and efficiency. With the help of the specification, the code generator adds observation code to the system in order to build parameterized filters. This generic code encompasses:

- the *notification mechanism.* This mechanism allows the monitored information to be sent to the Observer. This code is exactly the same for each object of the system. Details on the notification are given in Section 4.7.4
- a *selection mechanism.* This mechanism allows the user to select specific data to be monitored for each object. This selection mechanism is based on (Section 4.7.2) (i) operations `activate` and `deactivate` that enable or disable a specific filter; (ii) operations `select` and `unselect` that are offered to the user in order to choose what events to monitor, (iii) a database (table) containing all the data that can be monitored on the object, a special field is reserved for marking if a data has to be monitored. This field is updated by the `(un-)select` operation.
- some *specific* code to guarantee the causal dependences (order) of notifications (this is detailed in Section 4.8)

The different tasks of the instrumentations, whenever an event is detected, are depicted in Figure 4.4 and presented in the following pseudo-code:

```
IF the event is selected
THEN {
   Generate the notification;
   Handle the ordering consistency;
   Send the notification to the Observer;
 } ELSE break;
FI;
```

As seen in Figure 4.4 extra computations are executed each time an event occurs; this is the drawback of software monitoring. In order to minimize the intrusiveness of the instrumentation, we activate the Filters that detect events only when needed. For example, if no event to monitor occurs on an object, the filters of that object will not be activated and the intrusiveness is null. The second action to minimize the intrusive effect is to produce an efficient code for the implementation: do not execute unnecessary loops, unnecessary computations, send unneeded information. The impact of the instrumentation is analyzed in 4.9.

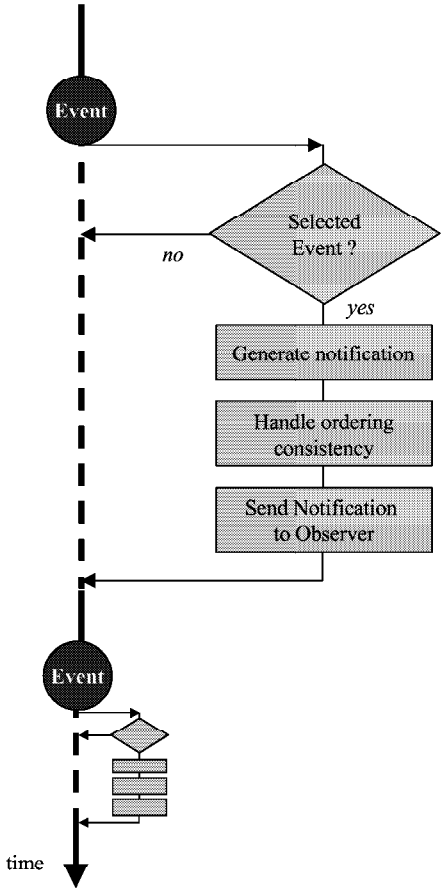


Figure 4.4: instrumentation tasks

4.7.1 How the Events are Detectable

In Table 4.1, the way the events can be detected is summarized. The events are defined generically. This means that the event model can, for example, be applied to any implementation of CORBA. Therefore, in Table 4.1, the first column is devoted to explain how events are detectable in a CORBA environment in general. The second column refined the description for a specific implementation of CORBA, Orbix^{TM} from IONA. Note that the unspecified entries require the use of some means beyond what CORBA/Orbix provides.

4.7.2 Selection Mechanism

The Selection mechanism is designed to reduce the impact of the instrumentation. Indeed, its role is only to activate the instrumentation for the needed information. Better than post-processing all the information and sorting the useful information, it allows the direct selection of the interesting information at the source. This handle at the source guarantees a lower network bandwidth use.

The Selection mechanism is composed of two parts. The first handles the activa-

Table 4.1: How events are detectable

Event	Description	CORBA	Orbix^{TM}-MT
o_outReq	outgoing operation request	Interceptor	Filter
o_inReq	incoming operation request	Interceptor	Filter
o_outRep	outgoing operation reply	Interceptor	Filter
o_inRep	incoming operation reply	Interceptor	Filter
p_inReq	incoming operation request	Interceptor	Filter
p_newO	object creation	-	-
p_delO	object deletion	-	-
p_newT	thread creation	-	Filter
p_delT	thread deletion	-	-
p_reqRef	request for an object reference	ORB Message	Operation `bind`
p_recRef	receive of an object reference		Operation `bind`
t_assT	thread assignment		Filter
t_relT	thread release		-
t_outReq	outgoing operation request		Filter
t_outRep	outgoing operation reply		Filter
t_inRep	incoming operation reply		Filter
s_oReg	object registration	ORB Message	Op. `Implementation_is_ready`
s_oDereg	object deregistration	-	timeout
s_newP	process creation	ORB Message	Message "New connection"
s_delP	process deletion	ORB Message	Message "End connection"

tion and deactivation of the Filters, according to the event to monitor. For example, the Filters on objects, where events to monitor will not occur, can be deactivated. The second consists in parameterizing the Filters in order to catch only the needed information. The Activation and deactivation of the Instrumentation is presented in Section 4.7.2.1, the parameterization of the instrumentation is the subject of Section 4.7.2.2.

4.7.2.1 Activation and Deactivation of the Instrumentation

The primary goal of software monitoring is to monitor all the information that is needed/useful, the second goal is to disturb the execution under scrutiny as little as possible. Toward the first goal we use a set of events (see Chapter 3). For the second goal our proposal is to design the instrumentation in a way that it can be activated and deactivated as needed.

This can be performed by the addition of a function in each interface of the system. In each IDL of the system, we add a method called `activate` that has a parameter `state` that is *true* in order to activate the Filter or is set to *false* in order to deactivate the Filter. When the user wants to monitor a specific event, then the Observer simply invokes the method `activate` on the specific object.

In [DLKH98b] and in Chapter 3, we presented a set of events that can be used to model distributed Object-oriented systems. Having the `activate` function that we presented above, we must define which filter has to be activated for each event, let us define a few functions that will help us.

Let us introduce a function called π, that gives the process id of the process in which a given event occurs (i.e, which gives the process id in which a Filter has to activated to detect the event). Let E be the set of all possible events, as summarized in Table 3.1. Let PID be the set of all processes id of a running application:

To define the function π we introduce two other functions θ and ω, that give the process id in which a thread, respectively an object is executed:

The function θ:

$$\begin{aligned} \theta: \ TID &\longmapsto PID \\ tid &\longmapsto pid_{tid} \end{aligned}$$

where pid_{tid} is the process id for a thread with the thread identifier tid [DLKH98b].

We also introduce a new function ω:

$$\begin{aligned} \omega: \ OID &\longmapsto PID \\ oid &\longmapsto poi_{oid} \end{aligned}$$

where poi_{oid} is the process id for an object with the object identifier oid.

The function π is defined as follows:

$$\begin{aligned} \pi: \quad E &\longmapsto P \\ e &\longmapsto pid_e \end{aligned}$$

where pid_e is the process on which the given event occurs as defined in Table 4.2.

Table 4.2: π function

Event e	$\pi(e)$
$(o\_outReq, (oid_1, oid_2, op, *))$	$\omega(oid_1)$
$(o\_inReq, (oid_1, oid_2, op, *))$	$\omega(oid_2)$
$(o\_outRep, (oid_1, oid_2, op, *))$	$\omega(oid_2)$
$(o\_inRep, (oid_1, oid_2, op, *))$	$\omega(oid_1)$
$(p\_inReq, (oid_1, oid_2, op, *))$	$\omega(oid_2)$
(p_newO, oid)	$\omega(oid)$
(p_delO, oid)	$\omega(oid)$
(p_newT, tid)	$\theta(tid)$
(p_delT, tid)	$\theta(tid)$
$(p\_reqRef, tid, cn, pn, *, *)$	$\theta(tid)$
(p_recRef, tid, oid)	$\theta(tid)$
$(t\_assT, tid, (oid_1, oid_2, op, *))$	$\theta(tid)$
$(t\_relT, tid, (oid_1, oid_2, op, *))$	$\theta(tid)$
$(t\_outReq, tid, (oid_1, oid_2, op, *))$	$\theta(tid)$
$(t\_outRep, tid, (oid_1, oid_2, op, *))$	$\theta(tid)$
$(t\_inRep, tid, (oid_1, oid_2, op, *))$	$\theta(tid)$
(s_oReg, oid)	N/A
(s_oDereg, oid)	N/A
(s_newP, pid)	N/A
(s_delP, pid)	N/A

Let us take an example, of a system with an object oid_1 and second object oid_2 and the event $(o\_outReq, (oid_1, oid_2, op, *))$. In the case of a system with only Filters on processes, the first row of the Table 4.2 tells us that the Filter that has to be activated is the one of the process $\omega(oid_1)$ that is the process in which oid_1 is running.

In the case of Filters for Objects and Filters for processes. The Filters on the Objects can be used for the four Object events: $(o\_outReq, (oid_1, oid_2, op, *))$ and $(o\_inRep, (oid_1, oid_2, op, *))$ require the Filter of oid_1 to be activated, $(o\_inReq, (oid_1, oid_2, op, *))$ and $(o\_outRep, (oid_1, oid_2, op, *))$ require the Filter of oid_2 to be activated.

4.7.2.2 Parameterization of the Instrumentation

The Parameterization of the instrumentation allows the Observer to tell a Filter which event it should report. For example, in the case of a Filter "around" an object, the

Filter is able to intercept all the incoming and outgoing requests, but the Observer is only interested in the outgoing event of a specific request.

Each Filter is thus equipped with a database (which might be as simple as a list) storing the events to report to the Observer. The content of the database for the selection is different for each object.

In order to update this database, there is the `select` operation. This operation has as parameter the list of events to monitor. An example is, if the `op_1` has to be monitored, the *select* operation invoked by the Observer on the object oid_1, after the user has selected the data to monitor by means of a menu, would be :

```
select(o_outReq(oid_1,*,op_1,*));
```

A big advantage of the `select` operation is that the configuration of the monitoring can change at run-time. The user can select a new data to monitor at run-time without the need to recompile the code or to add new instrumentations in the code. Our monitoring instrumentation is *parameterizable at run-time*. Similar to the *select* operation, an *unselect* operation is implemented.

4.7.3 Automatization of the instrumentation

The monitoring of applications based on the manual instrumentation of the code can be very efficient for the developer of the application, but rather inappropriate for a tester who does not have a full understanding of the implementation. Manual instrumentation is therefore currently an ad hoc solution for specific cases, but is not systematic and methodical enough to be applied in the general cases.

In our approach, it is possible to use a *code generator* in order to proceed to the instrumentation of the code. The code generator is responsible for:

1. Adding the Filter in the system under scrutiny,
2. Parsing the IDL specification to extract the operations, class name, attributes, etc. in order to offer menu to the user to select the events to monitor,
3. Add the `activation`, `select` and `unselect` operation in the system.

The code generator is dependent on the middleware that is used. It depends on the Filters that are used and how they are implemented.

4.7.4 Notifications

Our monitoring approach is based on the detection of occurring events and the reporting to the Observer. This reporting is made of a notification mechanism. A notification is an operation invoked on the Observer.

With the concept of polymorphism, it is easy to have an overloaded function

notify that accepts as first parameter the event type, and then the corresponding parameter. The IDL definition of the notify functions is given hereafter:

```
#include "type_definition.idl"
\\definitions of the different types used.

interface Notification {
  oneway void Notify( in o_outReq_type event,
                      in obj_id  oid1,
                      in obj_id  oid2,
                      in string  opName,
                      in paramaterList_type _parameterList );
  oneway void Notify( in o_inReq_type event,
                      in obj_id  oid1,
                      in obj_id  oid2,
                      in string  opName,
                      in paramaterList_type _parameterList );
  oneway void Notify( in o_outRep_type event,
                      in obj_id  oid1,
                      in obj_id  oid2,
                      in string  opName,
                      in paramaterList_type _parameterList );
  oneway void Notify( in o_inRep_type event,
                      in obj_id  oid1,
                      in obj_id  oid2,
                      in string  opName,
                      in paramaterList_type _parameterList );
  oneway void Notify( in p_inReq_type event,
                      in obj_id  oid1,
                      in obj_id  oid2,
                      in string  opName,
                      in paramaterList_type _parameterList );
  oneway void Notify( in p_newO_type event,
                      in obj_id  oid );
  oneway void Notify( in p_newT_type event,
                      in thread_id  tid );
  oneway void Notify( in p_delT_type event,
                      in thread_id  tid );
  oneway void Notify( in p_reqRef_type event,
                      in thread_id  tid,
                      in string className,
                      in string processName);
  oneway void Notify( in p_recRef_type event,
                      in thread_id  tid,
                      in obj_id  oid );
  oneway void Notify( in t_assT_type event,
```

```
                        in thread_id  tid,
                        in obj_id  oid1,
                        in obj_id  oid2,
                        in string  opName,
                        in paramaterList_type _parameterList );
    oneway void Notify( in t_relT_type event,
                        in thread_id  tid,
                        in obj_id  oid1,
                        in obj_id  oid2,
                        in string  opName,
                        in paramaterList_type _parameterList );
    oneway void Notify( in t_outReq_type event,
                        in thread_id  tid,
                        in obj_id  oid1,
                        in obj_id  oid2,
                        in string  opName,
                        in paramaterList_type _parameterList );
    oneway void Notify( in t_outRep_type event,
                        in thread_id  tid,
                        in obj_id  oid1,
                        in obj_id  oid2,
                        in string  opName,
                        in paramaterList_type _parameterList );
    oneway void Notify( in t_inRep_type event,
                        in thread_id  tid,
                        in obj_id  oid1,
                        in obj_id  oid2,
                        in string  opName,
                        in paramaterList_type _parameterList );
    oneway void Notify( in s_oReg_type event,
                        in obj_id  oid );
    oneway void Notify( in s_oDereg_type event
                        in obj_id  oid );
    oneway void Notify( in s_newP_type event
                        in process_id  pid );
    oneway void Notify( in s_delP_type event
                        in process_id  pid );
};
```

There are two other notifications termed, respectively, as *Creation* and *Deletion* and denote the creation and deletion of objects. The creation and deletion of an object can be observed by adding (via the code generator) an instruction for notification in the constructor and destructor of the objects, or by spying on the messages given by the DPE. *Creation* contains the name and type of the object created and returns its Id given by the Observer. *Deletion* encompasses the Id of the object deleted. This Id is the one that the Observer assigned to the object earlier, just after its creation.

```
void Creation( in  string  ObjName,
               out obj_id  oid  );
oneway void Deletion( in  obj_id oid  );
```

The approach used for the notification is *asynchronous* reporting, or otherwise called oneway function. The use of *synchronous* reporting might also be possible. Such a reporting mechanism blocks each object until the result of the notification is received. Thus, the delay introduced also depends on the processing rapidity of the Observer. This means that synchronous reporting can accentuate the impact of monitoring on the system by slowing down the entire system as depicted in Figure 4.5.

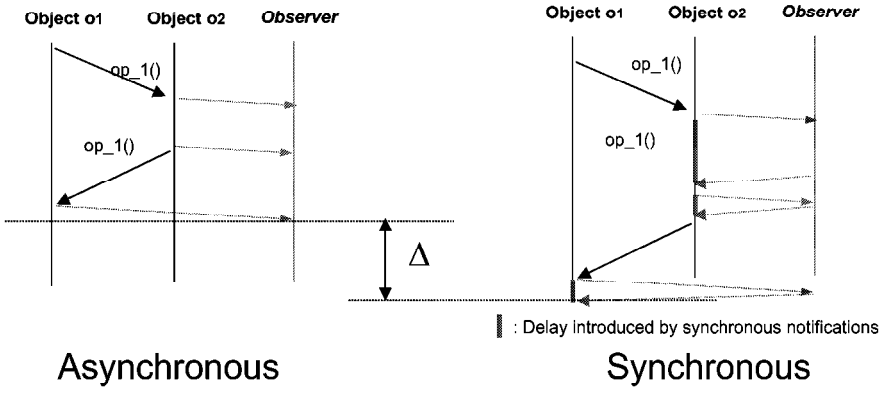


Figure 4.5: Comparison of the impact of asynchronous and synchronous reporting

4.8 Ordering Consistency

4.8.1 Ordering Problem

In general, we cannot assume that the DPE will guarantee that the Observer will receive the notifications in the same order in which they have been sent. Therefore, the establishment of the "true" order of the events is mandatory in order to be able to follow what happened in the system. For this, we rely on the Lamport's *happened-before* relation. Lamport's happened-before relation [Lam78] between two events in a distributed system can be described as follows:

The activity of a process P_i is perceived as a set of local atomic events E_i, totally ordered by a local precedence relation $<_i$. This set E_i can be partitioned into two subsets:

- I_i: the set of internal events of P_i (resulting from internal actions);
- X_i: the set of communication events of P_i (send and receive events).

The set $E = \bigcup_i E_i$ of all the events produced by the distributed execution is partially ordered by Lamport's relation called *happened-before* or *causal precedence*, denoted

by $\rightarrow$:

$$\forall x \in E_i, y \in E_j \; . \; x \rightarrow y \; =_{def}$$
$$\begin{cases} & i = j \text{ and } x <_i y \\ or & \\ & x \text{ is sending of a message \& } y \text{ its reception} \\ or & \\ & \exists z \; . \; x \rightarrow z \text{ and } z \rightarrow y \end{cases}$$

Let us consider a system of two objects o_1 and o_2, assumed to be already created, in which o_1 invokes the operation $op()$ on o_2 as depicted on the timeline of Figure 4.6. According to the definition of the *happened-before* relation, Figure 4.7 gives the causal dependencies of the events[3].

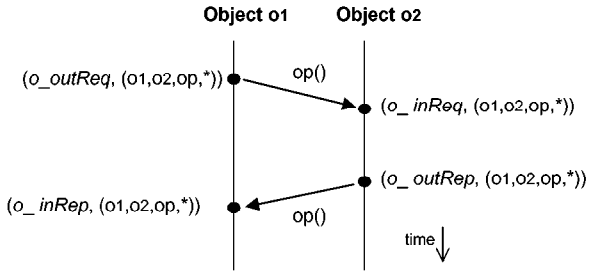


Figure 4.6: Timeline of the invocation of an operation

$$\begin{array}{lcl} (o\_outReq, (o_1, o_2, op, *)) & \rightarrow & (o\_inReq, (o_1, o_2, op, *)) \\ (o\_inReq, (o_1, o_2, op, *)) & \rightarrow & (o\_outRep, (o_1, o_2, op, *)) \\ (o\_outRep, (o_1, o_2, op, *)) & \rightarrow & (o\_inRep, (o_1, o_2, op, *)) \end{array}$$

Figure 4.7: Causal dependencies of events of Figure 4.6

The *happened-before* relation also allows to determine which events are concurrent. Two events ε_1 and ε_2 are said concurrent, $\varepsilon_1 || \varepsilon_2$, iff: $\varepsilon_1 \not\rightarrow \varepsilon_2$ and $\varepsilon_2 \not\rightarrow \varepsilon_1$.

The order of these events as seen by the Observer (via the notifications) must reflect the order of events as defined by the happened-before relation. In order to catch causal dependencies, we could try to synchronize the different clocks of each process involved in the distributed computation. However, the different distributed objects of our system do not share a global memory and communicate only by operations (two-way or one-way). Distributed systems may not have built-in physical time and can only approximate it. The Network Time Protocol (NTP) proposed within the Internet community [Mil90] allows clocks to be synchronized to a few tens of milliseconds. We might use such a synchronization system. But, even with the best synchronization algorithm, it is not possible to know, by looking at a global clock, if some events are missing. Therefore we are using another method that is explained hereater.

[3]The syntax of the events is explained in Section 3.4

The method to preserve causal dependencies is the use of *logical clocks*. A logical clock system associates a timestamp $\delta(x) \in \Delta$ to each event $\varepsilon \in E$ of the computation

$$\begin{aligned} \delta : E &\longrightarrow \Delta \\ \varepsilon &\longmapsto \delta(\varepsilon) \end{aligned}$$

All the logical clocks have to satisfy the consistency condition $\varepsilon_1 \rightarrow \varepsilon_2 \Rightarrow \delta(\varepsilon_1) < \delta(\varepsilon_2)$ stating that the clock never goes backwards. In order to be truly applicable, logical clocks have to satisfy stronger consistency condition: $\varepsilon_1 \rightarrow \varepsilon_2 \Longleftrightarrow \delta(\varepsilon_1) < \delta(\varepsilon_2)$. This strong condition avoids assuming an arbitrary ordering between unrelated events. For example, the linear clock based on scalar introduced by Lamport does not satisfy the strong consistency condition [Fid96]. However, vector [Fid96, Mat89] and matrix [RST91, Rug94] respect the strong consistency condition.

Logical clocks are implemented for each object by assigning a *timestamp* to events. This timestamp represents the logical clock and is advanced according to certain rules. The implementation of the timestamp mechanism requires deciding what data structure to use and then determining the set of rules to update the data structure. This data structure can either be a *scalar* [Lam78], a *vector* [Fid96, Mat89] or a *matrix* [RST91, Rug94]. The choice of one of these data structures is a trade-off between complexity and a number of interesting properties. For example, a matrix timestamp exhibits more powerful properties but also necessitates more processing power to be updated and handled [RS96].

The system of logical clocks is used to preserve the *causal ordering* of our notifications. In simple words, causal ordering is the natural extension of FIFO ordering in the case of more then one sender and receiver. A system respecting the *causal ordering* [RST91] [SES89] requires that the messages are delivered respecting the happened-before relation: an object receiving a request op_2 can know, thanks to the logical clock, if a request op_1, which has been sent before, has not been received. In this case, it delivers request op_2 only after op_1 has been received and proceeded.

Raynal, Schiper and Toueg present in [RST91] a simple way, called the *RST* algorithm, to implement a system respecting the causal ordering. For this, they use matrix timestamps associated with a vector at each object.

The algorithm we use to maintain the causal ordering of notifications is based on RST. Our algorithm is only used to ensure that the causal ordering can be reconstructed at the Observer site. Each object obj_i of the system is associated with a vector timestamp. Each element of the timestamp is a pair $(obj_{Id}, \#events)$, where obj_{Id} is the Id of one object of the system and $\#events$ is the number of notifications sent by object obj_{ID} to the Observer as known by obj_i.

The number of notifications sent by each object is propagated through the system by piggybacking the timestamp with the requests. The size of the timestamp is equivalent to the number of objects in the system. As the number of object in the system evolutes (depending on the creation and deletion of objects) our algorithm must be capable of maintaining a dynamically sized timestamp. The algorithm is detailed in next Section.

4.8.2 Reordering Algorithm

Our algorithm for causal ordering at the Observer site is based on the RST algorithm and maintains different data structures. The Observer handles a vector $DELIV[n_{Obs}]$ where n_{Obs} is the total number of objects in the system. This number changes whenever an object is created or deleted. The Observer detects or is notified of each creation or deletion (See Section 4.7.4). $DELIV[i]$ denotes the number of notifications delivered to the observer by object O_i[4]. Initially, each element of $DELIV$ is set to 0. The Observer also has a queue in which it stores the notifications that cannot be delivered due to missing causal dependent messages. All the other objects O_i of the distributed system maintain a vector $SENT_i[n_i]$, where n_i is the number of components in the system as seen by O_i. $SENT_i[j]$ denotes the knowledge that O_i has about the number of notifications sent by O_j to the Observer. Initially, all elements of the different $SENT$ vectors are set to 0. The algorithm used to maintain causal ordering at the Observer site is then explained. ('PI' and 'pi' stand for piggybacked data)

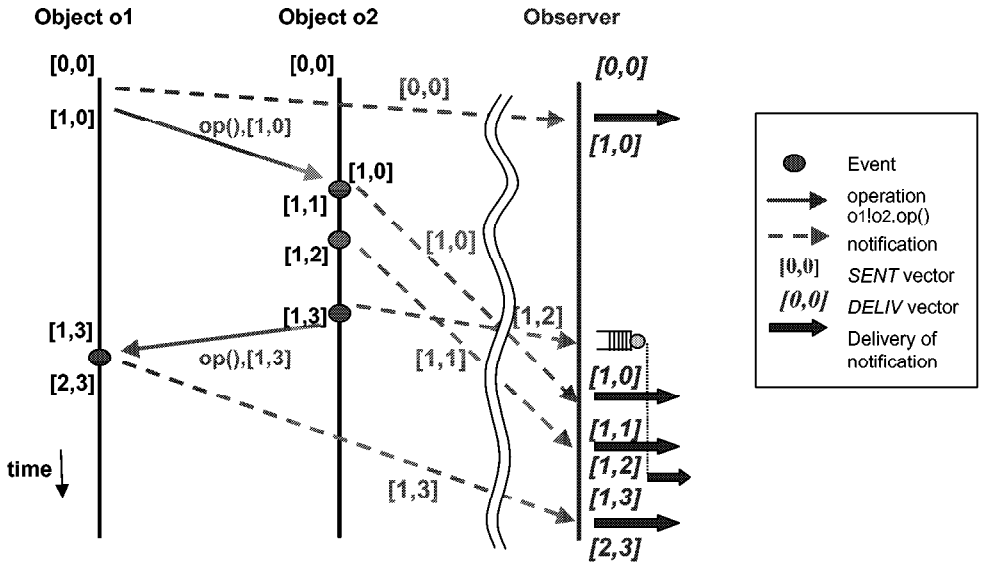


Figure 4.8: Adaptation of the RST algorithm in our framework

Emission of a *Notification* from O_i:

1. $if(event == inReq \ or \ event == inRep) \ then$
   $n_i := max(n_i, n_{pi})$;
   $for \ k, 1 \leq k \leq n_i, \ do \ (SENT_i[k] := max(SENT_i[k], PI[k])$
   $endif$
2. $send\_notification(Notification, n_i, SENT_i)$
3. $SENT_i[i] := SENT_i[i]{+}{+}$

[4]We make a difference between the reception of a request and its delivery

4. *if*(*event == outReq or event == outRep*) *then*
   piggyback n_i and $SENT_i$ with the request
   endif

Reception of a Notification from O_i at the observer site *Obs*:

1. *if* notification of type *Notify then* queue *Notification Until*(*forall* $k, 1 \leq k \leq n_{obs}$, $DELIV[k] \geq PI_i[k]$))
   endif,
   if notification of type *Creation then* send back *id* and n_{Obs}, update the list of Objects of the system
   endif,
   if notification of type *Deletion then* multicast *id* of object deleted and update the list of Objects of the system
   endif,

2. delivery of the notification to *Obs*
3. *if* notification *not* of type *Deletion then* $DELIV[i] := DELIV[i] + 1$ *endif*

Reception of a *multicast Deletion* from *Observer* at object O_i:

1. remove the row corresponding to the *id* received in $SENT_I$
2. add *id* in the list of deleted objects.

Reception of a *Creation*(give id) from *Observer* at object O_i:

1. store *id*
2. create a vector $SENT_i$ of size n received.

In Figure 4.8, our algorithm that respects causal ordering of notification at the Observer site is shown. This algorithm consists mainly in applying the RST algorithm only at the Observer site. The different objects O_i in the system do not have a $DELIV_i$ variable. In Figure 4.9, dynamic handling of timestamps is depicted. In the algorithm presented, the handling of the creation is easy, if a component receives a piggybacked timestamp of a bigger size then the one it handles, the size is just updated and the needed number of rows is added. In the case of a deletion, the problem is more complicated. Only the Observer will notice the deletion of a component O_i. Therefore it has to inform the components that O_i has been deleted. This can be done by a multicast message to all components containing the *id* of the component deleted. Each component O_i will then put this *id* in a list $DELETE_i$. Let us examine the use of this list $DELETE$. Assuming that (1) O_d has been deleted,(2) the Observer has sent the multicast message and that (3) O_i has received the message from the Observer and therefore already updated the size of $SENT_i$. If O_j invokes an

operation on O_i before having received the message from the Obs, the piggybacked timestamp will be of bigger size that the one of O_i. Therefore, O_i could believe that a new component has been added. This is illustrated in Figure 4.9 at the 'in Req' of $op_3()$ at o_2: $n_{pi} = 3$ whereas $n_2 = 2$. By refering at $DELETE_i$, O_i will know that O_d has been deleted.

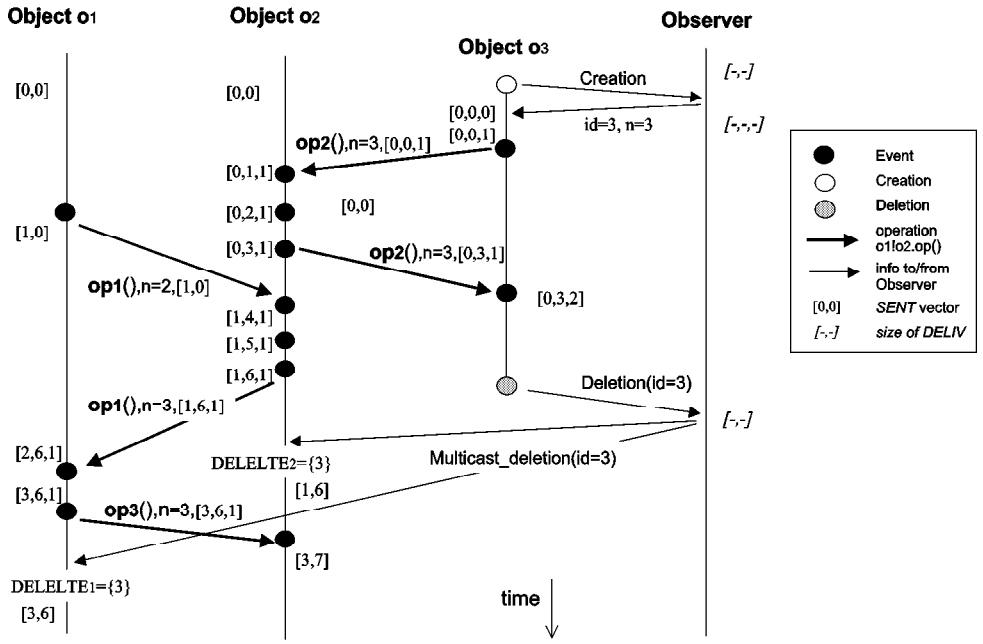


Figure 4.9: Our causal ordering algorithm

4.8.2.1 Optimization of the Reordering Mechanism

In the vector timestamp mechanism, the amount of information to be piggybacked is the size of n integers. In the case of a matrix clock, the size is of n^2 integers. When the number of objects is large, the timestamp represents a huge overhead.

Singhal and Kshemkalyani [SK92], Fowler and Zwaenepoel [FZ90] and, Jard and Jourdan [JJ94] all propose methods to optimize the implementation of vector clocks (this methods can also be used to optimize matrix clocks).

Singhal and Kshemkalyani propose a differential technique where instead of piggybacking the entire timestamp, only the entries of the vector clock that have changed since the last message are piggybacked. This technique cuts down on the communication bandwidth. However, a process needs to maintain two additional vectors to store the information regarding the clock values at the time of the last interaction with other processes. It requires also that the communication channels are first-in, first-out.

Fowler and Zwaenepoel and, Jard and Jourdan propose a method that does not maintain vector clocks on the fly. The method is efficient for removing piggybacking overhead but is not useful in our case because the correct ordering has to be computed

off-line.

4.9 Intrusive Effects of Monitoring

When monitoring a system, it is essential to know if a monitoring mechanism changes the behavior of the system under scrutiny. Furthermore, if the observed information is used for testing purposes, the evaluation of the intrusiveness of the monitoring mechanism is crucial. During the development of our monitoring and testing mechanism two solutions were adopted in order of minimizing the intrusive effect: asynchronous reporting for the notifications and the dynamic activation of the filters. The intrusive impact can be analyzed according to two ways: *theoretically*, by formalizing the problem and identifying properties of our system and, *experimentally*, by performing measurements in an implemented system, comparing the "pure" system against its modified version for monitoring/testing. The theoretical Analysis is presented hereafter. The experimental analysis is presented in Chapter 6.

4.9.1 Theoretical impact

In [HB89], Helmbold and Bryan demonstrated the possibility of identifying and using "intrusiveness" properties to design a monitoring mechanism. These properties give a formal framework to evaluate the intrusive effect of the monitoring approach chosen. For this purpose, they identify different sets of the possible computations of an application: the set of unmonitored (uninstrumented) computations, the set of monitored (instrumented) computations, and the set of observation (based on the monitoring mechanism) computations. Each element of these sets is an ordered set of the events occurring in a computation. Helmbold and Bryan identify properties by comparing the different elements of these sets. Their properties are useful to qualify the response to questions such as: "Among all possible computations, is there a computation that corresponds exactly to a given observation computation?". Unfortunately, in practice it is difficult to obtain the set of all different computations. Therefore, we prefer to focus on a single computation, and check, for example, if the instrumented computation and the observed computations are identical. To match our needs, we have adapted the intrusivness properties identified by Helmbold and Bryan.

Before describing the properties, let us define the different concepts needed. Let

- $\mathcal{C}$ be the set of possible computations of the application (Figure 4.10 (a)),
- $\mathcal{M}$, the monitored (instrumented) executions (Figure 4.10 (b)),
- $\mathcal{O}$ the observations produces by the Observer by the mean of the traces generated by the instrumentations (Figure 4.10 (c)).

In order to formally evaluate the impact of the monitoring mechanism, we introduce a notation to describe the partially ordered set (POset) representing the computations and observations.

A POset, (V, R), consists of two sets; V is a set of vertices and R is an asymmetric transitively closed relation on the vertices. In our case, V is a set of events performed and R a temporal relationship between the events, namely the partial order. The POset $C = (V_C, R_C)$ represents a particular computation in $\mathcal{C}$. When the monitoring mechanism is added onto the system, the set of possible computations resulting $\mathcal{M}$ can differ from $\mathcal{C}$. $M = (V_M, R_M)$ for a particular monitored computation. Each monitored computation produces a single observation denoted O and O_M denotes the observation produced by the monitored computation M. The set of events produced by a monitored computation can be categorized into two sets: the performable events in unmonitored computation and additional events performed as part of the monitored process. Given a particular monitored computation M, $V_{M,M}$ denotes the events of V_M which are part of the monitoring process. $V_{M,C}$ represents the performable events in an unmonitored computation. Thus, $V_{M,M} \cap V_{M,C} = \emptyset$ and $V_M = V_{M,M} \cup V_{M,C}$. $R_{M,C}$ denotes the set of edges $\{(v, v') : (v, v') \in R_M \wedge v, v' \in V_{M,C}\}$ where (v, v') denotes a directed edge from v to v'.

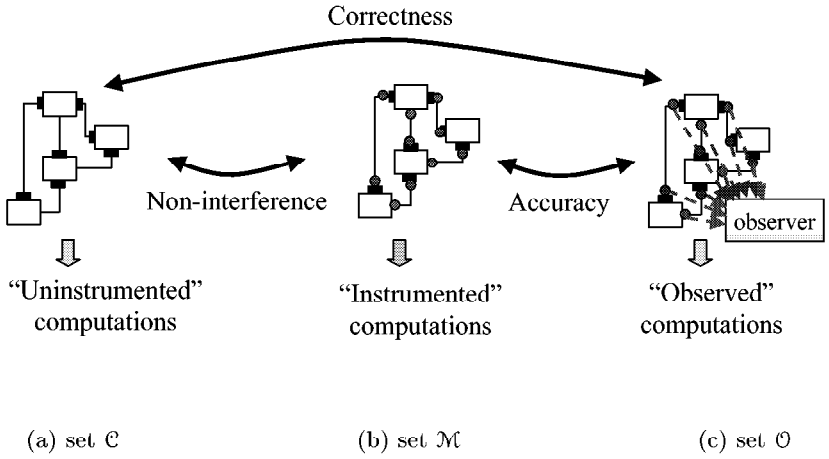


Figure 4.10: Intrusivness properties

Three properties allow us to qualify the differences of the three kind of computations: *non-interference, accuracy* and *correctness*, as depicted in Figure 4.10. Different degrees for these properties are defined: *total, strong* and *minimal.* In all degrees, we require that the entire set of events is conserved: $V_C = V_{M,C} = O_M$. The difference between "totally" and "strongly" depends only on the introduction of new ordering in the set of events monitored or observed. The degree "minimal" indicates that only the set of events is preserved, but that no assumption can be made on the order of the events.

- The *non-interfering* property qualifies the similarity between the computations of the monitored application and, of the unmonitored (uninstrumented) application. The *non-interfering* property is defined as follows:

$$\text{non-interfering} \begin{cases} \textit{total} & C = (V_{M,C}, R_{M,C}) \\ \textit{strong} & V_C = V_{M,C} \wedge R_C \subset R_{M,C} \\ \textit{minimal} & V_C = V_{M,C} \end{cases}$$

- The *accuracy* property qualifies the similarity between the computations of the monitored (instrumented) application and the observations of the application. The *accuracy* property is defined as follows:

$$\text{accuracy} \begin{cases} total & O_M - (V_{M,C}, R_{M,C}) \\ strong & V_{O_M} = V_{M,C} \wedge R_{M,C} \subset R_{O_M} \\ minimal & V_{O_M} = V_{M,C} \end{cases}$$

- The *correctness* property qualifies the similarity between the observations of the application and its unmonitored (uninstrumented) computations. The *correctness* property is defined as follows:

$$\text{correctness} \begin{cases} total & O_M = (V_C, R_C) \\ strong & V_{O_M} = V_C \wedge R_C \subset R_{O_M} \\ minimal & V_{O_M} = V_C \end{cases}$$

According to these properties, let us now evaluate our approach. Considering the event types that are identified, and the filtering mechanism generating the traces, our monitoring mechanism guarantees that all events are detected and reported to the Observer (and that no other events are added or removed). Our monitoring mechanism is at least *minimally non-interfering, minimally accurate* and *minimally correct.*

Once the code has been instrumented, its execution will involve a slightly greater amount of computations (e.g., sending of notification, handling of timestamps). These computations introduce some processing delays with respect to the un-monitored (un-instrumented) computation. These delays can influence the behavior of an application; some new orderings of events, which are not present in an un-monitored execution, can be enforced (this is analyzed in the next section). This implies that $R_C \not\subset R_{M,C}$ and $R_C \not\subset R_{O_M}$ and thus, our monitoring mechanism is *minimally non-interfering* and *minimally correct.*

Nevertheless, our monitoring system is *totally accurate.* This is proved in Lemma 1.

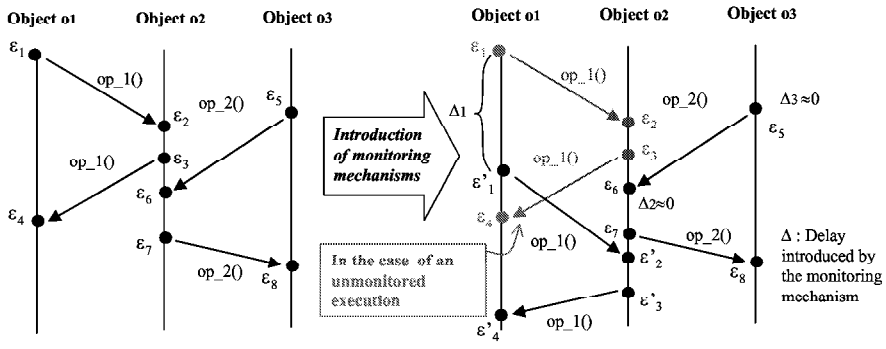
Lemma 1 Our monitoring system is totally accurate.

Proof By sending a notification for each event, we guarantee that all events are observed (we have defined the set of events and know how to observe them). Therefore, $V_{O_M} = V_{M,C}$. By using the timestamp mechanism, we guarantee that the partial order of observed events reported to the observer is the same as the one occurring in the monitored system. This is guaranteed by the correctness of the RST algorithm [RST91]. ■

4.9.2 Impact of the monitoring mechanism on the ordering of events

In Figure 4.11 an example where the monitoring mechanism changes the execution (order of events) is given. In this example, the delay introduced affects the order of

occurrence of events: op_1 is invoked before op_2 in an unmonitored execution. In a monitored execution, because of the processing delay Δ_1 introduced in object o_1, op_2 is invoked before op_1. In other words, the introduction of the monitoring mechanism has enforced an order that is not present in an unmonitored execution. The change of order of events in the system might however not be so dramatic. In most of the cases, the processing added to each object will have the same impact on each object. Therefore, the monitoring computation will only execute more slowly, without any other damages.



(a) Uninstrumented system (b) Monitored System

Figure 4.11: Example of the influence of the monitoring mechanism

If we want to be more precise in the quantification of our monitoring approach the happened-before relation (Section 4.8) has to be decomposed in order to analyze which "part" of the happened-before relation is not ensured. Let us first decompose the happened-before relation. $\varepsilon \rightarrow \varepsilon'$ can be decomposed into:

1. if events ε and ε' happen in the same object (we note: $\varepsilon <^i \varepsilon'$). This relation can be divided into two different parts:
   - if events ε and ε' happen in the same object but are not relative to the same operation (we note: $\varepsilon <^i_{\not op} \varepsilon'$) (e.g, in Figure 4.11(a), $\varepsilon_3 <^i_{\not op} \varepsilon_6$).
   - if events ε and ε' happen in the same object and are relative to the same operation occurrence. If they are happening in the same object during the same operation (we note: $\varepsilon <^i_{op} \varepsilon'$) (e.g, in Figure 4.11(a), $\varepsilon_2 <^i_{op} \varepsilon_3$).
2. if events ε and ε' happen in two different objects, and ε is the sending of message and ε' is its reception (we note $\varepsilon \vec{<} \varepsilon'$) (e.g, in Figure 4.11(a), $\varepsilon_3 \vec{<} \varepsilon_4$).

Thus, the happened-before relation $\rightarrow$ is equivalent to the union of the two relations $<^i$ and $\vec{<}$.

By using the filtering mechanism, we are sure that all events are monitored (we have defined the events and the way they can be detected). Therefore, $V_{M,C} = V_C$. As our monitoring mechanism is totally accurate, $V_{O_M} - V_{M,C} - V_C$ and $R_{M,C} - R_{O_M}$. The only missing part is the relation between $R_{M,C}$ and R_C.

The example of Figure 4.11 has shown that $R_C^{<^i_{op}} \not\subset R_{M,C}$. But, our monitoring mechanism does not change (or does not allow the change of) the sequence of events of an operation invocation. Indeed, by adding the filters, a new processing is added at each event. Nevertheless, this extra processing has to be finished in order to allow the system to proceed to a new action generating a new event and this extra processing does not allow he occurrence of an event that should not occur if the system would be unmonitored.

In other words, the processing of the Filters has to be finished in order to allow the system to send a new request (i.e, $R_C^{\vec{<}} \subset R_{M,C}$) ; the processing of the Filters has to be finished to allow the system to execute the next internal action of the same operation (i.e, $R_C^{<^i_{op}} \subset R_{M,C}$) .
Therefore, $R_C^{\vec{<}\cup<^i_{op}} \subset R_{M,C}$ and, thus $R_C^{\vec{<}\cup<^i_{op}} \subset R_{O_M}$.

4.9.3 Discussion

Our system of monitoring observe all the occurring events and without adding new events; however, it can change the order of the events, occurring in the same process, which are not part of the same operation invocation.

We can notice that we have put a very strong requirement in each of the intrusivness properties: $V_C = V_{M,C}$ or $V_{M,C} = V_{M_O}$. Such a requirement is generally not fulfilled by the usual monitoring methods as they do not offer the garantee that all events are observed. But, our system does not suffer from this problem. If we consider the usual way to monitor systems (the manual instrumentation of the code to collect traces (i.e., the use of 'printf()' statements)), this monitoring method is not minimally non-interfering, nor minimally safe, nor minimally accurate, nor minimally correct.

In summary, the analysis presented in this section reveals that our Observer reports exactly the computation of the instrumented application. However, due to the extra computations induced by the instrumentation, the computation of an uninstrumented and instrumented application can differ in the ordering of events occurring. The order of the events that are in te same operation invocation path are never changed by the monitoring mechanism. The order of the other events can be changed.

As the extra computations added are similar for all objects of the applications, the impact consists mainly in slowing down the application execution. Nevertheless, it might be possible that the delays introduced can provoke deadlock or mis-behaviors. In such cases, our mechanism will have, at least, demonstrated the sensibility of the application to the execution time, and thus to the processing power and/or network bandwidth.

Solving the problem would mean adding special hardware and network resources for monitoring purposes, which is very complicated and may not be feasible in many industrial situations, or change the middleware platform implementation (the protocol used, the scheduling mechanism, etc.) [WGS98].

4.10 Related Work

Since the emergence of distributed and parallel systems a great amount of work has been published in the domain of monitoring. Marzullo et al. propose the *Meta* [MCWB91] system tool for the monitoring of distributed applications built on top of the ISIS [BJ87] distributed environment. They use *sensors* to get monitored information and *actuators* to perform actions on the system. In our approach we do not, as yet, take into account the possibility of controlling the system. In [HKM+94], Hofmann and al. present a hybrid monitoring system. The approach can be classified as low level because they take into account software, as well as hardware monitoring. They use some PLL in order to maintain a global time of the system. This kind of synchronization is not feasible in an upper abstraction level, as defined by CORBA. For example, no assumptions can be made on the delay of a request delivered by the ORB nor the burstiness of the notifications. Therefore, any synchronization attempt is doomed to fail. Both monitoring approaches mentioned here suffer from the manual instrumentation of the system. In [BKS+96], Brutch et al. present a monitoring tool SMT (System Monitoring Tool) for OSF/DCE (Open Software Foundation's/Distributed Computing Environment). This tool is based on the interception by RMON (Remote Network Monitoring) Agents of Remote Procedure Call (RPC) PDUs. One RMONs is required on each of the network segments which are to be monitored. The SMT software runs on a host workstation, controls the RMONs via Simple Network Management Protocol (SNMP), and polls RMONs Agent periodically to get the information logged. As in our approach, the SMT tool does not need to change the code of the distributed application to get monitored information. However, it necessitates knowing how the network is configured in order to introduce the RMONs agents. Furthermore, this approach necessitates the mapping of the RPC PDUs into the different RPC transactions in order to get information about application level relevant informations. This is not the case in our approach; we get relevant information directly because we do not go as deep into the protocol used to communicate between the different parts of the application. Our approach is simpler.

More recent work has been proposed, which offers more complex and adequate monitoring. In [QM97] Queiroz and Madeira present a monitoring system for distributed applications built on top of a CORBA DPE. They use the monitored information for management purposes. Despite the originality of the approach, they do not take into account reordering of monitored informations, nor do they give a formalism of their concept or the possibility to dynamically change the configuration of the monitoring mechanism. Al-Shaer et al. [AAM97] propose a monitoring approach for large scale distributed applications. This proposal is scalable and allows the dynamic configuration of the monitoring mechanism in order to configure on-line the monitored information to gather. Al-Shaer et al. also define a language to specify the event to monitor. The order of the monitored information is maintained by a synchronization mechanism and the notification of the event to the monitored agent is performed by UNIX sockets. In this proposal, the user has to define the events to observe, in our approach the user will only have to select predefined useful information proposed according to the IDL specification of the application.

In [Per94], Graeme Perrow makes a survey of three different monitoring techniques for distributed systems management. The first one is due to Snodgrass and is based on a historical relational database that records all historical events. For this purpose he has defined a sophisticated language. Because of this language, the proposal is very complex and necessitates a language compiler. One of the disadvantages of the previous proposal is that all of the queries must be specified before any of the data is collected. Snodgrass later addresses this problem and gave a general outline of how to write a dynamic, distributed monitor. He assumes that processor clocks of the different nodes of the distributed system are always synchronized to within ten of microseconds. This may not always be the case.

In [MSS93], Mansouri-Samani and Sloman present a functional model of monitoring in terms of the generation, processing, dissemination, and presentation of information can help determine the facilities needed to design and construct distributed systems. In [Bat95], Bates present the *Event-Based Behavioral Abstraction (EBBA)* allowing for the definition of arbitrary events using an event description language. With this proposal, the user has to specify the events using the language proposed, in our approach the user only needs to select the events to monitor.

The problem of ordering of observed information of distributed computation has first been studied by Lamport [Lam78]. He has defined the notion of *causal precedence*. His work is always referred in the domain. Later, Fidge [Fid96], Mattern [Mat89] have developed a timestamp mechanism used to determine the causal precedence of distributed computations. Schiper et al. [SES89] and Raynal et al. [RST91] have propose an algorithm and a timestamp data structure that preserve the causal ordering of distributed computations. We used and adapted that algorithm to take into account the dynamic creation and deletion of entities in the distributed system. The work of Alagar and Venkatesan [AV97] which is devoted to the mobile computing domain takes into account the creation and deletion of nodes. This work is specific to the domain of mobile computing and the possible connection and disconnection of mobile hosts.

4.11 Conclusions

In this chapter we presented the method we developed for the monitoring of distributed applications, more precisely communication services. We explained how it is possible to detect the events in the running software. We defined a method to automatically add the instrumentation that is used to report the events to the Observer. For this, we stated a format for the notifications. To guarantee the correct ordering of the notification we proposed the use of an ordering algorithm. Finally we analyzed the theoretical intrusivness of the monitoring.

The principal achievements and originality of the method we designed, are analysed hereafter:

1. *A generic and implementation language independent monitoring method.* By

the use of the set of events that was defined in close collaboration with industrial partners, our monitoring method is not restricted to work only with specific middlewares. With our method, we monitor the events occurring at the implementation execution but without being dependent on the implementation language used. Furthermore, the Observer receiving all the notifications from the events occurring can be implemented in a different language than that of the application under scrutiny.

2. *A method that fits well in the development process.* Our method does not necessitate extra steps, extra specifications or extra implementation code to be written in order to be applied. The application under scrutiny is automatically instrumented by a specific tool in order to be monitored. This avoids any manual instrumentation steps and does not require in-depth knowledge of the application. The instrumentation code is automatically added by a code generator that is the only part that is implementation language dependent.
3. *A monitoring method that is easy to use.* The monitoring tool that is the result of the method explained in this chapter encompasses a number of features that makes it user-friendly. The user can chose dynamically, at run-time the event he wants to monitor, he can get a behavioral view of the application at run-time with a guarantee of correctness.

The above points were the determining factors when Alcatel Research, France decided to introduce our method in the PERCO platform. PERCO is designed as a CORBA distributed platform and several features (such as fault Tolerance, load sharing, remote object creation) are sold to customers to develop complex distributed applications.

Based on the information that can be gathered at run-time, in the following Chapter we present the design of a run-time testing mechanism for communication services.

Chapter 5

Testing

Selected portions of this Chapter were published in [DLKH98a] and [LDH98]

5.1 Chapter Overview

In this chapter we present the architecture and methods we developed in order to test communication services. The language that is used to specify the properties that are checked at runtime is explained. We present the algorithm to translate properties into Automata. The Automata representation and simplification are explained. Finally we explore how the performances of the mechanism could be increased.

5.2 Introduction

Testing is defined as [IEE97]:

> *The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspects of the system or component.'*

Testing and debugging are often lumped under the same headings and often used together as "testing and debugging". Nevertheless, the purpose of testing is to show that a program is free of faults (commonly called bugs). The purpose of debugging is find error or misconception that led to the program's failure and to design and implement the program changes that correct the error. Monitoring consists of collecting, interpreting and displaying information concerning a distributed system [J+87]. Clearly, monitoring and testing are two complementary things.

The practice of testing software is as old as that of developing software itself. But it has generally been seen as a secondary activity, something done afterwards to confirm that the correct software has been built. Executing the software after its initial production, during a test or integration phase, may take as much or even more

effort than the development of the software up to that point.

Even after decades of exploration, testing is still a human-intensive activity, thus error-prone. This is to a certain extent due to a lack of formality, which would otherwise allow testers to do their job efficiently and to take advantage of powerful tools. To make the use of formal approaches more appealing to software designers and testers, it is beneficial to encapsulate formal methods concepts and/or algorithms: users do not need to know how it works, or even that it is there.

The landmark characteristics of our approach are the expression of properties in an independent manner of the implementation language and the testing of those properties at system run-time without requiring any help from the programmer/tester to map the properties to the implementation level. Only the property specification must be derived manually. Some generic observation and testing code is generated automatically. At service run-time, properties can be entered dynamically; the corresponding service behavior is observed constantly and checked against the formally specified properties. The construction of test oracles and the generation, collection and analysis of the test traces can be hidden from the service tester.

5.3 The property language

In order to capture the requirements of the application we want to test, we are using *Linear Time Temporal logic* (LTL). LTL is a special type of modal logic used for describing and reasoning about how the truth values of assertions change over time. *Modal logic* was originally conceived as the logic of necessity and possibility, and developed by philosophers to study different "modes" of truth. *Temporal logic* is a special type of modal logic used for describing and reasoning about how the truth values of assertions change over time.

With respect to temporal logic, time can be regarded in two different ways: One is that the course of time is linear (at each moment there is only one possible future moment). The other is that time has a branching (tree-like) structure: at each moment, time may split into alternate courses representing different possible futures. Accordingly, the temporal logics are called either *linear-time temporal logic* or *branching-time temporal logic.*

Linear-time temporal logic provides a number of operators that refer to the future or to the past. These operators are explained in Section 5.3.1.

In order to express properties from the requirements, we use LTL over the set of events that we have defined in Chapter 3. Normally, LTL formulas are interpreted over a sequence of states. Therefore, we explain in Section 5.3.2 what is a state and what is an event and how one can express properties with events.

5.3.1 Basic introduction to Linear Temporal logic

In this section, the basic operators of LTL [MP91a] are defined and listed in Table 5.1[1].

Future Temporal Operators	
$\Box p$	Henceforth p
$\Diamond p$	Eventually p
$p\ \mathcal{U}\ q$	p Until q
$p\ \mathcal{W}\ q$	p Waiting-for (Unless) q
$\bigcirc p$	Next p

Table 5.1: The temporal operators

Given a model $\sigma = \langle s_0, s_1, \ldots, s_n, \ldots \rangle$ and a formula p, then

$$(\sigma, j) \models p$$

denotes that formula p holds at a position $j, j \geq 0$ in the state sequence σ.

The *Henceforth* operator $\Box$: If p is a temporal formula, then so is $\Box p$, read henceforth p or always p. Its semantics is defined by

$$(\sigma, j) \models \Box p \iff \textit{for all } k \geq j, (\sigma, k) \models p$$

Thus, $\Box p$ holds at position j iff p holds at position j and all following positions.

The *Eventually* operator $\Diamond$: If p is a temporal formula, then so is $\Diamond p$, read as *eventually* p. Its semantics is defined by

$$(\sigma, j) \models \Diamond p \iff \textit{for some } k \geq j, (\sigma, k) \models p$$

Thus, $\Diamond p$ holds at position j iff p holds at some position $k \geq j$.

The *Until* operator $\mathcal{U}$: The *Until* formula $p\ \mathcal{U}\ q$ (read p until q) combines the two previously defined constructs by predicting the eventual occurrence of q and stating that p holds continuously, at least until the first occurrence of q. If p and q are temporal formulas, then so is $p\ \mathcal{U}\ q$. Its semantics is defined by

$$(\sigma, j) \models p\ \mathcal{U}\ q \iff \begin{array}{l} \textit{for some } k \geq j, (\sigma, k) \models q \\ \textit{and for every } i \textit{ such that } j \leq i < k, (\sigma, i) \models p \end{array}$$

Thus, $p\ \mathcal{U}\ q$ holds at position j iff q holds at some position $k \geq j$ and p holds continually from j through $k - 1$.

[1] Only the future operators are presented since we do not use the past operators in this work (see 5.3.3)

The *Unless* (*Waiting-for*) operator $\mathcal{W}$: If p and q are temporal formulas, then so is $p \mathcal{W} q$, read as p *unless* q. Its formal semantics is defined by

$$(\sigma, j) \models p \mathcal{W} q \Longleftrightarrow (\sigma, k) \models p \mathcal{U} q \text{ or } (\sigma, j) \models \Box p$$

Thus, $p \mathcal{W} q$ holds at position j iff p holds continuously either until the next occurrence of q or throughout the sequence.

The *Next* operator If p is a temporal formula, then so is $\bigcirc p$, read as *next* p. Its semantics is defined by

$$(\sigma, j) \models \bigcirc p \Longleftrightarrow (\sigma, j+1) \models p$$

Thus, $\bigcirc p$ holds at position j iff p holds at the next position $j+1$.

Example Following are a few temporal logic formulas with their meaning:

- $p \rightarrow \Diamond q$ states if initially p then eventually q.
- $\Box(p \rightarrow \Diamond q)$ states that every p-position coincides with or is followed by a q-position.
- $\Box\Diamond p$ states that the sequence σ contains infinitely many p-positions.
- $\Diamond\Box p$ states that there exists a position such that p holds at all later positions.

5.3.2 States & Events

In [DLKH98b] and in Chapter 3 a set of events to model object-oriented distributed application was defined. An object-oriented distributed system is represented as a set of *state variables*. The value of each of these state variables indicates the *state* of the system. An *event* occurrence changes the value of at least one state variable thus leading to a new state of the system.

LTL is built to express temporal properties which are interpreted over a state sequence. In our monitoring and testing mechanism the information we gathered to test our system are events and not states. Therefore, in [DLKH98b] a new operator "$\odot$" which is called "state" has been defined. This operator allows for the properties to be specified by using events. Let e be an event, $\odot e$ is equivalent to the state that follows the occurrence of event e.

The interested reader can find more information on the notion of event and state in [Die00].

5.3.3 Interesting Properties

One can not assume an efficient, scalable, on-line, testing mechanism if all past events have to be recorded or stored. One of the key issues for on-line testing is the absence

of past operators in the properties. The properties have to be expressed using only future operators ($\Box, \Diamond, \mathcal{U}, \mathcal{W}, \bigcirc$) presented in Table 5.2. The operator $\bigcirc$ is not used in this work for the expression of properties, since it is very difficult to predict the next event in a completely concurrent program as are distributed applications we consider.

Henceforth p	$\Box p$
Eventually p	$\Diamond p$
p Until q	$p \mathcal{U} q$
p Unless q	$p \mathcal{W} q$
Next p	$\bigcirc p$

Table 5.2: Future Temporal Operators

Manna and Pnueli classified the different kinds of linear temporal logic properties that are useful for validating concurrent programs [MP91b]:

- **Invariants**: $\Box p$
  "An invariance property refers to an assertion p, and requires that p is an invariant over all the computations of a program P, i.e., all the states arising in a computation of P satisfy p."
- **Response**: $\Box(p \rightarrow \Diamond q)$
  "A response property refers to two assertions p and q, and requires that every p-state (a state satisfying p) arising in a computation is eventually followed by a q-state."
- **Precedence**: $\Box(p \rightarrow q \,\mathcal{U}\, r)$ where '$\mathcal{U}$' is the *unless* operator (weak *until*)
  "A simple precedence property refers to three assertions p, q, and r. It requires that any p-state initiates a q-interval (i.e., an interval all of whose states satisfy q) which, either runs to the end of the computation, or is terminated by an r-state. Such a property is useful in order to express the restriction that, following a certain condition, one future event will always be preceded by another future event. For example, it may express the property that, from the time a certain input has arrived, there will be an output before the next input. Note that this does not guarantee that output will actually be produced. It only guarantees that the next input (if any) will be preceded by an output."

In this thesis we focus on these three kind of properties. Nevertheless the algorithm for the building of automata that we describe in Section 5.7 is applicable to any LTL formula.

5.4 Expression of Properties

Manipulating LTL formula requires some experience and it is sometimes difficult to express what is evident in a natural language under the form of a formal specification. In Figure 5.1, we explain how our approach helps the specification of properties.

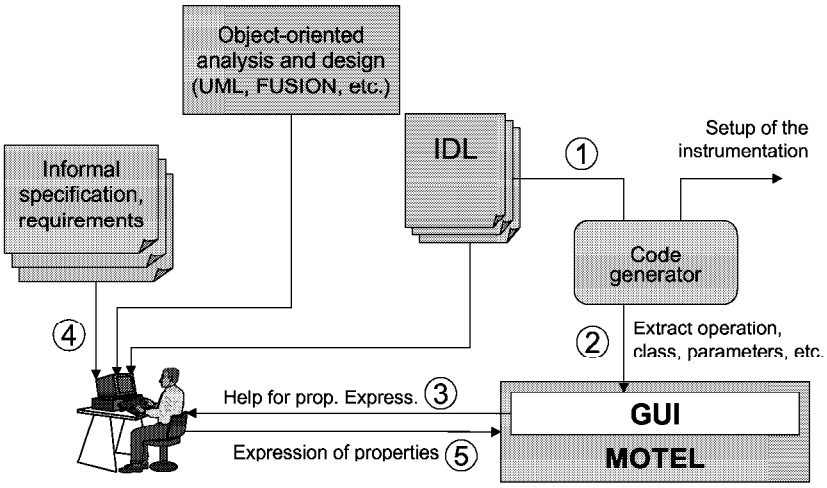


Figure 5.1: How ro express properties

Usually, the developer has different documents to describe the application he has to develop/test. The first is the informal requirements that gives the instructions in a natural language. The second are the documents issued from the object-oriented development method used: UML, FUSION, etc. And finally, since we deal with middleware based applications, the IDL definitions of the different interfaces of the system.

In our approach, (1) the IDL specification is given to a code generator that extracts the information (name, type, et.) of the operations, classes, parameters that are in the specification and gives that (2) to the tool that integrated the information in the GUI in order to offer (3) menus to inform the user of the different elements of the system. Based on this information, the informal requirements (4) and the documents issued from the object-oriented development phase (5) the developer can express properties. In addition to menus, the GUI can also offer a library of typical usefull properties (e.g., properties that were used in other applications and the three propertiy classes that we defined in the previous section).

5.5 Automata Theory

The following definitions are reminders of the basic automata theory [Per90]. A *word* is a finite sequence of symbols taken out of a given set called *alphabet*. Let A be an alphabet. We denote a word with a mere juxtaposition of its letter as $x = a_1 a_2 \ldots a_n$ when a_i are letters from A for $1 \leq i \leq n$. We denote by A^* the set of all words on the alphabet A.

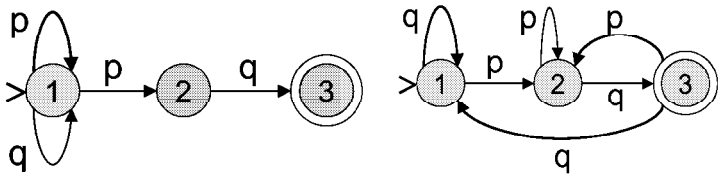
A *finite automaton* on the alphabet A is given by a finite set Q of *states*, two subsets I, T, of Q called the sets of *initial* and *terminal* states and a set $E \subset Q \times A \times Q$ of *edges*. We denote the automaton as a quadruple (Q, I, T, E).

A *path* in the automaton $\mathcal{A} = (Q, I, T, E)$ is a sequence $c = (e_i)_{1 \leq i \leq n}$ of consecu-

tive edges: $e_i = (p_i, a_i, p_{i+1})$. The word $w = a_1 a_2 \ldots a_n$ is the *label* of the path, state p_i is its *origin* and state p_{i+1} its *end*. The number n is its *length*. A word w on the alphabet A is *recognized* by the automaton $\mathcal{A}$ if it is the label of some successful path. The *set recognized* by the automaton, denoted $L(\mathcal{A})$ is the set of all words recognized by $\mathcal{A}$. A set $X \subset A^*$ is said to be *recognizable* if there exists a finite automaton $\mathcal{A}$ such that $X = L(A)$.

An automaton $\mathcal{A} = (Q, I, T, E)$ is said to be *deterministic* if $card(I) = 1$ and if for each pair $(p, a) \in Q \times A$ there is at most one state $q \in Q$ such that $(p, a, q) \in E$. It is said to be *complete* if for each $(p, a) \in Q \times A$ there is at least one $q \in Q$ such that $(p, a, q) \in E$.

The automata represented in Figure 5.2 corresponds to $A = \{p, q\}$, $I = \{1\}$ and $T = \{3\}$ and they recognize the set of words ending with pq. The automaton of Figure 5.2 (b) is deterministic and complete; the automaton of Figure 5.2 (a) is neither deterministic nor complete.



(a) Non-deterministic (b) Deterministic

Figure 5.2: Example of automata

Theorem 1 *For each finite automaton, there exists a deterministic and complete one recognizing the same set [Per90].*

Proof Starting from $\mathcal{A} = (Q, I, T, E)$, one builds the automaton $\mathcal{B}$ whose states are the subsets of Q, having the set I as initial state, the set $U \subset Q$ that meet T as terminal states and whose edges are the triples (U, a, V) such that V is the set of all states $q \in Q$ such that there exists an edge $(p, a, q) \in E$ with $p \in U$. ■

The construction mentioned in the proof of Theorem 1 is presented in Figure 5.3. The construction is based on Figure 5.2 (a) and if one removes all the states that cannot be reached from the initial state, Figure 5.2 (b) is obtained.

In the worst case, the size of a DFA (Deterministic Finite State Automaton) can be exponentially larger than that of the corresponding Nondeterministic Finite Automaton (NFA). However, as stated by Dillon and Yu [DY94], preliminary investigation suggests that DFAs for formulas expressing standard safety and liveness requirements are typically much smaller than worst case DFA.

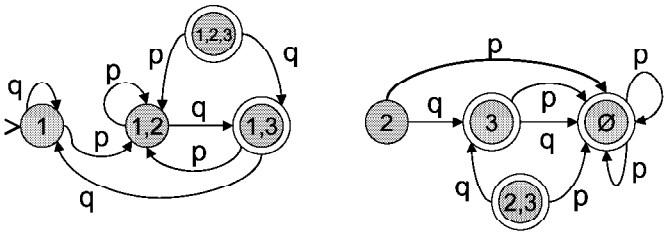


Figure 5.3: Illustration of Theorem 1

5.6 Büchi Automata

A^ω denotes the set of ω-word over A. An ω-word over A is written in the form $\alpha = \alpha(0)\alpha(1)\alpha(2)\ldots$ with $\alpha(i) \in A$.

Büchi automata [Büc60] are non-deterministic finite automata equipped with an acceptance condition that is appropriate for ω-words: an ω-word is accepted if the automaton can read it from left to right while assuming a sequence of state in which some final state occurs infinitely often (*Büchi acceptance*).

A *Büchi automaton* over the alphabet A is of the form $\mathcal{A} = (Q, q_0, \Delta, F)$ with finite state set Q, initial state $q_0 \in Q$, transition relation $\Delta \subseteq Q \times A \times Q$, and a set $F \subseteq Q$ of final states. A *run* of $\mathcal{A}$ on an ω-word $\alpha = \alpha(0)\alpha(1)\alpha(2)\ldots$ from A^ω is a sequence $\sigma = \sigma(0)\sigma(1)\ldots$ such that $\sigma(0) = q_0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \Delta$ for $i \geq 0$. The run is called *successful* if some state of F occurs infinitely often in it. $\mathcal{A}$ *accepts* α if there is a successful run of $\mathcal{A}$ on α. Let $L(\mathcal{A}) = \{\alpha \in A^\omega \mid \mathcal{A}\ accepts\ \alpha\}$ be the ω-language *recognized* by $\mathcal{A}$. If $L = L(\mathcal{A})$ for a Büchi automaton $\mathcal{A}$, L is said to be *Büchi recognizable.*

5.7 Translation of the properties

5.7.1 Introduction

The theory of translation LTL formulas into automata [DR96, GW89, Hol91] determines that a LTL formula is equivalent to a finite state automaton, which accepts precisely those state sequences that satisfy the formula. Thus, a state sequence s violates a LTL formula f if the automaton associated with $\neg f$ accepts s. By using the automaton for $\neg f$ rather than f, information in the accepting run can be used, in case s violates f, to give detailed info on the sequences of states that produced the error as well as used to generate input for a next test sequence [JPP+97]. When checking for the violation of property f, it is easier to check for the occurrence of $\neg f$.

Formally a Büchi automaton accepts only a sequences of infinite words. Instead of terminating, a Büchi automaton goes to an accepting state where it can remain infinitely, independent of any condition. This is depicted in Figure 5.4. In order to be

applicable in practice, the capability to capture an infinite word of Büchi automata has to be interpreted differently. The event traces generated during a testing scenario are finite. Thus, our automata do not require the power -or complexity- of Büchi automata.

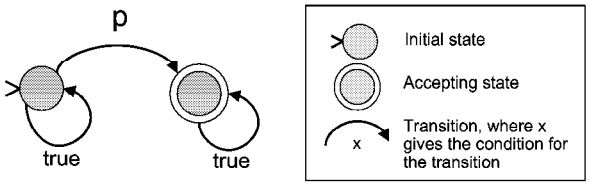


Figure 5.4: Büchi automata for the property $\Diamond p$

In [GW89], Grimont and Wolper claim that "For an automaton on infinite words, one says that an execution is accepting if it ends in an accepting state" . When dealing with infinite words, this notion can no longer be used since the execution is infinite and hence does not have a last state. We will say that an execution of a Büchi automaton is accepting if it contains a least one accepting state an infinite number of times. More formally, an execution $\sigma = s_0 s_1 \ldots$ is accepting if there is some state $s \in \mathcal{A}$ ($\mathcal{A}$ is the set of accepting states of the Büchi automaton) and an infinite number of integers $i \in \mathbb{N}$ such that $s_i = s$". In other words, the last state can be considered as repeating infinitely.

In [Eti95], Etique pointed out that in practice it is not very convenient to wait for the end of the test sequence in order for the errors to be detected. It would be interesting to be able to detect error before the end of the test scenario. Thus Etique proposes another solution to handle infinite sequences: "If a Büchi automaton remains in an accepting loop during a time τ, it is said to be accepting". Note that making τ greater than the duration of the test sequence makes Etique's solution equivalent to Grimont and Wolper's one.

We adopt the solution of Etique and we extend it by adding a new state type, that we call *error-state*, as a special case of an accepting case.

> *An accepting state of a Büchi automaton, is defined as an* **error-state** *iff there is only a transition with the condition "true" that loops over that state.*

With this definition, the Büchi automata representing the formula $\Diamond p$ of Figure 5.4 is transformed into a new automata that is presented in Figure 5.5. The introduction of the error-state, allows to detect the error as soon as it is encountered. With the other two solutions, the error would have been detected at the end of the test sequence, or after a time τ.

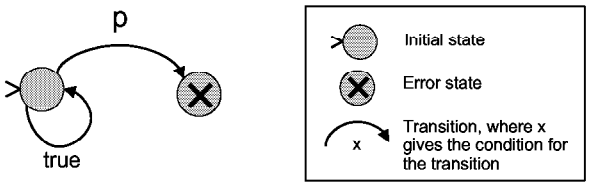


Figure 5.5: Automata for the property $\Diamond p$ with an error-state

5.7.2 Basic Translation method

Most of the algorithms used to translate LTL properties into automata produce Non-Deterministic Automata (NDA). NDAs contain nondeterministic choices; consequently, the management of the automaton can become unwieldy due to its size: at each nondeterministic transition a new automaton has to be created in order to explore the other choices. For example, SPIN transforms the properties into *never claims* [Hol91], which are NDAs. In [GW89] the interested reader can find a method to translate LTL formulae onto (non-deterministic) Büchi automata. In [DR96], a method to translate any temporal property into NDA is presented. We based our work on that algorithm, since it is generic for all the temporal logic formal methods.

In [DR96], Dillon developed a method to translate temporal logic formulae into FSMs. The method is new in the sense that it can be applied to LTL, as well as other temporal logics. This method could easily be applied in our case.

These rules for building an automaton from a LTL formula are based on the following equivalences:

$$
\begin{aligned}
p \rightarrow q &\Leftrightarrow \neg p \vee q \\
\Box p &\Leftrightarrow p \wedge \bigcirc \Box p \\
\Diamond p &\Leftrightarrow p \vee \bigcirc \Diamond p \\
p \mathcal{U} q &\Leftrightarrow q \vee (p \wedge \bigcirc (p \mathcal{U} q) \\
\neg (p \mathcal{U} q) &\Leftrightarrow (\neg p \wedge \neg q) \vee (\neg q \wedge \bigcirc \neg (p \mathcal{U} q))
\end{aligned}
$$

According to theses equivalences, one can notice that a formula f can be split into two formula f_{now} and f_{next}. f_{now} corresponds to the formula that must to be true for the current state, and f_{next} to the formula that will be checked in the next state. The rules for the translation of properties into automata are summarized in the Table 5.3. The procedure consists in transforming the properties in order to get only conjunction, disjunctions, negations and the "$\bigcirc$" operator.

The corresponding automata of the three properties are presented in the next three sections.

Table 5.3: Reduction rules for common LTL operators

Label	Formula	Reduction options
$[r\wedge]$	$f_1 \wedge f_2$	$\{f_1, f_2\}$
$[r\vee]$	$f_1 \vee f_2$	$\{f_1\}, \{f_2\}$
$[r\rightarrow]$	$f_1 \rightarrow f_2$	$\{\neg f_1\}, \{f_2\}$
$[r\Box]$	$\Box f$	$\{f, \bigcirc \Box f\}$
$[r\Diamond]$	$\Diamond f$	$\{f\}, \{\bigcirc \Diamond f\}$
$[r\mathcal{U}]$	$f_1 \mathcal{U} f_2$	$\{f_2\}\{f_1, \bigcirc(f_1 \mathcal{U} f_2)\}$
$[r\neg\mathcal{U}]$	$\neg(f_1 \mathcal{U} f_2)$	$\{\neg f_1, \neg f_2\}, \{\neg f_2, \bigcirc \neg(f_1 \mathcal{U} f_2)\}$

5.7.3 Invariant

For the invariant, as explained in Section 5.7.1, $\Box p$ we want to check if there is an occurrence of $\neg\Box p$. As $\neg\Box p \Leftrightarrow \Diamond\neg p$ the corresponding automaton is presented in Figure 5.6. This automaton is exactly accepting the sequence of states that violate

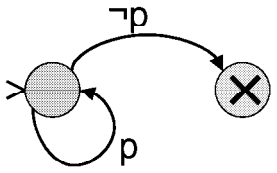


Figure 5.6: Automaton for the property $\neg\Box p$

the property.

5.7.4 Response

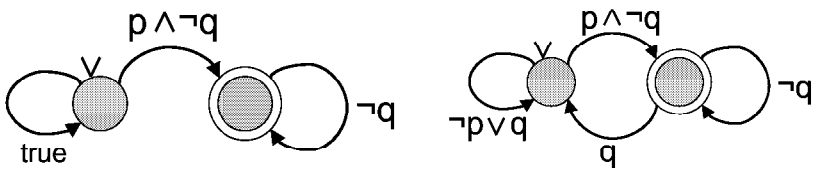
The automaton corresponding to the Response property $\Box(p \rightarrow \Diamond q)$, accepts the state sequence satisfying $\neg\Box(p \rightarrow \Diamond q)$. The property $\neg\Box(p \rightarrow \Diamond q)$ can be transformed:

$$\neg\Box(p \rightarrow \Diamond q) \Leftrightarrow \Diamond(p \wedge \neg\Diamond q) \Leftrightarrow \Diamond(p \wedge \Box\neg q)$$

The automata associated with $\Diamond(p \wedge \Box\neg q)$ is given in Figure 5.7:

if $p \wedge \neg q$ is true then $\neg q$ has to be infinitely true, else $p \wedge \neg q$ is waited for (the transition true in the automata[2]). In Figure 5.7.b the DFA of the NFA of Figure 5.7.a is presented. This DFA is intuitively easy to understand: if $p \wedge \neg q$ has been first true, then $\Box\neg q$ has to be true. Then, if once q is true then the property cannot be violated anymore until the next occurrence of $p \wedge \neg q$ (transition from the accepting state to the initial state with the label q). Note that this DFA can also be built by applying the constructs of Theorem 1.

[2]The transition "true" has to be used in order to catch all the occurrences of $p \wedge \neg q$. The property states: $\Box(p \rightarrow \Diamond q)$, and not $p \rightarrow \Diamond q$.



(a) Non-deterministic (b) Deterministic

Figure 5.7: Automata for $\neg\Box(p \rightarrow \Diamond q)$

5.7.5 Precedence

The automaton corresponding to the Precedence property has to accept the sequence of states satisfying $\neg\Box(p \rightarrow q\mathcal{U}r)$. The property $\neg\Box(p \rightarrow q\mathcal{U}r)$ can be transformed:

$$\neg\Box(p \rightarrow q\mathcal{U}r) \Leftrightarrow \Diamond(p \wedge \neg(q\mathcal{U}r))$$

In Table 5.4, the application of the reduction rules (Table 5.3) is presented.

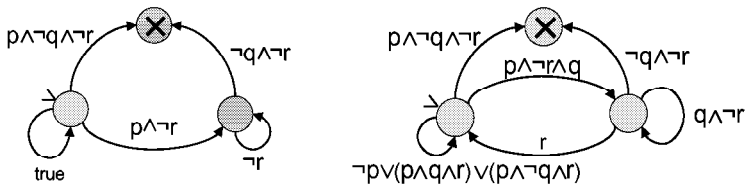
Table 5.4: Translation of $\neg\Box(p \rightarrow q\mathcal{U}r)$ into an automaton

Formula	Comments
$\Diamond(p \wedge \neg(q\mathcal{U}r))$	apply $[r\Diamond]$
$\{p, \neg(q\mathcal{U}r)\}\{\bigcirc\Diamond(p \wedge \neg(q\mathcal{U}r))\}$	apply $[r\neg\mathcal{U}]$
$\{p, \neg q, \neg r, \bigcirc true\}\{\neg r, p, \bigcirc\neg(q\mathcal{U}r)\}\{\bigcirc\Diamond(p \wedge \neg(q\mathcal{U}r))\}$	state L_1 and error state L_2
$\neg(q\mathcal{U}r)$	apply $[r\neg\mathcal{U}]$
$\{\neg q, \neg r, \bigcirc true\}\{\neg r, \bigcirc\neg(q\mathcal{U}r)\}$	

The application of the translation algorithm gives the following automata, which is depicted in Figure 5.8.a:

$$\begin{aligned} Q &- \{L_0, L_1, L_2\} \\ I &= \{L_0\} \\ T &= \{L_2\} \\ E &- \{(L_0, p \wedge \neg q \wedge \neg r, L_2), (L_0, true, L_0), (L_0, p \wedge \neg r, L_1), (L_1, \neg r, L_1), (L_1, \neg q \wedge r, L_2)\} \end{aligned}$$

According to the Theorem 1, the NFA automaton can be transformed into a deterministic one. For that, in order to build a complete automaton, each combination of the conjunction of p, $\neg p$, q, $\neg q$, r, $\neg r$ have to be interpreted for each state. For example, from the initial state, because of the transition with the label *true* from L_0 to L_0, all the transitions with the label of all conjunctions of p, $\neg p$, q, $\neg q$, r, $\neg r$ are built. Given this, the method to construct deterministic automata from a non-deterministic can be applied, as depicted hereafter: (The indice U is used for the



(a) Non-deterministic (b) Deterministic

Figure 5.8: Automata for $\neg\Box(p \rightarrow p\mathcal{U}q)$

un-deterministic automata. The indice D is used for the deterministic one.) Notice that (Q_U, I_U, T_U, E_U) is the automata depicted in Figure 5.8.a.

$$
\begin{aligned}
Q_U &= \{L_0, L_1, L_2\} \\
I_U &= \{L_0\} \\
T_U &= \{L_2\} \\
E_U &= \{(L_0, p\wedge q\wedge r, L_0), (L_0, p\wedge q\wedge\neg r, L_0), (L_0, p\wedge\neg q\wedge r, L_0), (L_0, p\wedge\neg q\wedge\neg r, L_0), \\
&\quad (L_0, \neg p\wedge q\wedge r, L_0), (L_0, \neg p\wedge q\wedge\neg r, L_0), (L_0, \neg p\wedge\neg q\wedge r, L_0), (L_0, \neg p\wedge\neg q\wedge\neg r, L_0), \\
&\quad (L_0, p\wedge\neg q\wedge\neg r, L_2), \\
&\quad (L_0, p\wedge q\wedge\neg r, L_1), (L_0, p\wedge\neg q\wedge\neg r, L_1), \\
&\quad (L_1, p\wedge\neg q\wedge\neg r, L_1), (L_1, \neg p\wedge\neg q\wedge\neg r, L_1), (L_1, p\wedge q\wedge\neg r, L_1), (L_1, \neg p\wedge q\wedge\neg r, L_1), \\
&\quad (L_1, p\wedge\neg q\wedge\neg r, L_2), (L_1, \neg p\wedge\neg q\wedge\neg r, L_2)\}
\end{aligned}
$$

$$
\begin{aligned}
Q_D &= \{L_0, L_{01}, L_{02}, L_{012}, L_1, L_{12}, L_2, L_\emptyset\} \\
I_D &= \{L_0\} \\
T_D &= \{L_{02}, L_{12}, L_2, L_{012}\}
\end{aligned}
$$

E_D is given by:

	$p\wedge q\wedge r$	$p\wedge q\wedge\neg r$	$p\wedge\neg q\wedge r$	$p\wedge\neg q\wedge\neg r$	$\neg p\wedge q\wedge r$	$\neg p\wedge q\wedge\neg r$	$\neg p\wedge\neg q\wedge r$	$\neg p\wedge\neg q\wedge\neg r$
L_0	L_0	L_{01}	L_0	L_{012}	L_0	L_0	L_0	L_0
L_{01}	L_0	L_{01}	L_0	L_{012}	L_0	L_{01}	L_0	L_{012}
L_{02}	L_0	L_{01}	L_0	L_{012}	L_0	L_0	L_0	L_0
L_{012}	L_0	L_{01}	L_0	L_{012}	L_0	L_{01}	L_0	L_{012}
L_1	$L_\emptyset$	L_1	$L_\emptyset$	L_{12}	$L_\emptyset$	L_1	$L_\emptyset$	L_{12}
L_{12}	$L_\emptyset$	L_1	$L_\emptyset$	L_{12}	$L_\emptyset$	L_1	$L_\emptyset$	L_{12}
L_2	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$
$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$	$L_\emptyset$

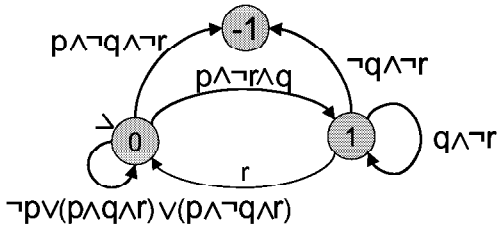
Then, after simplification of the states that cannot be reached from the initial one, the determination gives the automata depicted in the Figure 5.8.b, which accepts the same words as the non-deterministic of Figure 5.8.a.

The number of states after simplification of the DFA is similar to the state number of the NFA. This is not always the case.

5.8 Representation of the Automata

The graphical representation of the automata is not suitable for the Observer. A better representation is proposed in this section.

A deterministic automaton can be represented by a $A \times Q$ matrix called its *transition array*. We use "-1" to indicate an error state, and n a negative integer such that $n < -1$ to indicate each accepting state. For example, the transition array of the deterministic automata of the property $\Box(p \rightarrow q \mathcal{U} r)$ is represented in Figure 5.9.



(a) Automata

	0	1	-1
$p \wedge q \wedge r$	0	0	-
$p \wedge q \wedge \neg r$	1	1	-
$p \wedge \neg q \wedge r$	0	0	-
$p \wedge \neg q \wedge \neg r$	-1	-1	-
$\neg p \wedge q \wedge r$	0	0	-
$\neg p \wedge q \wedge \neg r$	0	1	-
$\neg p \wedge \neg q \wedge r$	0	0	-
$\neg p \wedge \neg q \wedge \neg r$	0	-1	-

(b) Transition Array

Figure 5.9: Transition Array for $\neg\Box(p \rightarrow q \mathcal{U} r)$

5.9 Simplification of the Automata

As stated in Section 5.3, the basic element for the expression of behavioral constraints is the notion of *events*. Considering that some events cannot happen at the same time, it is possible to simplify certain transitions of an automata. If two events e_1 and e_2, which cannot happen at the same time, appear in a transition of an automata under the form $e_1 \wedge e_2$, this transition can be removed since it will be never valid. This kind of simplification can be very useful in large automata in order to reduce the size of the transition matrix.

In the set of events that can be used to model distributed Object-oriented systems, only a few of them can occur simultaneously. They are given hereafter, assuming that object o and thread t are part of the process p, and *op* is an operation with the parameter *prm*. We use $e_1 \leftrightsquigarrow e_2$ to indicate that e_1 and e_2 are the same event and $e_1 \rightsquigarrow e_2$ to indicate that the occurrence of e_1 implies the occurrence of e_2 at the same time.

$$
\begin{array}{lcl}
(t\_outReq, *, (o_1, o_2, op, prm)) & \leftsquigarrow & (o\_outReq, (o_1, o_2, op, prm)) \\
(t\_assT, *, (o_1, o_2, op, prm)) & \leftsquigarrow & (o\_inReq, (o_1, o_2, op, prm)) \\
(t\_outRep, *, (o_1, o_2, op, prm) & \leftsquigarrow & (o\_outRep, (o_1, o_2, op, prm)) \\
(t\_inRep, *, (o_1, o_2, op, prm)) & \leftsquigarrow & (o\_inRep, (o_1, o_2, op, prm)) \\
(s\_delP, p) & \rightsquigarrow & (p\_delO, o) \wedge (p\_delT, t) \\
 & & \\
(p\_delO, o) & \not\leftsquigarrow & (p\_delT, t) \\
(p\_delO, o) & \not\rightarrow & (s\_delP, p) \\
(p\_delT, t) & \not\rightarrow & (s\_delP, p)
\end{array}
$$

5.10 Summary of the Translation Method

In summary, the translation method, that can be fully automatized, for a property f is given hereafter:

1. write the property $\neg f$,
2. apply the tableau algorithm to $\neg f$, and construct the automaton,
3. transform the automaton to include all conjunctions of the different elements in the transitions. For example, for two transitions p and q between the two states, there will be the following new transitions: $\neg p \wedge q$, $p \wedge \neg q$, $p \wedge q$ and $\neg p \wedge \neg q$.
4. apply the Theorem to obtain a deterministic automaton,
5. simplify the automaton.

5.11 Testing Architecture

The Observer is divided into two functional parts: a *"Monitoring Manager"* and a *"Testing Manager"*. In Figure 5.10, the structure of the Observer is depicted.

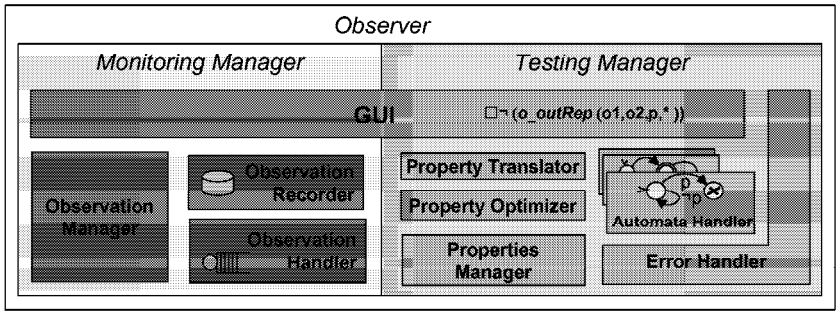


Figure 5.10: Observer Structure

5.11.1 The Monitoring Manager

The Monitoring Manager (defined in Chapter 4) is in charge of all the activities needed for the observation of the running implementation. It is responsible for setting up the instrumentation in order to get the needed information and handling the information received in order to ensure their consistency and, finally, for forwarding that information to the Testing Manager and/or to the Monitoring GUI. The Monitoring Manager is divided into an *Monitoring GUI, Observation Manager* an *Observation Handler* and an *Observation Recorder.*

Monitoring GUI By the means of the Monitoring GUI, the users have a functionality to select what information to observe, more precisely what event(s) are to be observed. This GUI also presents the different information observed, the observed information can be displayed according to different formats, such as a timeline diagram or a table.

Observation Manager The Observation Manager makes the interface with the system under scrutiny. It is able to configure the instrumentation in order to get reports only on the requested events and to receive the information under the forms of notifications. When a notification is received, the Observation Manager forwards it to The Observation Handler.

Observation Handler The Observation Handler guarantees the consistency of the notification received by the Observation Manager. It executes the reordering algorithms in the case of mis-ordered notifications. Then, it delivers the ordered notification to the Monitoring GUI and/or the Testing Manager and/or the Observation Recorder.

Observation Recorder The Observation Recorder can be activated by the Monitoring GUI, in order to record all the notifications that reach the Observation Handler. These records can then be used to analyzed the behavior of the system off-line, or to replay specific sequence of the testing scanario performed.

5.11.2 The Testing Manager

The Testing Manager is responsible for the testing of the behavioral constraints expressed by LTL properties, according to the knowledge of the behavior of the system, as reported by the Monitoring Manager. The Testing Manager is divided into a *Testing GUI,* a *Property Translator,* an *Automata Optimizer,* a *Property Manager* and several *Automaton Handlers*, one per property.

Testing GUI The Testing GUI offers to the users the functionality of expressing behavioral properties. Based on the IDL specification of the application under

scrutiny, different menus are offered to select the different events that can be part of a property and other menus are offered with the LTL operators in order to write the properties that have to be checked at run-time.

Property Translator The Property Translator builds the automata corresponding to the properties expressed by the user via the Testing GUI. The output of the property translator is an automaton for each property (states and transitions).

Automata Optimizer The role of the Automata Optimizer is to take the automata produced by the Property Translator, and to optimize each automata, i.e, to transform each automata into a deterministic one, and to simplify the transitions that can never be true. The output of the Automata Optimizer is the transition matrix for each automaton.

Property Manager The Property Manager is responsible for receiving the notification from the Observation Handler and for dispatching them to each Automata. It manages the different Automata Handlers. It also gives a test result at the end of the test sequence (test statistic, number of events, number of properties violated, events that produced errors, etc.)

Automaton Handler Each Automaton Handler contains an automaton, and is responsible for its management. According to the information received, it orders to cross a specific transition, and to store the current state. When an error occurs, an Error report is produced, which is then given to the Properties Manager for the test sequence report.

5.12 Testing Scenario

The testing scenario is depicted in Figure 5.11. When the user enters a property, this property is first given to the property translator (1) and at the same time given to the Observation Manager (1'). When the property has been translated, it is handled by the Property Optimizer (2), which then gives the automata to the Property Manager (3) and creates an Automata Handler (4) for that property. In parallel, the Observation Manager sets up (2') the instrumentation of the code in order to receive notifications (5) when the system is running. These notifications are sent to the Observation Manager which will then forward them (6) to the Observation Handler. The Observation Handler forwards the notification, which have been previously reordered, to the Observation recorder (7) and, the Properties Manager and the GUI (8). The Properties Manager dispatches the information received to each Automata Handler (9), which uses this information to check if this new input will allow the automata to fall in an error-state and thus generates an error (10).

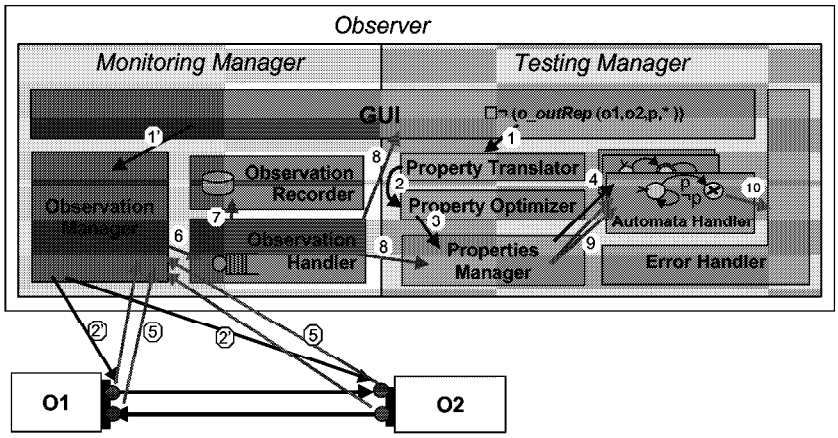


Figure 5.11: Testing Scenario

5.13 Performances Improvements

One of the major drawbacks of our monitoring/testing mechanism is the central Observer. It is somehow disappointing to need to centralize the testing mechanism when the application is fully distributed. The major performance enhancement that can be performed is to distribute the functions of the Observer.

There are different approaches to achieve the distribution of an Observer mechanism. The first, presented in Section 5.13.1, consists in the addition of an "Observer Agent", on each workstation that hosts a piece of the distributed application. Each of these Observer Agents receives the information from the processes and objects residing on the same workstation and then forwards them to the central Observer.

The second is based on the notion of "Mobile Automata", presented in Section 5.13.2, where the central Observer downloads the automata on the different entities of the network and these automata are "piggybacked" between the objects.

5.13.1 Observer Agents

We explore two solutions to build Observer Agents that we call *Full Observer Agents* and *Partial Observer Agents*. We outline these two solutions in the next two sections, respectively Section 5.13.1.1 and Section 5.13.1.2. With the Full Observer Agents, a new process is added on the different nodes of the distributed system in order to play the role of an Observer for the local activities of the application. In the second proposal, Partial Observer Agents, an "agent" is added in the processes.

5.13.1.1 Full Observer Agent

In [AAM97], Al-Shaer et al propose a monitoring approach for the monitoring of large-scale distributed multimedia systems. This work is based on *Monitoring Agents*, that are of two categories: *local monitoring agents (LMA)* and *domain monitoring agents (DMA)*. The former are responsible for the detection of primitive events generated by local applications in the same machine while the latter is responsible for detecting composite events that are beyond the scope of LMAs. The LMAs forward the events to their conducting DMA, which will then forward the info to the consumer (user) or accomplish the action involved by the reception of that specific event.

In this design, an Observer Agent is added in each workstation, handling all that can be handled locally and reporting the rest to the central Observer. We can apply the same strategy in our system.

5.13.1.2 Partial Observer Agent

If the property involves only events relative to the same process, the automata can be moved to that process, and checked by the process itself. A message is sent to the Observer when the property is violated or when it cannot be violated anymore. The instrumentation is similar to the one described in Section 4.7, except that instead of sending a notification to the Observer for the events involved in the property, we only execute a local function to handle the automata, and check if a transition becomes true. This function has to be integrated by the automitized instrumentation of the code. In order for the Observer to be able to download the automata to the specific process, a function has to be added in the interface of the process.

Let us examine this scenario with an example. Assuming that we have a system with one user u, a video on demand server vs and a billing server bs. Among the service -operation- that the server offers, there is the `request_movie(movie_name)` operation. On the user side there are also some operations that are offered, e.g., `billing(movie_name,price)`. Based on this, we can specify a property stating that "when the user has requested a movie it will then be billed". The LTL property expressing the same is:

$$\Box(\odot(p\_outreq, (u, vs, request\_movie, (movie\_name))) \rightarrow \\ \Diamond \odot (p\_inreq, (bs, u, billing, (movie\_name, *))))$$

When this property has been entered through the Observer and because it involves the process u, the automata of the property can be downloaded to the process u, by the operation `dowload_automata(transition_matrix)`, which has been added by the *code generator* at compilation time. When the event involved in the property occurs, the operation to manage the automata is called, and the property is checked. If an error occurs, a notification is sent to the Observer. For that, the notification type `error_notification` is added in the Observer.

Notice that in order to take advantage of the possibility to un-load the Observer by downloading automata to processes, the properties have to be expressed carefully. The following example expresses the same property as the above example but cannot be downloaded!

$$\Box(\odot(p\_outreq, (u, vs, request\_movie, (movie\_name))) \rightarrow \\ \Diamond \odot (p\_outreq, (bs, u, billing, (movie\_name, *))))$$

The event $(p\_outreq, (bs, u, billing, (movie\_name, *)))$ occurs at the bs whereas the event $(p\_outreq, (u, vs, request\_movie, (movie\_name))$ occurs at u.

5.13.2 Mobile Automata

The use of the Observer Agent is very useful in the case of properties involving only events occurring in the same process, but is useless for all other properties. In this section, we propose to break the automata relative to a property and distribute the different pieces into the process where the events involved in the property occur.

With this approach, the automata that have to be checked are broken into different parts that are handled by different sites of the distributed application and by the "addition" of messages to communicate between the different parts of the automata.

If the event provokes the reach of an error state (-1) then the process will raise an error. In other words an error-notification is sent to the central Observer. This notification contains the property id, the event producing the error. After having sent the error-notification, the process will also send a message to the other process in order to stop the handling of that property.

The two following examples illustrate the algorithm.

5.13.2.1 Example 1, Precedence property

Let us take the example of the precedence property $\Box(e_1 \rightarrow (\neg e_2)\, \mathcal{U} e_3)$ in order to illustrate the distributed testing of properties. Considering 3 events, e_1, e_2 and e_3, and 3 processes, p_1, p_2 and p_3, there are 5 combinations of $\pi(e_1)$, $\pi(e_2)$, and $\pi(e_3)$ as presented in Table 5.5. The function π is defined in Section 4.7.2.1 and gives the process id in which an event occurs.

Table 5.5: Combination of events and processes for a precedence property

	$\pi(e_1)$	$\pi(e_2)$	$\pi(e_3)$
case 1	p_1	p_1	p_1
case 2	p_1	p_1	p_2
case 3	p_1	p_2	p_2
case 4	p_2	p_2	p_1
case 5	p_1	p_2	p_3

Let us now outline how it would work for each of the five cases:

case 1 All the events occurs in the same process. This case is equivalent to the proposal of Observer Agent in each process. The details are presented in Section 5.13.1.

case 2 When e_1 occurs, p_1 starts to wait for e_2. If e_2 occurs, then p_1 will raise an error. If e_3 occurs, p_2 is informed to stop handling the property, and a message is sent to p_1 in order to continue the checking (catch the next occurrence of e_1).

case 3 When e_1 occurs, p_1 sends a message to p_2. If e_2 occurs, then p_2 will raise an error; whereas, if e_3 occurs, p_2 will stop handling the property and will send a message to p_1 to continue checking the prop. Then, p_1 will wait for the next occurrence of e_1.

case 4 When e_1 occurs, a message is sent by p_1 to p_2, and p_1 will start to wait for e_3. If e_2 occurs, p_2 will raise an error. If e_3 occurs, p_1 will start again to wait for e_1 and will send a message to p_2 to stop waiting for e_2.

case 5 When e_1 occurs, p_1 sends a message to p_2 and p_3, which will respectively wait for e_2 and $e3$. If e_2 occurs p_2 will raise an error, if e_3 occurs, p_3 will send a message to p_2 to stop handling the property and to p_1 which will wait for the next occurrence of e_1.

5.13.2.2 Example 2, response property

In this example we examine the distributed checking for the property $\Box(e_1 \rightarrow \Diamond e_2)$. If the system is composed of two processes, the different combinations are presented in Table 5.6

Table 5.6: Combination of events and processes for a response property

	$\pi(e_1)$	$\pi(e_2)$
case 1	p_1	p_1
case 2	p_1	p_2

case 1 All the events occur in the same process. This case is equivalent to the proposal of Observer Agent in each process. The details are presented in Section 5.13.1.

case 2 When e_1 occurs, p_1 sends a message to p_2; p_2 will then start to handle ∞ (e.g., counting the number of events or check a duration). If e_2 occurs, p_2 sends a message to p_1 and continue to handle ∞.

5.13.3 Discussion

There are different criteria that can be used in order to judge the quality and performance of our monitoring/testing mechanism:

- Size and number of notifications

- Size and number of control messages
- Solving of the ordering problem
- Possibility to automatize different steps
- Impact on the Automata
- Impact on the complexity of the Central Observer
- Use of the ORB
- Intrusive effect
- Overall complexity
- Scalability
- Overall feasibility and implementability

In Table 5.7, a summary of the different approaches for the distribution of the testing mechanism is presented. From that, we can notice that there is always a weakness for each solution and that it is not possible to extract a best solution for all cases.

For example, concerning the distribution of the automata, in the case of a large number of automata, it is not clear that the distribution will be a benefit : perhaps the number of message between the different objects will disturb more the system than notifications to a central observer.

The area of "distributed testing" is left for further study, that would be possible after experience with and measurements of the proposed solution in many different contexts.

5.14 Related Work

There is considerable work on the use of formal methods for testing. Many propose new features in existing Formal languages (e.g, real-time LOTOS, extended LOTOS [Bri88]) but few of them look at the applicability to industrial problems. Many of them also focus on the validation of an abstract model and do not look at the final and operational implementation. In [LDH+99a], we present challenges and direction for the use of formal methods in the area of hybrid services. We conclude that to be accepted by the industry, a testing method based on a Formal Technique has to focus on the implementation and should hide as much as possible all the details of the Formal Technique (language, symbols, traslation, etc.).

At the University of Santa Barbara, Dillon et al. [DR96, DY94] indicate that temporal logic can serve as a suitable vehicle for testing. However, in their approach test, traces are obtained by manually instrumenting Ada source programs and executing it on a uniprocessor. Delay statements were inserted to introduce different behaviors. In a practically feasible framework, these tasks should be handled by a tool and remain hidden from the service tester. In their approach they also developed a new tableau algorithm for the translation of temporal logic properties into automata.

Table 5.7: Summary of the different distribution techniques

	Observer Agent		Distributed
	Partial	Full	
Notifications type	$\nearrow$, new type to signal local errors	$\nearrow$, new type to signal local errors	$\nearrow$, new type to signal errors
Number of notifications	$\searrow$	$\searrow$, but replaced by local notifications	$\searrow$
Ordering problem	$=$, as with central Observer	$=$, as central Observer	$=$, as central Observer
Automation	feasible	feasible only in the case it is possible to execute new processes on remote sites	feasible
Automata	download to specific processes	download to specific Agent	Break them into pieces that are downloaded among the processes
Complexity of central Observer	$\nearrow$, new operation for the new notification type, new functionality to download Automata	$\nearrow$,new operation for the new notification type, new functionality to download Automata	$\nearrow$, Analyze of the properties to distribute the different part of the automata in the system
use of ORB	as with central Observer, but with less messages on the network	as with central Observer, but with less messages on the network, but more messages locally	more message between the different entities of the system, but less to the central Observer
Intrusive effect	can be large for a process	can be large for the site, depending on its load, etc.	more impact on each process and on the network, but less bottleneck effect
Overall complexity	$\cong$	$\searrow$	$\nearrow$
Unload of Observer	properties involving only single processes	the properties involving only single sites	All properties can be distributed
Others	-	-	new message type to forward the automata status between the processes

At AT&T, Jagadeesan, Puchol et al. [JPO95, JPP+97] developed different approaches, patterned to the work of Dillon et al., to verify temporal properties ex-

pressed using Esterel. In [JPO95] they used their method to test 30 different implementations of a telephone switching system. They concluded that the approach is highly effective at finding defects in the implementation. This is one more argument to advocate in the favour of the approach developed in this thesis. Their proposal is very similar to ours but is restricted to safety properties and does not consider middlewares.

Holzmann, also working at AT&T, is the inventor of a formal language called Promela associated with its validation tool SPIN. Promela-SPIN is very efficient protocol validation method [Hol91, Hol97]. In [Hol94], Holzmann describes the NewCoRe project. The goal of the NewCoRe project was to test the viability of formal verification techniques on a regular industrial-size design project. The target application was a portion of Signalling System 7 in the 5ESS switching system: the ISDN User-Part protocol as defined by the CCITT. The final verified model contained 7500 lines of non-commented SDL source. A total of 145 correctness properties was formalized in three temporal logic property classes: this corresponds to the three property classes proposed by Manna and Pnueli [MP91b]. More than 100 serious design errors were uncovered in the design requirements, by automated verification runs alone, and could have been prevented from reaching the product, without requiring test-fields. Holzmann claims that the use of automated formal verification techniques for industrial software design had never been attempted on such a scale before. To our knowledge, this fact is still true.

In [Eti95], Etique developed a formal language, called FUS++, derived from the analysis model of the object-oriented development method Fusion [CAB+94] for the specification of services of the Intelligent Network. The model of the service is then translated into Promela [Hol91] , a protocol specification language that is associated with the verification tool (SPIN) [Hol97], by an adequate compiler, in order to be validated.

Convinced of the problem of state space exploration and thus of the limited coverage of the test, Etique also developed an Observer called Service Modeller and Verifier Function (SMVF) for the Intelligent Network. This Observer, is fed by temporal properties, expressed using FUS++, that are checked against the FUS++ model specification of the services that is updated at run-time by the observation of the system. In fact, we worked on that aspect [Log96, EHL98]in a project called "Ernestine" with Swiss Telecom PTT. The work presented in this thesis is, to some extent, the continuation of the above mentioned work in the area of modern communication services built on top of middleware. One of the major differences, beside the different environment, is the absence of a complete formal specification of the service in our case, which is an important advantage considering the acceptance of formal methods in the industry.

Temporal logic has also been used in the project described in [SMC+96] which deserves special attention since it describes work that made the step into a commercial product. For the specification of services in the Intelligent Network, Siemens Nixdorf Informationssysteme, Munich, Germany and the University of Passau, Germany, developed an environment for the creation of Intelligent Network services. The services

described in this framework can be formally verified by model checking. The service properties are expressed using temporal logic. By making the use of formally specified constraints an option, it is up to the service designer to decide to which degree he is willing to invest into formality; the specification of more constraints leads to a more faithful verification of the service design.

The Temporal Rover [TR] is a specification based verification tool sold by Time Rover. Within specially marked comments it is possible to write formal specifications inside source files (e.g., C++, Java). The formal specifications have to be written by combining Linear-time Temporal Logic and the language of choice (e.g., Java). The formal specification is converted into executable code to be executed when the program is being tested. An example of a property specification with Temporal Rover is given in Figure 5.12.

```
/* TRBegin
// Asserting that if light is Green, camera is not on.
// This assertion detects a bug where the camera operates one cycle
// too much and actually takes shots of cars going through a green
// light.
   TRAssert{ []( {isGreen()} -> {!isCameraShooting()} )
           } =>
   {System.out.println("Assertion: SUCCESS");
    System.out.println("Assertion: FAIL");
    System.out.println("Assertion DONE!");
   }
TREnd*/
```

Figure 5.12: Example TemporalRover

As it can be seen from the example, the expression power of The Time Rover solution is not sufficient enough to be applied in the case of communications services. Their solution also suffers from the manual annotation of the code.

5.15 Conclusion

In this Chapter we presented our testing approach based on the monitoring method we also developed in this thesis. We first presented the way the tester specifies behavioral constraints using LTL. Then we explained how to translate these formulas into automata, how to build deterministic automata and how to optimize the automata, in order to be checked at run-time. We defined the architecture of the testing mechanism to introduce in the Observer that was presented in Chapter 4. Finally, the typical testing process is described.

The contributions of this chapter are as follows:

- *we designed a method for the testing of communications services.* Based on the information that can be gathered at run-time by the monitoring mechanism,

we developed a method that is efficient to check if the actual implementation conforms to the requirements. The requirements are expressed using LTL. One of the main advantages of our testing methodology is that it does not require the formal specification of the entire system to test. The tester can focus only on the behavioral function that is of interest. Having expressed what to check, all the remaining operations are automatically done for him. Our method also provides a help for the expression of properties, since it is possible to generate menus helping the developer.

- *we designed the architecture of the Observer for the testing.* This architecture is made of different modules that have a specific task: the translation of properties, the optimization of automata, the management of the automata, the checking of the automata, the generation of error reports. This structure leads to a testing scenario that is efficient and practical. This scenario requires from the developer the specification of the properties and the remaining steps are performed automatically.
- *we showed how to translate LTL properties into automata* We came up with an algorithm for the translation of properties that builds automata that are deterministic as well as optimized in their number of states. The translation of the properties into deterministic automata is an interesting feature of our approach because it avoids addressing the complicated handling of automata with non-deterministic choices.

As it was mentioned in Chapter 4, the central Observer might be a bottleneck in the system and thus disturb the behavior of the system being test. We also have given some avenues to follow in order to solve that problem. The one that seems the more convincing consists in performing a distributed checking of properties. This might be a challenging domain to explore.

In the following chapter we present the prototype called MOTEL (MOnitoring and TEsting tooL) that we implemented based on the methods explained in Chapter 4 and in this Chapter.

Chapter 6

MOTEL - MOnitoring and TEsting tooL

The prototype implementation presented in this Chapter was used as the implementation illustration for the patent [LDH+99b] and delivered to Alcatel Research, France

6.1 Chapter Overview

In this chapter, we present the implementation of a prototype we call MOTEL (MOnitoring and TEsting tooL) that reflects the monitoring methods developed in Chapter 4 and the testing method of Chapter 5. We present its use, how it can be introduced in industrials platforms, the performances and finally its experience report.

6.2 Introduction

During our collaboration with Alcatel and Swisscom, we implemented two prototypes. The first was developed on top of HP UX workstation using CORBA Orbix^{TM} 2.2 from IONA. For this, we used C++ and Expectk/Tcl/Tk for the GUIs. The second prototype was developed on Windows NT OS, using CORBA OrbixWeb^{TM} 2.3. The language used for the second prototype was JAVA. In the following sections we present the first one, which is currently being introduced into the PERCO platform of Alcatel.

6.3 MOTEL Implementation

MOTEL is made up of two different parts: (1) A *generic* part and (2) a *specific* part. The *generic* part does not depend on the environment and consists of the mechanism and algorithms used to analyze, display and treat the -monitored- information retrieved from the distributed application. The *specific* part depends on the middleware and deals with the mechanisms -base on addition of instrumentation- to obtain

the information needed from the application, and the protocol used to communicate the information to the generic part. The specific part is implemented by the *Observation Manager* since it is the only part that interacts with the system.

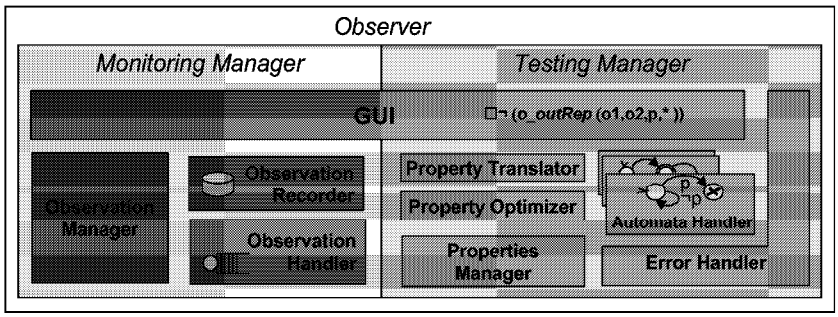


Figure 6.1: Observer Structure

We present the *generic* part in Section 6.3.1. The specific part is explained and detailed in Section 6.3.2.

6.3.1 MOTEL's Graphical User Interfaces

The GUI of the tool is divided into four parts:

- The *main* GUI, with the monitoring messages and the choice to select other options;
- The *Properties* GUI, with the list of properties to be checked in the implementation and the capability to edit them (testing specific);
- The *Selection* GUI, with the list of information to be monitored and the possibility to select them (monitoring specific);
- The *Automata* GUI, with the schema of the properties represented as automata (testing specific).

In Section 6.3.1.1, the main GUI is presented. In Section 6.3.1.2, the Properties GUI is detailed. The GUI representing the representation of the properties as FSMs is provided in Section 6.3.1.3.

6.3.1.1 Main MOTEL GUI

The main GUI of MOTEL offers different options for testing and monitoring.

The principal concept behind the testing is the definition of properties to be checked. The definition of properties is offered by the option *Properties*. The details concerning the Properties' part is given in Section 6.3.1.2.

In order to display the monitored information, two options are available: a *Table* and a *Timeline*. The result of the *Table* options is presented in Figure 6.3, under the

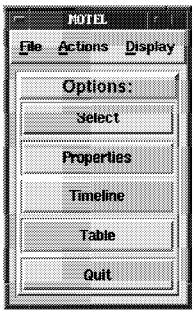


Figure 6.2: MOTEL main window

Observed Messages window part. In the *Observed Messages* of the main MOTEL GUI part, different information about notifications received is displayed. This information is presented in Table 6.1.

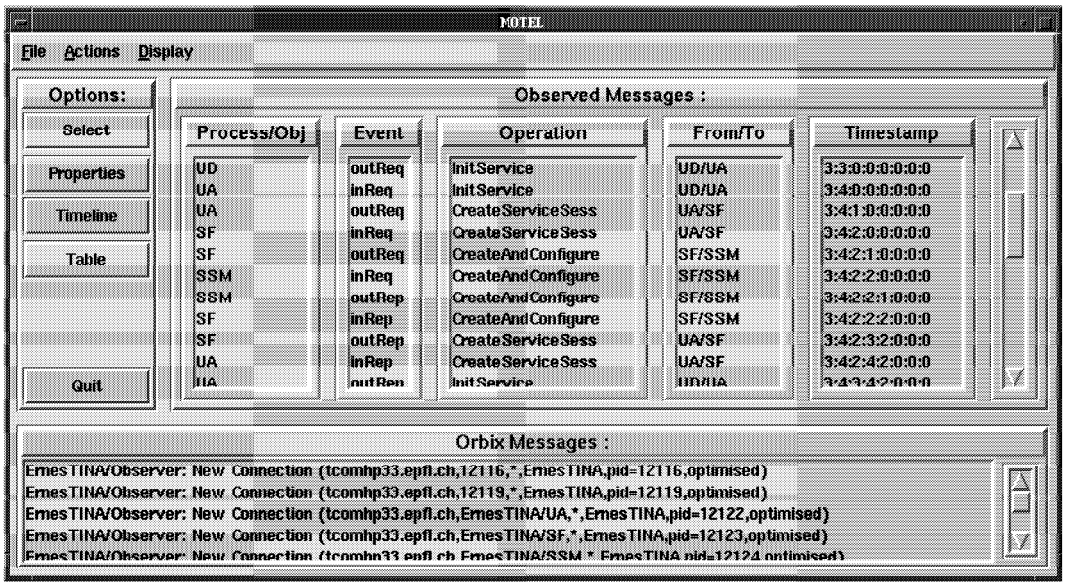


Figure 6.3: MOTEL main window with the options "table" and "Orbix Messages"

With the *Timeline* options, the monitored information is displayed under the form of a timeline, as depicted in Figure 6.4. This timeline allows the sequences of operation in the distributed application to be followed closely.

MOTEL offers the possibility for the different messages given by the ORB to be displayed. In Figure 6.3, the *Orbix message* part gives the Orbix ORB messages.

6.3.1.2 Properties' GUI

In order to define the properties to be tested in the system, the *Properties* GUI proposes different options:

Table 6.1: Monitored Information

Parameters	Information
Process/Obj	Identification of the process and/or object sending the notification
Event	Event identification (e.g., out request)
Operation	Name of the operation invoked
From	Identification of the object invoking the operation that is the event source
To	Identification of the object on which the operation is invoked
Timestamp	Timestamp of the process when the notification was sent

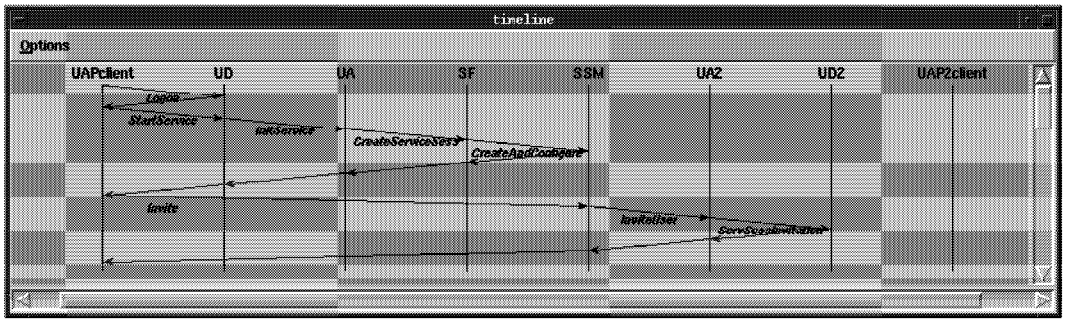


Figure 6.4: Timeline Diagram of the monitored information

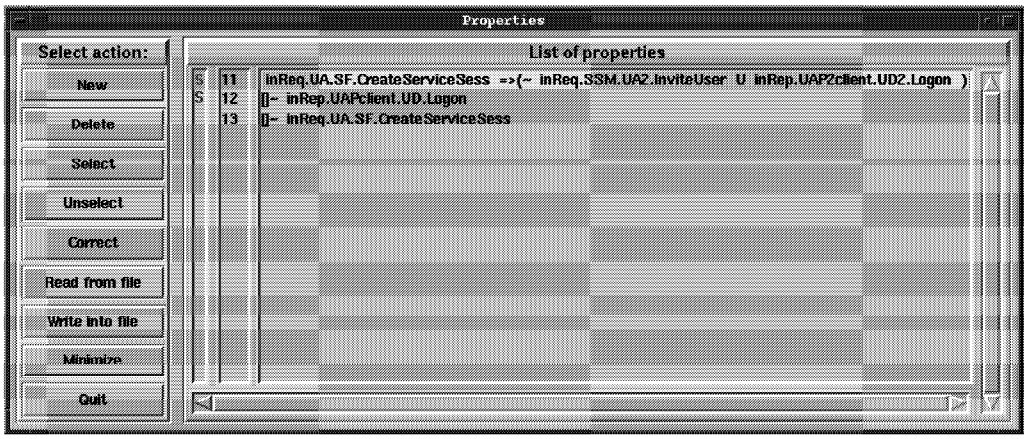


Figure 6.5: Properties'part main window

- *New*: Raise a new window to edit a new property (Figure 6.6),
- *Delete*: Delete the property selected in the list,
- *Select*: Put the property selected in the set of property to be checked. Those properties are identified by an "S" in the *List of properties* window part,
- *Unselect*: Remove the selected property from the list of properties to be checked,
- *Correct*: Raise a window allowing to edit and correct the selected property,

- *Read from file*: Read the list of properties that has been saved into a file,
- *Write into file*: Write the current list of properties from the file,
- *Minimize*: Iconify the window,
- *Quit*: Quit the testing part of MOTEL.

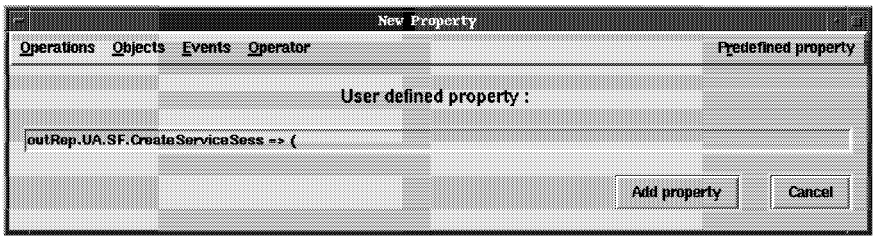


Figure 6.6: Definition of a new property

In Figure 6.6, the interface to specify and edit properties is shown. The different menus (*Operations, Objects, Events* and *Operators*) offers the items that are needed to specify the property. For example, the menu *Operations* offers to select one among all the operations of the objects within the system. The *Predefined Property* gives a new window, as in Figure 6.7. When the edition is finished, the button *Add property* adds the property in the list.

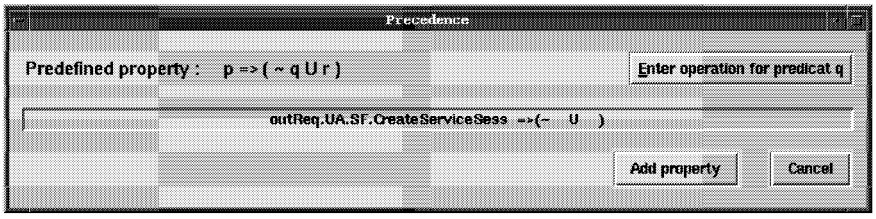


Figure 6.7: Definition of a new precedence property

This menu allows to define from three types of properties: *Invariant, Precedence* and *Response*, as defined in [MP91b]. After having chosen one of these types, the user only has to enter the value of the predicates, as depicted in Figure 6.7.

6.3.1.3 Automata

When the first property is selected in the *Properties* GUI (Figure 6.5) the user will be asked if he wants to see the automata. If the user responds "yes" then a window such as the one depicted in Figure 6.8 will appear displaying the automata (FSMs) corresponding to each of the selected property.

Even if the *Automata* window has been selected, a window will raise when a selected property is violated. Such a window is depicted in Figure 6.9. This window gives the reference number of the violated property and reports the event that provoked the violation.

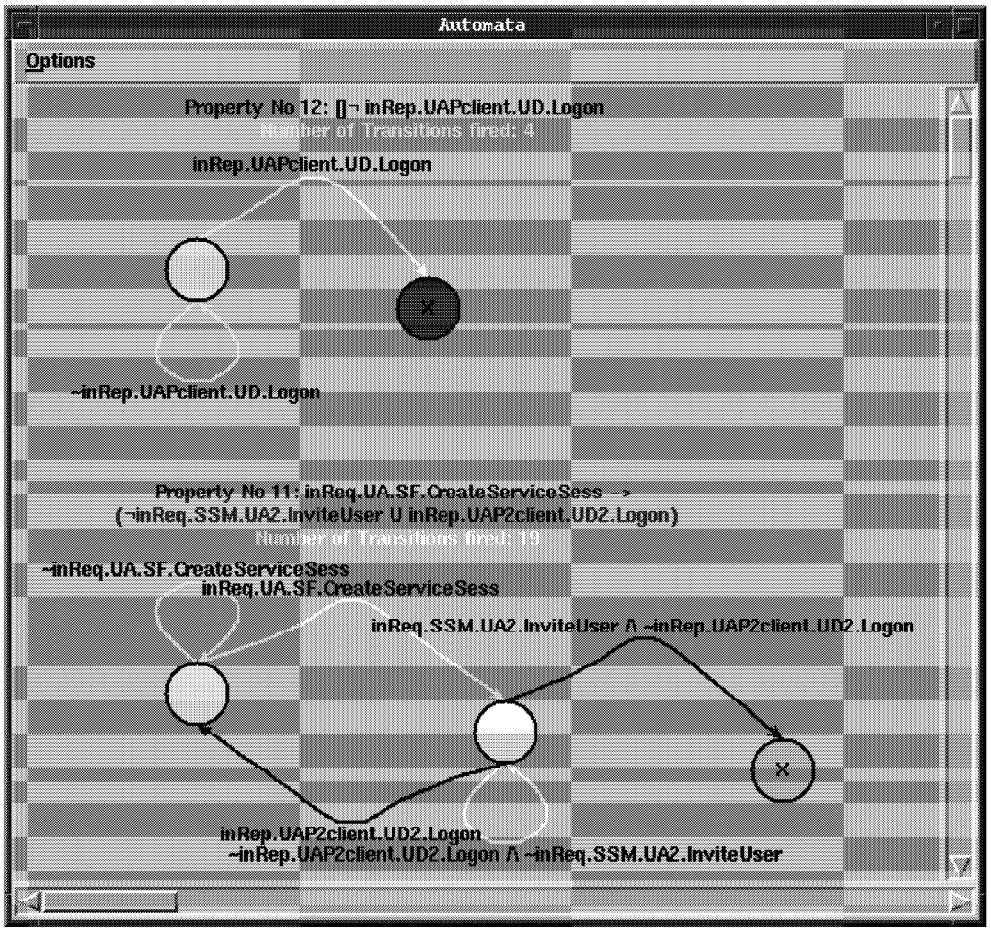


Figure 6.8: Display of the Automata corresponding to the selected properties

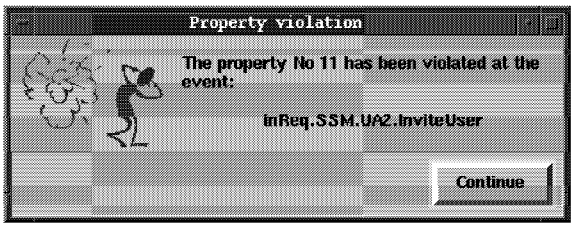


Figure 6.9: Display of the violation of a property

6.3.2 Instrumentation

In order to spy on the distributed system, we use the *Filtering* mechanism provided by Orbix (see Section 2.6.9.1), which is of two forms: *per-process* and *per-object* filtering. Per-process filters monitor all incoming and out-going operations to and from an address space; per-process filtering is applied when an invocation leaves or arrives at an address space. Per-object filters apply to individual objects. The observation is based on four forms[1] of per-process filters *out request, in request, out*

[1]Orbix provides eight points for per-process filtering but only four are useful in our case

reply, in reply and two forms of per-object filtering *pre, post.* The filtering mechanism offers the possibility of piggy-backing information with the requests. For example, at the 'out request filter' extra data can be added to the request, as long as the 'in request filter' removes it afterwards.

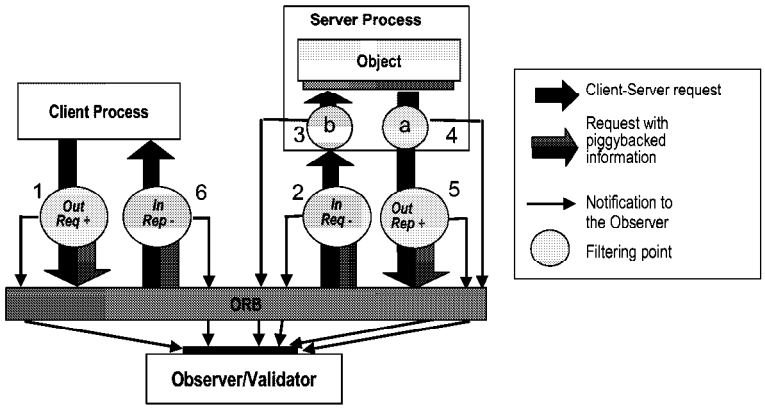


Figure 6.10: Observation mechanism based on process filters

In Figure 6.10, the different filters are illustrated in the case of an object in a client process that invokes an operation on an object within a server process. When the request exits the object, the first filter point (out request) is reached. At that point, a notification is sent to the observer, the timestamp algorithm is executed, and then the timestamp and useful information are piggybacked to the other object. The request is then proceeded by the ORB and reaches the server process. When the 'in request' filtering point is reached, the piggybacked information is extracted and removed from the request and a notification is issued. The server will then handle the piggybacked timestamp, update its current one and forward the request to the right object. When the request reaches the object, the Object filtering point 'pre' is reached and a notification is sent. The same scenario is applied for the reply phase of the request in the case of an operation request (the chain of filters being in that case: post, out reply and in reply). In the case of a one-way operation, no return is sent.

According to the Orbix Filters we have six filters, which can be mapped to our events as indicated in Table 6.2.

#	Orbix Filter Level	Event
1	process	*t_outReq*
2	process	*p_inReq*
3	object	*o_inReq*
4	object	*o_outRep*
5	process	*t_outRep*
6	process	*t_inRep*

Table 6.2: Mapping Orbix Filters to Events

Furthermore, the Orbix run-time system delivers a few notifications by default. These notifications and their mapping to the events in our model are summarized in Table 6.3.

Notification	Event
New Connection (server ready)	*s_newP*
End of Connection	*s_delP*

Table 6.3: Mapping Orbix Messages to Events

Some events from our model can be mapped in a straightforward way to specific Orbix functions. For example, in Orbix there is a function *_bind()* that finds a particular object and sets up a proxy for it in the client's address space. It is possible to specify the exact object required or, by using default parameters, Orbix may be allowed certain degrees of freedom when choosing the object. This function corresponds to our *p_reqRef* and *p_recRef* events.

In Figure 6.11 the code that is executed in a Filter is presented. The 1^{st} line is the Orbix standard operation for the out request filter. On the 3^{rd} line the event type is set. Then (4^{th} line), we check if the operation intercepted at the filtering point is an operation defined by the user or if it is an operation of the ORB/deamon. If it is a user operation (5^{th} line) a notification is sent to the Observer with the following arguments:

- 5^{th} line: The name of the process where the event occurs,
- 6^{th} line: the Event type,
- 7^{th} line: the name of the operation,
- 8^{th} line: the name of the process on which the operation is invoked,
- 9^{th} line: the timestamp.

Then the timestamp is updated (11^{th} line) and piggybacked in the request carrying the operation invocation (11^{th} line).

6.4 Performances

As already stated in Section 4.9, when monitoring a system, it is essential to know if a monitoring mechanism changes the behavior of the system under scrutiny. We showed that our monitoring mechanism might change the order of the events occurring in the system. In this section we are interested in the intrusive effect of the mechanism with regards to the performance's degradation in terms of time.

6.4.1 Experimental measurements of the impact

We performed measurements in order to quantify the typical intrusive effect of our observation mechanism. We measured the time τ for **the worst case scenario**

```
1 CORBA::Boolean PrFilters::outRequestPreMarshal(CORBA::Request& r,
2                                                CORBA::Environment&)
3 { CORBA::string FilterName="outReq";
4   if (IsInTargetList(r) && IsNotPingAndObserverFunction(r)) {
5       Notify (PrName,
6               FilterName,
7               r.operation(),
8               GetTargetPrName(r),
9               TimeStamp );
10      UpdateTimeStamp();
11      InsertTimeStampInReq(r);
12  }
13  return 1;
14 }
```

Figure 6.11: Example the code executed in a Filter

of an invocation of an operation and the reception of the result, from a client to a server with Orbix 2.3 from IONA on workstations running the HP UX.10 OS., as depicted in Figure 6.12. Indeed, we took the worst case, because we measured the impact when all events are reported to the Observer (which is not always the case, since it depends on the events that are used to express the properties) and when the processing time of the operation is null..

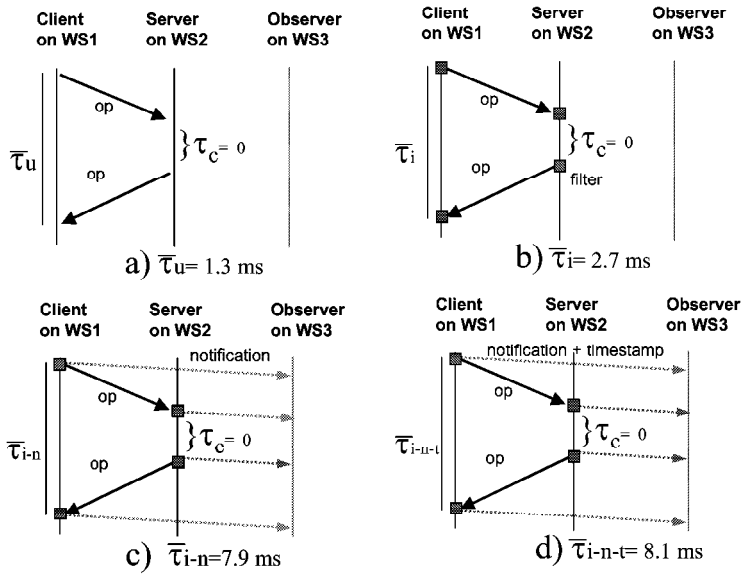


Figure 6.12: Measure of the worst case scenario

This basic configuration allows for the exact measure we are interested in to be extracted, independent of other factors (workstations load, network load, etc.), which become important (and random) in a more complex configuration. In the configuration investigated, the client, the server and the Observer work on different workstations in the same Ethernet LAN. The setup chosen was as follows: uninstrumented

system ($\overline{\tau}_u$ in Figure 6.12 (a)), system with the instrumentation but without sending notifications ($\overline{\tau}_i$in Figure 6.12 (b)), system with the instrumentation and the notifications but without the timestamp mechanism ($\overline{\tau}_{i-n}$ in Figure 6.12 (c)), system with the instrumentation, the notifications, and the timestamps ($\overline{\tau}_{i-n-t}$ in Figure 6.12 (d)). In order to inform the Observer, 5 requests (1 operation + 4 notifications)[2] are sent by our mechanism. Our machinery also adds the causal ordering algorithm. We can therefore suppose that our monitoring mechanism adds at least "4 invocation durations" to the execution of the uninstrumented system, which is the *worst case* since each event leads to a notification. The results are presented in Figure 6.12. We noticed that during these experiments, 10% to 25% of the notifications arrived at the Observer out of order, which confirms the need for a reordering mechanism.

The standard deviations measured are as follows: case a) 0.7 ms, case b) 1.1 ms, case c) 1.9 ms and case d) 2 ms. The relatively large values of the standard deviations measured can be explained by the platform on which the system is running. All the resources are shared: the network is shared by the workstations, each CPU is also shared between different processes. In other words, the available resources at two different moments in time can differ, which implies some deviation in the time measured.

From the results of the measures, we notice that 6.8 ms have been added to the invocation time plus the return time of an operation. At first sight, this can be considered as a substantial impact. In the tested scenario 100% of the time measured was devoted to operation invocation and result reception. In a real case, the operation invoked will handle some computation (invocation of another operation, calculations, writing to a database, etc.), which will also take time. We call that, the computation time of an operation (τ_c). In our case, $\tau_c = 0$. The invocation plus the return time is therefore merged with the computation time of the operation. The impact of our monitoring mechanism will therefore also depend on the computation time of the operations in the system, which is difficult to measure a priori. In our setup, considering that the computation time of the operation would be τ_c, the time to get the result of the operation in the case of an uninstrumented (τ_u) and the fully instrumented system (τ_{i-n-t}), will be, respectively, $\tau_u + \tau_c$ and $\tau_{i-n-t} + \tau_c$. Clearly, if τ_c is large, τ_{i-n-t} becomes negligible and thus, the impact is small. To have a better idea of the effect, it would be interesting to know the average time spent in an object in a industrial application. Unfortunately, this kind of information is either not valid or confidential (at least with our industrial partners).

Some solutions are conceivable for reducing the impact of the monitoring mechanism.

6.4.2 Solutions for reducing the intrusive effect

There are different means to improve the performances of our approach.

[2]In fact, the time for the notifications is $\frac{4}{5}$ to $\frac{3}{4}$ of the time of a normal operation request: a one-way operation implies 3 messages at the ORB level and a two-way operation 4-5 messages.

1. Reduce the number of the notifications: this can be performed, for example, by concatenating the two notifications sent at the *o_outReq* and *o_inReq* points and the one sent at the *o_outRep* and *o_inRep* (this will lead to an invocation time of $\tau_{i-n-t} - 2\tau_i$; 5.5 ms in the same scenario as the one of the measurements). These notifications can later be interpreted by the observer and split into two different types of events. The processing is moved from the Filters to the Observer.

   Another solution for reducing the number of notifications is to distribute the monitoring mechanism, as explained in Section 5.13

2. Reduce the size of the notifications: this can be performed by compressing the data of the notifications.

3. Use a different networking system to send the notification: this can be performed by separating the network for the notifications from the network for the distributed applications. For example the notifications might be sent using a simple socket mechanism instead of using the sophisticated ORB communication system. This might very useful, because measures from our partner Alcatel showed that $\frac{2}{3}$ of the time of a request is spent in the TCP/IP stack.

6.5 How to express properties with MOTEL

Let us examine an example to show how we can express properties. The details of the expression of properties can be found in [Die00].

Assuming that we have a service of video on demand, with one user u, a video on demand server vs and a billing server bs. Among the service -operation- that the server offers, as will be described in IDL. There is the
`request_movie(string movie_name)` operation. On the user side, there are also some operations that are offered, e.g., `billing(string movie_name, real price)`. On the billing server there is the operation
`request_billing(string movie_name, string user)`.

From the informal requirements, we can find the constraint: "when the user has requested a movie it will then be billed". The LTL property expressing the same is:

$$\Box(\odot(p\_outreq, (u, vs, request\_movie, (movie\_name))) \rightarrow \\ \Diamond \odot (p\_inreq, (bs, u, billing, (movie\_name, *))))$$

When looking at the MSC of the application, we can even refine (e.g., to help the identification of a fault in the case of the violation of the previous property) the property as:

$$\Box(\odot(p\_outreq, (u, vs, request\_movie, (movie\_name))) \rightarrow \\ \Diamond \odot (p\_inreq, (vs, bs, request\_billing, (movie\_name, u)))) \\ and \\ (\odot(p\_outreq, (vs, bs, request\_billing, (movie\_name, u))) \rightarrow \\ \Diamond \odot (p\_inreq, (bs, u, billing, (movie\_name))))$$

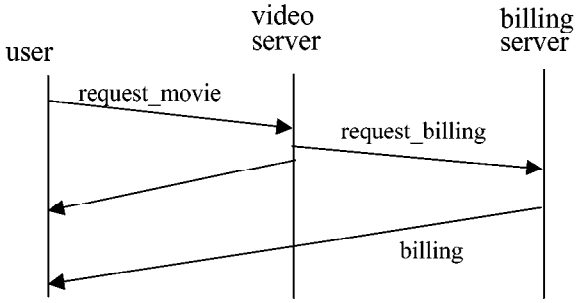


Figure 6.13: MSC of the video on demand service

This second property states that: "when a movie is requested, the video server requests a billing and, if a billing is requested then the billing is send to the user".

6.5.1 Property patterns

Manipulating LTL formula requires some experience and it is sometimes difficult to express what is evident in a natural language under the form of a formal specification.

In [Kir96], Kirkwood describes an approach in which the system is implemented in LOTOS and temporal logic is used to formulate properties that the LOTOS implementation should satisfy. One of the problems he pointed out is that it is sometimes difficult for the specifier to be sure that the desired property has been captured. We completely agree with this remark. It is not always easy to come up with a temporal logic formula for complex properties. While many properties can be specified relatively easily, there are some more complex properties that require a considerable experience in developing LTL formulas. This problem could be solved in the following way. Deeper investigation, would be needed to identify the property classes that occur often in services.

Recent work performed by Peper, Gotzeim et al. [PGK97] use the "pattern" paradigm to classify the set of different properties that are useful in practice, and thus facilitate the expression of properties by giving a library of properties. By identifying several classes, it would be possible to construct tools that help the property specifier to choose the right property structure. He might even be unaware of the fact that temporal logic is behind the representation of the property he expresses.

Typically, a property pattern has the following form:

Name The name of the pattern

Intention The informal specification of the kind of requirement expressed by the pattern

Example An example of the use

Definition The definition of the pattern using a FDT (LTL in our case).

With this method, the tester is able to refer to a library of predefined patterns

that match most common useful properties.

6.6 Use of MOTEL in Industry

Since the beginning stages, our work has given rise to industrial interest; first of all by motivating our two industrial partners, Swisscom and Alcatel, to partially fund our work, secondly by patenting the invention [LDH+99b] reported in this dissertation and thirdly by introducing MOTEL as a development tool in the distributed platform PERCO developed at Alcatel. Our work has been perceived as one solution to bridge the gap between academic research and industrial use of formal techniques.

Let us, in Section 6.6.1, explain more in detail the industrial use of MOTEL in an industrial environment by an illustrative example, for the detection of Feature Interactions. In Section 6.6.2 we present the integration of MOTEL in Alcatel's PERCO distributed platform and finally in Section 6.6.3 some other uses of MOTEL.

6.6.1 MOTEL used for the Detection of Feature Interactions

Feature interactions (FI) happen when different features of a software system interfere with each other. In [BDC+89], a FI is defined by

> *"... a behavior through which one feature inhibits or subverts the intended execution of another feature or another instance of itself, or creates joint execution dilemmas that are not resolved within the individual feature instances".*

Two services can work fine when executed separately, but can interact strangely when executed at the same time on the same infrastructure. The problem of FI is getting worse as the number of services and their speed of delivery increases.

Let us take a scenario to illustrate the use of MOTEL in the context of FI. An operator or a service deploying company A gives a mandate to two other companies, B and C for the implementation of two value-added services, that will be available on the same network infrastructure. The logic and management of the services are ensured by objects on a CORBA platform. Within this environment, the users have different basic services: chat, videoconferencing, phone call, etc. The infrastructure is depicted in Figure 6.14, the logic is ensured by CORBA objects, whereas the communication connections are directly established through the network.

The system is designed with one server, called Service Server (SS), that deals with all the services (activation, management, etc.), another server, called Connection Performer (CP), that is in charge of setting/establishing the different connections, and the user software (for the user's terminal). That software provides a GUI interface to the user in order to hide the implementation details and another part which a usual "middleware object".

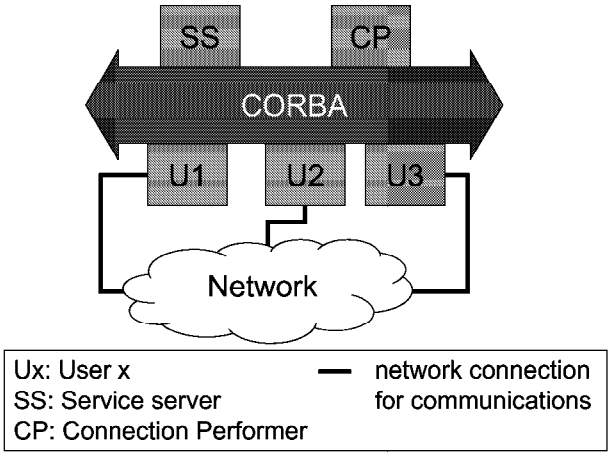


Figure 6.14: Example of system infrastructure

The operator A asks the company B to implement a value-added service *Originating Call Screening (OCS)*. This service allows a subscriber to specify that outgoing calls[3] be either restricted or allowed, according to a screening list, and optionally, by time of day control.

The company C is in charged of the implementation of *Call Forwarding (CF)*. This service allows a user to have his incoming call/connections forwarded to another terminal. Since only the operator has detailed information about all the services deployed in the network, FI detection and resolution are not issues that can be handled by a single exterior company that has developed one of the services. No verification and testing at any single service developing company can detect FI, unless all the deployed services with their specifications and/or implementations are available to the developing company. Given a competitive market, there might be only a limited willingness to share information about the developed services.

The typical use of MOTEL in this scenario is showed in Figure 6.15.

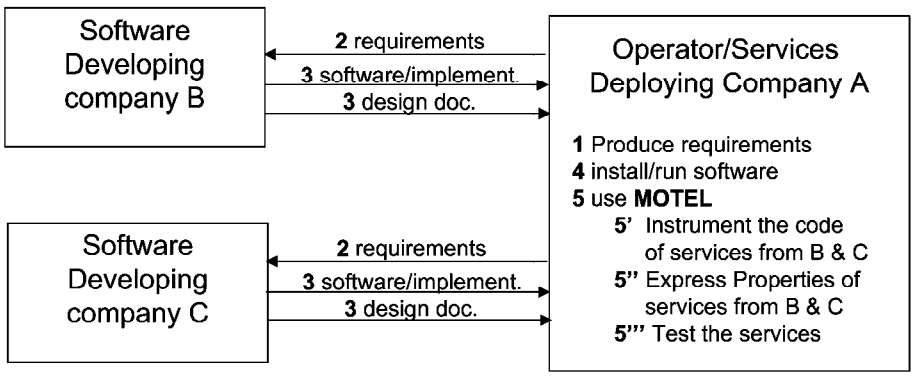


Figure 6.15: MOTEL helping the detection of FIs

[3]In this example, we consider a call in a general meaning: a connection.

The operator will give the requirements to the developing companies. These companies will implement the services and delivering the software as well as the design documentation. The operator will then use MOTEL to instrument the code of the to services in order to be able to monitor their behavior when they will be running.

By looking at the design documents and at the requirements. the tester expresses different properties that each services must not violate. In our scenario, in the design documents from the developer companies we find the information in order to activate/deactivate the different services. For this, the users invoke, by the means of the GUI, operations on the interface of the Service Server (SS):

`ocs_start(u)`: to start OCS for user u,

`ocs_stop(u)`: to stop OCS for user u,

`cf_start(u)`: to start CF for user u,

`cf_stop(u)`: to stop CF for user u.

From the requirements of the OCS services, considering that user u_1 wants to use OCS to prevent any call/connections to user u_2, the property, that we call the OCS property, would be expressed by the tester:

$$\Box(\odot(o\_outreq, (u_1, ss, ocs\_start, (u_2))) \rightarrow$$
$$\neg \odot (o\_outreq, (ss, cp, connect(u_1, u_2)))\, \mathcal{U} \odot (o\_outreq, (u_1, ss, ocs\_stop, (u_2))))$$

Informally, this property states that "as soon as the user u_1 start the OCS service by using `ocs_start(u2)` with the parameter u_2, then he cannot be connected to user u_2 unless he had previously stopped the OCS service by invoking `ocs_stop(u2)`; this always has to be true".

For the testing, the tester starts MOTEL and the services. Typically, it first tests OCS only, and then introduces CF. He uses a scenario, such as in Figure 6.16, where the two services are used at the same time. In that scenario, user u_1 wants to prevent all connections to user u_2 (number 1 on Figure 6.16) and user u_3, forwards all incoming call/connections to user u_2 (number 2). Later, user u_1 asks for the establishment of a connection with user u_3 (number 3). Then, the FI problem will appear. Depending on the implementation of the SS (number 4), the SS will ask the CP to connect user u_1 with user u_2, since u_3 has activate CF to user u_2 (number 5) and a connection is established (number 6). By expressing the OCS property in MOTEL, the problem would have been detected. At number 5, the OCS property is violated and an error is reported to the tester by MOTEL. Note that in the more general case, the property would have been written like:

$$\Box(\odot(o\_outreq, (x, ss, ocs\_start, (y))) \rightarrow$$
$$\neg \odot (o\_outreq, (ss, cp, connect(x, y)))\, \mathcal{U} \odot (o\_outreq, (x, ss, ocs\_stop, (y))))$$

and x would be "instantiated" for all the users that start the OCS service.

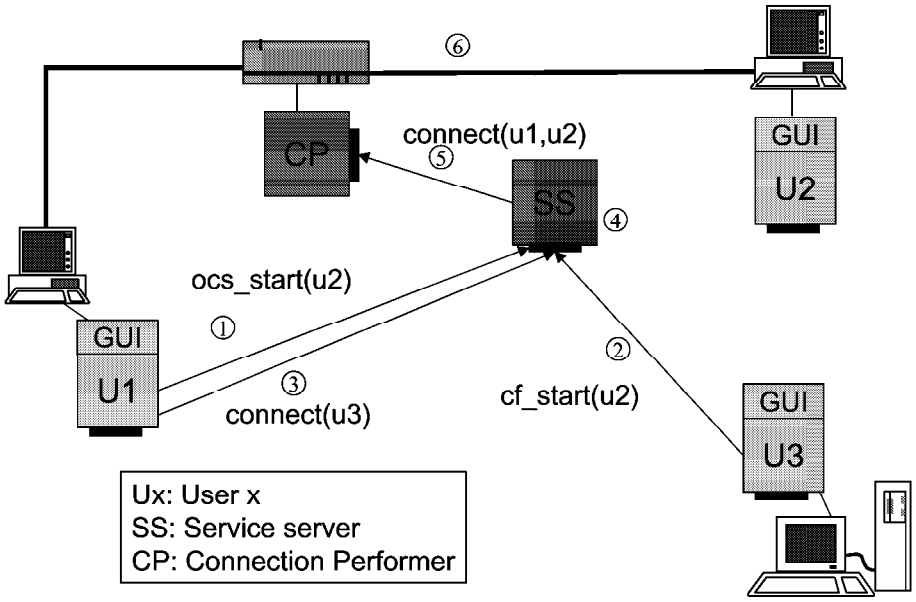


Figure 6.16: Example of FI

Each time a new service is introduced in the network, the service provider activates the properties for the existing services and for the newly developed service and observes whether or not any service misbehaves. It is interesting to note that MOTEL can be used in the testing environment of the operator as well as on the deployment infrastructure. As the system under scrutiny is the implemented system, all the details of the implementation are present. MOTEL can be used at the operational stage, when a problem is encountered by simply activating the instrumentation. Our solution facilitates the testing process for complex services even if only limited information about the actual implementation is known. More specifically, the tester by the operator does not need to understand all the details of the implementation of the two services developed by the developer companies.

MOTEL can also be given to the service deployment companies as a tool for testing the services before the delivery to the operator.

6.6.2 MOTEL integrated in PERCO

The PERCO approach is to take advantage of the potential of UML to integrate the application and its environment at a high level. It makes a broad use of CORBA and its integration within UML to solve the aspects related to portability distribution, reliability, openness and heterogeneity.

The architecture is presented in Figure 6.17. PERCO can be considered as a CORBA extended:

- fault tolerance,
- load balancing,

- real-time.

As PERCO makes an extensive use of CORBA and adds requirements and directions for future evolutions, Alcatel uses this experience to be involved in the CORBA standardization process, notably for minimum CORBA, Realtime CORBA and Fault-Tolerant CORBA.

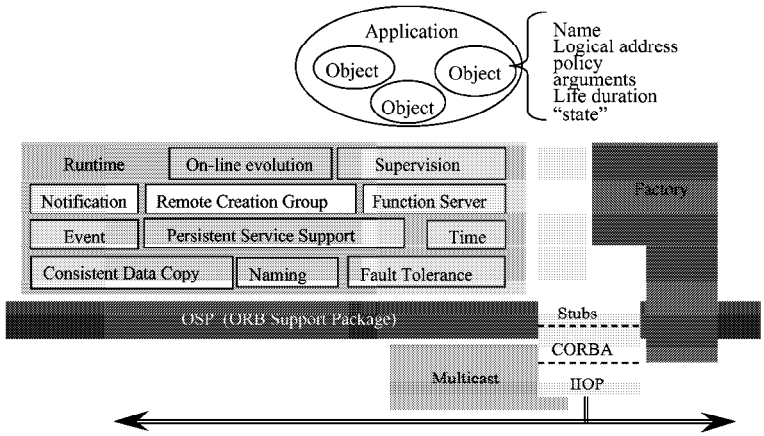


Figure 6.17: The PERCO platform

An example of the use of PERCO is the provision of an air traffic management application developed conjointly by Alcatel and Thomson.

6.6.2.1 Use of MOTEL in PERCO

The use of MOTEL in PERCO is illustrated in Figure 6.18.

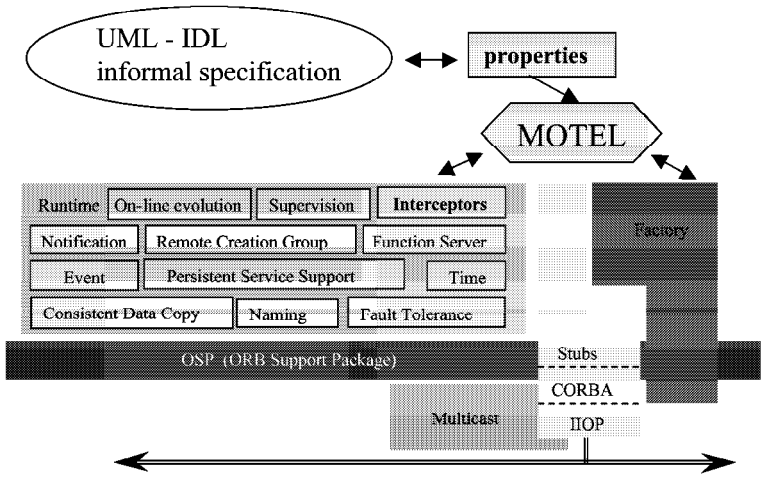


Figure 6.18: The PERCO platform

MOTEL is introduced in PERCO as an add-on to the platform and will be de-

livered to the customers. Customer using the PERCO platform to develop services will have direct access to the technology of MOTEL to help them test and diagnose their developments. In this sense, MOTEL is a nice complement to the features that PERCO offers and CORBA does not: fault tolerance, load balancing, etc.

6.6.2.2 PERCO specificities

PERCO has some specificities that makes the use of MOTEL even more efficient. When deploying an application with the help of PERCO, the user has to write a *configuration file*. The configuration file is used to describe an application's deployment and source code. This file is parsed by PERCO to produce PERCO code. The configuration file contains the following information:

- Policies: gives the priority of threads and the concurrency level,
- Events: gives the IDL file where the events' interfaces can be found,
- Classes: gives the description of the different classes of the system,
- Objects: gives the description of the different objects in the system,
- Programs: gives information of the OS used, the policies, where to find the source code, etc.
- Processes: gives information about the process launching, arguments, directory, etc.
- Hosts: indicates which process run on which machine
- Systems: give the name of the application, the hosts, etc.

When the configuration file is parsed by PERCO, some makefiles are produced to build the application but also some code generation is performed. The code generated is called the *Factory*. The Factory is where the objects are created in the system. The Factory allows objects to be created remotely and their creation and management to be standardized. For example, with the Factory each Object has a logical address/name and not only a physical one. There is one Factory in each process.

6.6.2.3 Expression of properties in PERCO

In PERCO the properties are expressed in the same way as described in Section 6.5. PERCO offers also the help of the Configuration file to express properties.

The next step is the expression of properties directly at the UML specification level. Alcatel has developed a tool to automatically generate code from their UML specification and would like to add the derivation of properties.

6.6.3 Other use of MOTEL

In a more general approach then what has been developed in Chapter 4, 5 and 6, MOTEL can be seen as made of a modular pieces, as depicted in Figure 6.19:

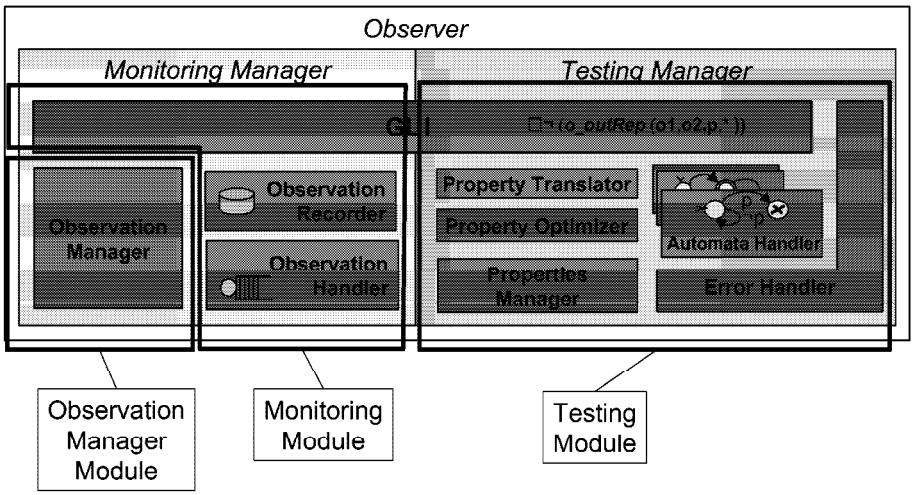


Figure 6.19: MOTEL seen as modules

- The *Observation Manager Module*: this module is the only piece that is directly dependent of the platform used. This module makes the interface between the service under scrutiny and the Observer.
- The *Monitoring Module*: this module is generic.
- The *Testing Module*: this module is only linked to the formal language that is used to express the properties.

This structure of three modules lead to the following comments:

1. *use of MOTEL on all middleware platforms*: the only things that has to be adapted for a specific middleware is the detection of the occurring events and the sending of the notifications. Only parts of the Observation Manager Module has to be adapted. In the case where MOTEL would be introduced in a product line, there will be an Observation Manager Module for each platform, the other modules remain the same.

   MOTEL was developed using Orbix^{TM} from IONA. PERCO is implemented using ORBACUS from OOC. We are currently porting the Observation mechanism from the Orbix filters to the ORBACUS interceptors for the integration of MOTEL in PERCO.
2. *use of MOTEL with a different formal language*: In this work we used LTL as the formal technique to express properties. In Section 6.5.1, we discussed the difficulty to manipulate LTL formula. We can imagine to build different testing modules one for each Formal Technique that are used. We are currently investigating the expression of properties from an UML specification, within

our collaboration with Alcatel. A specific Testing Module might be developed to accept directly UML specifications.

3. *use MOTEL for management, security checking, etc*: In this work, we use the Monitoring mechanism for the purpose of testing. It would be possible to use all the monitoring mechanism for other purposes. For example, for the management of a service, for the detections of alarms, for the checking of security issues. For this, the Testing Module would be replaced by a specific Management or Security Module.

Considering that, we think that there are still a lot of potentials in the use and extensions of the work of this thesis.

6.7 Experience Report of MOTEL

In this section we discuss the strengths and weaknesses of our approach. Because MOTELis still be integrated, there are not a lot of comments on its use (for example, how many errors were found with MOTEL), nevertheless we discuss hereafter the early feedbacks.

6.7.1 Strengths

In the following we will state the strengths of our approach.

- *Independence of implementation language:* The approach is independent of an implementation language and can therefore be applied to a wide range of systems. We have shown the application for the case of CORBA. Only the part that observe the system under scrutiny is implementation dependent.
- *Managability:* In order to make use of the model, a practitioner does not have to understand all the features of formal techniques. He can use guidelines and menus for the expression of properties, all the remaining steps are hidden.
- *Automatic instrumentation:* The method for monitoring is built in such a way that it is possible to perform source code annotation in an automatic manner.
- *Automatic generation of automata:* The generation, optimization of automata from temporal logic formulas can be performed automatically. The translation mechanism produces deterministic automata, which facilitates their handling for testing.
- *Scalability:* The number of properties can be specified according to need. Our approach can therefore be used as an add-on in the normal development process.
- *Tool support:* The property language is supported by a tool that helps the specifier during the specification process. The tool can be used for monitoring only and gives a complete and correct representation of the behavior of the system.
- *Ease of use of the tool:* It offers an easy-to-understand graphical user interface.

- *Hiding of formal methods concepts:* Only the property specification has to be derived manually. All other tasks like the construction of automata from the temporal logic formulae, derivation of observation- and testing code, the selection of relevant filters, the examination of the observation messages and the checking of properties are all automated and are transparent to the tool user.
- *Use of notations well-known to engineers:* The monitored information can be displayed in graphical ar detailed tabular formats. These presentation methods are well known to software engineers.
- *Dynamicity:* All the task that the tool offer to the user are available dynamically. The user can activate/deactivate a specific monitoring at run-time, can add a property to be checked at run-time, etc. This a major feature of MOTEL.

6.7.2 Weaknesses

We identified several weaknesses of our approach, some of which quite obvious, others are more subtle. Some of these weaknesses will be addressed in our following work.

One of the obvious limitations results from our design goal of expressing behavioral constraints independent of a given implementation language. Because an object is only characterized by the behavior that can be observed at its interface, internal object events cannot be handled. Furthermore, the expression of properties in LTL in not always intuitive. Therefore, we are currently studying, with our partner Alcatel, the possibility to automatically derive properties from the UML specification.

The use of a central Observer to receive all the notifications from the objects of the system can provoke a bottleneck in the network and thus have some damages on the performances. We already discussed the intrusiveness of our approach and think that the distribution of the central observer might be a good improvement.

As stated in the introduction of this dissertation, with did not covered the generation of test cases and test scenarios. It should be possible to use the information given by the properties and the automata to generate efficient test cases. With this kind of test cases, the input are optimized to increase the probability to reach errors.

6.8 Conclusions

In this chapter we presented our prototype MOTEL. We presented its implementation, how to specify properties with it, and its performances. We also presented how MOTEL can be used in the industry.

The principal achievement of MOTEL is its adequacy with the industrial needs: a powerful tool that integrates well in their environment and practice. Indeed, MOTEL is currently introduced in an industrial platform that is developed by Alcatel. MOTEL hides most of the monitoring/testing mechanism from the user. Only the properties have to be expressed manually by the help of the tool, all the rest is automatically performed (setup of the instrumentation, observation of the system, etc.)

More precisely MOTEL allows for:

- *The expression of properties with the guidance of a GUI.* The tool offers menus and typical properties to ease the expression and to hide as much as formality from the tester. Indeed, the industry is not familiar with the sophisticated formal notations.
- *The expression of properties independent from the implementation.* The properties do not refer to implementation details but to events that can be automatically observed. This feature allows different players to test the application: the company that develops the software, the company that deploys the software.
- *The testing of the implementation.* The properties are expressed independently from the implementation, however MOTEL allows the actual implementation to be checked. The test/verification is not performed on a formal model of the application but on the implementation. In this sense, MOTEL is complementary to other testing methods (e.g., testing at the analysis phase, validation of a complete specification of the system) since it can detect error on the final implementation, which is the more important. With the enable/disable capacity of the monitoring feature of MOTEL, it can be used at the operational stage, as well as at the early testing phase. Indeed, even at the operational stage, the Observation mechanism can be activated and the properties can be checked.
- *The dynamic monitoring and testing.* Beside the fact that MOTEL allows the monitoring and testing at run-time, when the communication services execute, it also allows to chose/change the properties to check, the event to monitor at run-time. This is an interesting feature that adds another capability in monitoring and testing with MOTEL.

A good example of the use of MOTEL is the detection of feature interaction problems. This is usually not an easy task. MOTEL simplifies the process since many features can be added in the system, each with its own requirements, and MOTEL can check all of the requirements at the same time without the need to rewrite a formal specification or express new properties.

We identified some weaknesses of MOTEL. The observation mechanism has some impact on the system under scrutiny, it slows down the system slightly and might change some behavior. We have identified some enhancements in order to reduce the impact: reduction of the number of notifications, distribution of the Observer, etc. This might be the subject of future work.

In order to increase the efficiency of the testing mechanism advocated by MOTEL, it would be ideal to exploit the properties expressed (or the automata) to extract adequate inputs to stimulate the system being tested.

Chapter 7

Conclusions

With the present day's exponential growth of the (tele-)communications market, the explosion of the number of mobile communication customers, and the tremendous growth of the number of IP hosts, ensuring the reliability of communication services is one of the most challenging tasks in today's software engineering.

This Thesis addresses this problem by providing a new method for increasing confidence in the implementation of communication services. We developed a *monitoring and testing* approach to help the developers/deployers of communication services to verify if their implementations conform to the requirements. To solve this problem we:

1. developed and designed a *monitoring* mechanism that is able to observe the events occurring in the middleware at runtime, when the services are running,
2. developed and designed a *testing* mechanism that checks at runtime if properties expressed using LTL and events, are violated by the execution of the application.
3. designed and implemented a prototype *tool*, MOTEL, that is currently introduced in an industrial platform.

For this:

- In Chapter 2, we defined the notion of communication service, and the different tasks involved in the provision of new services. Among these tasks of service engineering, this work focuses on the testing of services. For this, we carefully looked at the infrastructure and environment on which services are deployed. We presented different middlewares that are used for this purpose, with a special emphasis on CORBA. We also present several telecommunications projects in which middlewares are used.
- In Chapter 3, we presented the set of events that was defined with our industrial partners, Swisscom and Alcatel, for the modelling of distributed applications built on top of a middleware. The set of events is defined at the middleware layer and thus allows an abstraction that is independent of the implementation language. In other words, by changing the implementation language, the set of

events modelling the application remains valid.

- In Chapter 4, we presented the method we developed for the monitoring of distributed applications, more precisely communication services. We explained how it is possible to detect the events in the running software. We defined a method to automatically add the instrumentation that is used to report the events to the Observer. For this, we stated a format for the notifications sent to the Observer and defined the architecture and internal working of the Observer. To guarantee the correct ordering of the notification we proposed the use of an ordering algorithm. Finally we analyzed the theoretical intrusivness of the monitoring and we positioned our approach with regards to other works.
- In Chapter 5, we presented our testing approach based on the monitoring method. We first presented the way the tester specifies behavioral constraints using LTL. Then we explained how to translate these formulas into automata in order to be checked at run-time. We show how to obtain deterministic automata and hope to optimize them. We defined the architecture of the testing mechanism to introduce in the Observer. Our testing process is described. Finally, we positioned our approach with regards to other works.
- In Chapter 6, we presented our prototype MOTEL. We focused on its implementation, how to specify properties with it, what are the performances. We also presented how MOTEL can be used in the industry. We explained how MOTEL is currently being introduced in the industrial platform PERCO of Alcatel. Finally, we discussed the advantages and weaknesses of MOTEL.

Overall, the major achievements and novelties of this work are:

- *A monitoring and testing method/mechanism for middleware based applications.* We used a set of generic events that capture the behavior of middleware based application. We developed a monitoring mechanism that allows on the final implementation, the observation at run-time of these events, regardless of the implementation language. We developed a testing mechanism that is able, based on properties expressed using the events, to check if the behavior of the system under test conforms to the requirements. This is performed at run-time based on the observation of the system. Our approach is innovative in several aspects: the setup of monitoring and testing can be changed dynamically at run-time and this does not require some manual instrumentations of the code; the instrumentation necessitated for the detection and reporting of events is added automatically; our method is not specific to an implementation language or a middleware.
- *A method that fits in software development.* Our monitoring and testing mechanism fits well in the development process because it does not add noticeable new steps for the developer. Most of the mechanisms are automatic. The instrumentation of the code to detect events, the report of the events to the central observer with notifications, the reordering of the notifications, the translation of properties into automata, the optimization of the structure of the automata, the checking for the violation of properties, etc. are hidden to the user. Thus, he can focus on the principal: the expression of behavioral constraints based

on the requirements. This is an interesting feature because it avoid the specification, using a formal techniques, of the entire system and services to test. It also avoids the state space explosion problem that occurs with a lot of testing methods based on formal methods.

- *A method that is efficient for industrial use.* We developed a tool, MOTEL, according to our monitoring and testing methods. This tool hides much of the formality to the specifier of properties. The tester does not need to be a specialist of formal methods to be able to specify properties. This is an important point, since few industrials are familiar with the use of Formal methods. Our tool offers guidance for the specification of properties. Furthermore, the tester does not need to understand the source code of the application being tested since the properties are independent from the implementation language, but nevertheless check on the final implementation. Thanks to the dynamic activation/deactivation features of MOTEL, it can be used at the testing stage and operational stage.

In this thesis we put together many concepts from various domains (distributed systems, object-oriented development, CORBA, communication services, formal method Temporal Logic, automata theory, monitoring, testing). By enhancing, developing and combining, and integrating them together, we have reached a high level of integration. This provides a new, powerful and useful monitoring and testing method for middleware based applications. MOTEL can be seen as an all-in-one tool for the monitoring and testing of communication services.

In this thesis, one of the goals was to propose an innovative testing method that can be used *in* and *by* industry to develop communication services.

We believe that there is not a more valuable argument for the application and practical relevance of our our approach in the industry, than the introduction and use of our method and tool in and by the industry. Our partner, Alcatel, is currently integrating MOTEL in the distributed platform PERCO that they are jointly developing with Thomson.

Furthermore, the major ideas developed in this thesis led to an invention, MOTEL that we patented in the USA with our industrial partners Swisscom AG, Switzerland and Alcatel Research, France.

After a few months of using MOTEL in an industrial environment, it will be interesting to perform a detailed investigation of the benefits and experiences report of MOTEL. This will certainly lead to ideas to further enhance the capabilities of our tool. Among the different avenues to follow in order to further improve MOTEL, one is the distribution of the central observer. And the second is the use of the properties expressed for generating test cases that increase the probability to detect errors.

Appendix A

Acronyms

API	Application Programming Interface
CF	Call Forwarding
CLUE	Closed User group Environment
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DFA	Deterministic Finite-state Automata
DII	Dynamic Invocation Interface
DPE	Distributed Processing Environment
EBBA	Event-Based Behavioral Abstraction
FDT	Formal Description Technique
FI	Feature Interactions
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IOR	Interoperable Object Reference
IIOP	Internet Inter-ORB Protocol
IR	Interface Repository
ISDN	Integrated Service Digital Network
LOTOS	Language of Temporal Ordering Specification
LTL	Linear Time Temporal Logic
MOTEL	Monitoring and Testing Tool
NDA	Non-deterministic Automaton
OCS	Originating Call Screening
ODL	Object Definition Language
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open System Interconnections

PERCO	Plate-forme d'Exécution Répartie Commune
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SPOT	Service Pilot on TINA
TINA	Telecommunications Information Networking Architecture
TINA-C	TINA Consortium

Bibliography

[AAM97] E. Al-Shaer, H. Abdel-Wahad, and K. Maly, "A scalable monitoring architecture for managing large-scale distributed multimedia systems," in *Proc of Management of Multimedia Network and Services (MMNS)*, 1997.

[AFFH96] C. Abarca, P. Farley, J. Forslöw, and T. Hamada, *Service Architecture, Version 4.0, TINA-C Deliverable.* TINA-C, October 1996.

[AV97] S. Alagar and S. Venkatesan, "Causal ordering in distributed mobile systems," *IEEE Transactions on Computers*, vol. 46, pp. 353–361, March 1997.

[Bat95] P. Bates, "Debugging heterogeneous distributed systems using event-based model of behavior," *ACM Transactions on Computer Systems*, vol. 13, pp. 1–31, February 1995.

[BDC+89] T. Bowen, F. Dworak, C.-H. CHow, N. Griffeth, G. Herman, and Y.-J. Lin, "The feature interaction problem in telecommunication systems," in *Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, (Bournemouth, UK), 1989.

[BJ87] K. Birman and T. Joseph, "Reliable communications in presence of failures," *ACM Transactions on Computing Systems*, vol. 5, no. 1, pp. 47–76, 1987.

[BKS+96] P. Brutch, A. Karmarkar, G. S., K. Walzel, W. Marti, and P. . U.W., "SMT: A system monitoring tool for DCE," in *Distributed Platforms* (A. Schill, C. Mittasch, O. Spaniol, and C. Popien, eds.), pp. 245–257, Chapman & Hall, 1996.

[Bri88] E. Brinksma, *On the Design of Extended LOTOS - A Specification Language for Open Distributed Systems.* PhD thesis, University of Hengelo, Netherlands, 1988.

[Büc60] J. Büchi, "On a decision method in restricted second order arithmentic," in *Proc. International Congress on Logic, Methodology and Philosophy of Science* (E. N. et al., ed.), pp. 1–11, Stanford University Press, Stanford, CA, 1960.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollinm, H. Gilchrist, F. Hayes, and P. Jeremeas, *Object-Oriented Development: The FUSION Method.* Prentice Hall, 1994.

[CG94] M. Chechik and J. Gannon, "Automatic verification of requirements implementation," in *ISSTA'94*, (Seattle, Washington), 1994.

[COR97] *The Common Object Request Broker Architecture and Specification (CORBA),version 2.1*, Aug 1997.

[Die00] F. Dietrich, *Modelling and Testing Object-Oriented Communication Services with Temporal Logic.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, Communication Systems Division, 2000.

[DKPR96] R. Dssouli, K. Karoui, A. Petrenko, and O. Rafiq, "Towards testable communication software," in *Protocol Test Systems VIII* (A. Cavalli and S. Budkowski, eds.), pp. 237–251, Chapman & Hall, 1996.

[DLH98] F. Dietrich, X. Logean, and J.-P. Hubaux, "Testing temporal logic properties in distributed systems," Tech. Rep. SSC/1997/027, Swiss Federal Institute of Technology Lausanne, Communication System Division, 1998.

[DLKH98a] F. Dietrich, X. Logean, S. Koppenhoefer, and J. Hubaux, "Middleware supoort for monitoring and testing multimedia services," tech. rep., Swiss Federal Institute of Technology, Communication Systems Division, 1998.

[DLKH98b] F. Dietrich, X. Logean, S. Koppenhoefer, and J. Hubaux, "Modelling and testing object-oriented systems with linear-time temporal logic," Tech. Rep. /SSC/1998/011, Swiss Federal Institute of Tchnology, Lausanne, Communications Systcm Division,, march 1998.

[DR96] L. Dillon and Y. Ramakrishna, "Generating oracles from your favorite temporal logic specifications," in *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, October 1996.

[DY94] L. Dillon and Q. Yu, "Oracles for checking temporal properties of concurrent systems," in *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, vol. 19, pp. 140–153, December 1994.

[EHL98] P. Etique, J.-P. Hubaux, and X. Logean, "Service specification and validation for the intelligent network," *Interoperable Communications Network, Balzer*, vol. 1, pp. 41–69, January 1998.

[Eti95] P.-A. Etique, *Service Specification, Validation and Verification for the Intelligent Network.* PhD thesis, Swiss Federal Institute of Technology Lausanne, Telecommunications Laboratory, 1995.

[Fid96] C. Fidge, "Fundamentals of distributed system observation," *IEEE Software*, vol. 13, no. 6, pp. 77–83, 1996.

[FZ90] J. Fowler and W. Zwaenepoel, "Causal distributed breakpoints," in *Proc of 10th International Conference on Distributed Computing Systems*, 1990.

[Gai85] J. Gait, "A debugger for concurrent programs," *Software Practice Exper..*, vol. 15, no. 6, pp. 539–554, 1985.

[Gba95] C. Gbaguidi, "Design and implementation of virtual private network as a distributed application," Master's thesis, Swiss Federal Institute of Technology Lausanne, Electrical Department, 1995.

[Gba99] C. Gbaguidi, *Programmable Architecture for the Creation and Seamless Control of Hybrid Services.* PhD thesis, Swiss Federal Institute of Technology, Communication Systems Division, 1999.

[GGM95] J. Gaspoz, C. Gbaguidi, and J. Meinkoehn, "Vpn on dce: From reference configuration to implementation," in *IS&N, Greece*, October 1995.

[GHHT99] C. Gbaguidi, J.-P. Hubaux, M. Hamdi, and A. Tantawi, "A programmable architecture for the provision of hybrid services," *IEEE Communications Magazine*, vol. 37, July 1999.

[Gla96] R. Glass, "Formal methods are a surrogate for a more serious software concern," *IEEE Computer*, April 1996.

[Gom98] G. Gomez, "Corba in the telco environment - the case at telefonica i&d," *Orbix Journal*, 1998.

[Gri96] D. Gries, "The need for education in useful formal logic," *IEEE Computer*, April 1996.

[GW89] P. Grimont and P. Wolper, *From Modal Logic to Deductive Database*, ch. 4 Temporal Logic, pp. 165–233. Wiley, 1989.

[Har98] "Distributed computing at telecoms." In "Component development strategies", The monthly Newsletter from Cutter Information Corp. on Managing and Developing Component-Based Systems, ed. P. Harmon, vol. VII. no. 10, October 1998.

[HB89] D. Helmbold and D. Bryan, "Design of run-time monitors for concurrent prograns," Tech. Rep. CSL-TR-89-395, Computer Systems Laboratory, Stanford, 1989.

[HKM+94] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle, "Distributed performance monitoring methods, tools and applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 585–597, June 1994.

[Hol91] G. Holzmann, *Design and Validation of Computer Protocols.* Prentice-Hall Int., 1991.

[Hol94] G. Holzmann, "The theory and practice of a formal method: NewCoRe," in *Proceedings of the IFIP World Computer Congress*, vol. I, (Hamburg, Germany), pp. 35–44, North-Holland Publ., Amsterdam, The Netherlands, August 1994.

[Hol97] G. Holzman, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, May 1997.

[IEE97] IEEE The Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard Collection Sofware Engineering*, 1997.

[Inp98] Inprise, *Visibroker for Java, Programmer's Guide.* Version 3.3, 1998. http://www.inprise.com.

[ION98] IONA, *Orbix/OrbixWeb, version 3.1.* IONA Technologies Ltd., 1998. http://www.iona.com.

[J+87] J. Joyce *et al.*, "Monitoring distributed systems," *ACM Transactions on Computer Systems*, vol. 5, May 1987.

[JJ94] C. Jard and G. Jourdan, "Dependency tracking and filtering in distributed computations," Tech. Rep. 851, IRISA, Beaulieu, France, 1994.

[JPO95] L. Jagadeesan, C. Puchol, and J. Olnhausen, "A formal approach to reactive systems software: A telecommunications application in ESTEREL," *Journal of Formal Methods in System Design*, 1995.

[JPP+97] L. Jagadeesan, A. Porter, C. Puchol, J. Ramming, and L. Votta, "Specification-based testing of reactive sofware: Tools and experiments," in *19th International Conference on Sofware Engineering*, May 1997.

[Jub98] H. Jubin, ed., *JavaBeans by example.* Prentice Hall, 1998.

[Kir96] C. Kirkwood, "Specifying properties of basic LOTOS processes using temporal logic," in *Formal Description Techniques VIII* (G. Bochmann, R. Dssouli, and O. Rafiq, eds.), pp. 109–116, Chapman & Hall, 1996.

[KLMR95] B. Kitson, P. Leydekkers, N. Mercouroff, and F. Ruano, *TINA Object Definition Language (TINA-ODL) Manual 1.3.* TINA-C, June 1995.

[Lam78] L. Lamport, "Time, clocks and ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[LBGD99] P. Leveilleé, E. Bertrand, S. Grisouard, and G. Donnan, *The PERCO Platform: runtime user manual.* Alcatel Research, Object Architecture Unit, Marcoussis, France, 1999.

[LBK95] A. Lazar, S. Bhonsle, and L. K., "A binding architecture for multimedia networks," *Journal of Parallel and Distributed Computing*, vol. 30, pp. 204–216, November 1995.

[LDH98] X. Logean, F. Dietrich, and J.-P. Hubaux, "TINA service validation: The ErnesTINA project," in *IEEE ICC Conference*, (Atlanta), June 1998.

[LDH+99a] X. Logean, F. Dietrich, J. Hubaux, S. Grisouard, and P.-A. Etique, "On applying formal techniques to the development of hybrid services: Challenges and directions," *IEEE Communications Magazine*, vol. 37, July 1999.

[LDH+99b] X. Logean, F. Dietrich, J. Hubaux, S. Grisouard, P. Etique, and S. Koppenoefer, "Monitoring and testing of middleware based software applications." pending US patent, June 1999.

[LDKK98] X. Logean, F. Dietrich, H. Karamyan, and S. Koppenhoefer, "On-line monitoring of distributed applications," in *Middleware 98*, September 1998.

[Log96] X. Logean, "Improving confidence in the implementation of services in the intelligent network," in *Proc of EUNICE'96*, 1996.

[LP85] H. LeDoux and S. Parker, "Saving traces for ada debugging," in *Ada In Use, Proceedings of the Ada International Conference*, 1985.

[Mat89] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, (North-Holland), pp. 215–226, Corsnard M. et al. ed., 1989.

[MBHL97] N. Mercouroff, J. Bengtsson, P. Hellemans, and L. Lehmann, "Implementation of services for computer supported cooperative work on TINA : the SPOT project," in *Proc of ISS'97*, 1997.

[MBR+98] K. Murakami, R. Buskens, R. Ramjee, Y. Lin, and T. LaPorta, "Design, implementation, and evaluation of highly available distributed call processing systems," in *28th IEEE International Symposium on Fault-Tolerant Computing*, (Munich), June 1998.

[MCL99] J. Maisonneuve, S. Chabrion, and P. Leveillé, "The PERCO platform," in *ISORC'99, St. Malo, The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing*, Laboratoire Commun Alcatel-Thomson, May 1999.

[MCWB91] K. Marzullo, R. Cooper, M. Wood, and K. Birman, "Tools for distributed application management," *IEEE Computer*, vol. 24, pp. 42–51, August 1991.

[Mil90] D. Mills, "On the accuracy and stability of clocks synchronized by network time protocol in the internet system," *ACM Computer Communications*, vol. 20, pp. 65–75, Jan 1990.

[MP91a] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[MP91b] Z. Manna and A. Pnueli, "Tools and rules for the practicing verifier," tech. rep., Computer Department of Stanford University, June 1991.

[MSS93] M. Mansouri-Samani and M. Sloman, "Monitoring distributed systems," *IEEE Network*, vol. 7, pp. 20–30, Nov 1993.

[MSS94] M. Mansouri-Samani and M. Sloman, *Network and Distributes Systems Management*, ch. 12, pp. 303–347. Sloman, M., addison wesley ed., 1994.

[Obj98] Object Oriented Concepts, Inc., *ORBacus, an open architecure for distributes solutions*. version 3.0, 1998. http://www.ooc.com.

[OMG98] OMG, *The Common Object Request Broker Architecture and Specification (CORBA),version 2.2*. Object Management Group (OMG), February 1998.

[OMG99] OMG, *The Common Object Request Broker Architecture and Specification (CORBA),version 2.3*. Object Management Group (OMG), June 1999.

[Per90] D. Perrin, "Finite automata," in *Formal Models and Semantics, Handbook of theoretical computer science* (J. Leeuwen, ed.), vol. B, ch. 1, pp. 1–57, Elsevier Science Publishers B.V., 1990.

[Per94] G. Perrow, "Monitoring techniques in distributed systems management," Tech. Rep. 421, Computer Science Department, University of Western Ontario, 1994.

[PGK97] C. Peper, R. Gotzhein, and M. Kronenburg, "A generic approach to the formal specification of requirements," in *1st IEEE International Conference on Formal Engineering Methods (ICFEM'97*, (Hiroshima, Japan), 1997.

[QM97] J. Queiroz de and E. Madeira, "Management of corba object monitoring for multiware platform," in *Open Distributed Processing and Distributed Platforms*, pp. 122–133, Chapman & Hall, 1997.

[RS96] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems," *IEEE Computer*, vol. 29, pp. 49–56, February 1996.

[RST91] M. Raynal, A. Schiper, and S. Toueng, "The causal ordering abstraction and a simple way to implement it," *Information Processing Letters*, vol. 39, pp. 343–350, 1991.

[Rug94] P. Ruget, "Cheaper matrix clocks," in *Proc of 8th International Workshop on Distributed Algorithms*, (Terschelling, the Netherlands), Oct 1994.

[San89] S. Sankar, *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* Tech. report no.stan-cs-89-1289, Stanford University, 1989.

[SBC+99] S. Staamann, L. Butty/'an, A. Coignet, E. Ruggiano, U. Wilhelm, and M. Zweiacker, "Closed user group in internet service centers," in *Proceedings of DAIS'99*, IFIP, 1999.

[SES89] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," in *Third Int'l Workshop Distributed Algorithm*, (Berlin), pp. 219–232, 1989.

[Ses98] R. Sessions, *COM and DCOM.* Wiley, 1998.

[Sie96] J. Siegel, *CORBA, Fundamentals and Programming.* John Wiley & Sons, Inc., 1996.

[SK92] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, vol. 43, pp. 47–52, 1992.

[SM90] S. Sankar and M. Mandal, "Concurrent run-time monitoring of formally specified programs," Tech. Rep. CSL-TR-90-425, Computer Systems Laboratory, Stanford University, 1990.

[SMC+96] B. Steffen, T. Margaria, A. Claßen, V. Braun, and M. Reitenspieß, "A constraint-oriented service creation environment," in *PACT'96, 2nd International Conference on Practical Application of Constraint Technology, London, UK*, 1996.

[Som96] R. Sommer, "Wir erleben gerade eine Revolution," *Der Spiegel*, pp. 31–34, January 1996.

[TO99] J. Toga and J. Ott, "Itu-t standardization activities for interactive multimedia communications on packed-based networks: H.323 and related recommendations," *Computer Networks*, vol. 31, no. 3, pp. 205–223, 1999.

[TR] Time-Rover, "http://www.time-rover.com."

[WGS98] W. Wu, R. Gupta, and M. Spezialetti, "Experimental evaluation of online techniques for removing monitoring intrusion," in *2nd SIGMETRIC Symposium on Parallel and Distributed Tools*, (Oregon, USA), Aug 1998.

Index

Xavier Logean

Education/Employment

03/96 - present	**Swiss Federal Institute of Technology, Lausanne** Institute for computer Communications and Applications Communication Systems Department Research and teaching assistant, Ph.D. candidate.
06/95 - 02/96	**Swiss Federal Institute of Technology, Lausanne** Telecommunications Laboratory Department of Electrical Engineering Research and teaching assistant.
10/90 - 04/95	**Swiss Federal Institute of Technology, Lausanne** Department of Electrical Engineering. Diploma in Electrical Engineering (M.Sc.).
09/85 - 07/90	**Lycée Collège des Creuset, Sion, Switzerland** Maturité Fédérale Scientifique (High School degree)

Projects

Ernestine	Verification and Validation of services for the Intelligent Network Joint project with Swiss Telecom PTT, F&E, Bern, Switzerland.
EnesTINA	Creation and Validation of telecommunication services for TINA Joint project with Swisscom AG, Corporate Technology, Bern, Switzerland.
MOTEL	Runtime Monitoring and on-line testing of middleware based communication services. Joint project with Alcatel Research, Marcoussis, France.

Publications

Patent

- X. Logean, F. Dietrich, J.-P.Hubaux, S. Grisouard (Alcatel), P.-A. Etique (Swisscom) and S. Koppenhöfer, *"Monitoring and Testing of Middleware based Software Applications"*, pending US patent, June 1999.

Journals

- X. Logean, F. Dietrich, J.-P. Hubaux, S Grisouard, P.-A. Etique, *"On Applying Formal Techniques to the Development of Hybrid Services: Challenges and Directions "*, in IEEE Communications magazine, vol. 37, no. 7, July 1999.
- F. Dietrich, X. Logean, S. Koppenhöfer, J.-P. Hubaux: *" Modelling and Testing Object-Oriented Distributed Systems with Linear-time Temporal Logic"*, Submitted to the journal Theory and Practice of Object Systems, Wiley.

- P.-A. Etique, J.-P. Hubaux, X. Logean *"Service Specification and Validation for the Intelligent Network"*, Journal of Interoperable Communication Networks (ICON), ed. Dr. Sathya Rao, Balzer Science Publishers, vol. 1, no. 1, January 1998, pp. 41-69.

Conferences

- X. Logean, F. Dietrich, S. Koppenhoefer and H. Karamyan, *"Run-time monitoring of Distributed applications"*, MIDDLEWARE'98, Lake District, September 98.
- X. Logean, F. Dietrich, and J.-P. Hubaux *"TINA Service Validation: The ErnesTINA project"*, Proc. of IEEE ICC'98, Atlanta, June 1998.
- F. Dietrich, X. Logean, S. Koppenhoefer and J.-P. Hubaux, *"Testing Temporal Logic Properties in Distributed Systems"*, Proc. of IFIP IWCTS'98 (International Workshop on Testing of Communicating System), Tomsk Siberia, August 1998.
- X. Logean, J.-P. Hubaux, S. Znaty, *"A Generic Component Model for the Design of Telecommunication Services"*, Proc. IEEE MMNS'97 (Management of Multimedia Network and System), Montreal, Canada, ed. R. Boutaba, A. Hafid, P. Dini, (Chapman & Hall), July 97.
- X. Logean, S. Znaty, J.-P. Hubaux *"A method to ease creation of telecommunications services based on a Generic Component Model"*, Proc. IEEE GLOBECOM'97, Phoenix, Nov. 1997.
- T. Saydam, X. Logean, and S. Znaty, *"A Service Management Architecture"*, ICT International Conference on Telecommunications, Melbourne, Australia, April 1997
- X. Logean, *"Improving Confidence in Service Implementation in an Intelligent Network"*, Proc. EUNICE'96, Lausanne, 1996.
- X. Logean, S. Znaty, B. Dufresne, *"Telecaching over Broadband Networks: The BETEUS Project"*, Virtual Mobility Workshop, Brussels, December 1995.